Data Structures and Algorithms for Engineers

Module 2: Complexity of Algorithms

Lecture 2: Complexity Theory

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Complexity of Algorithms

- Performance of algorithms, time and space tradeoff, worst case and average case performance
- Big O notation
- Recurrence relationships
- Analysis of complexity of iterative and recursive algorithms
- Recursive vs. iterative algorithms: runtime memory implications
- Complexity theory: tractable vs intractable algorithmic complexity
- Example intractable problems: travelling salesman problem, Hamiltonian circuit, 3-colour problem, SAT, cliques
- Determinism and non-determinism
- P, NP, and NP-Complete classes of algorithm

Complexity and Intractability

Tractable and intractable problems

- What is a ”reasonable” running time?
- NP problems, examples
- NP-complete problems and polynomial reducibility

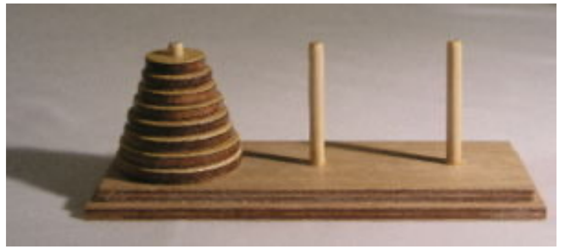
Some elements of the following are adapted from notes by Simonas Šaltenis, Aalborg University

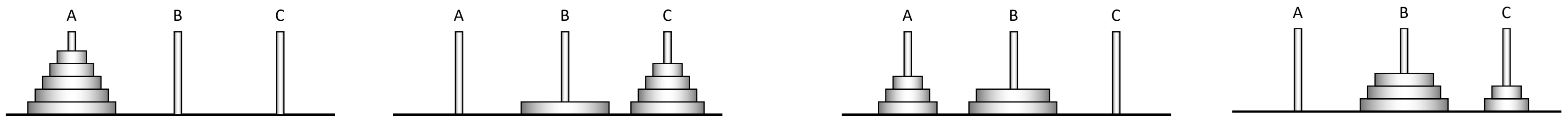
Towers of Hanoi

Goal: transfer all *n* disks from peg A to peg B

Rules:

- move one disk at a time
- never place larger disk above smaller one





Towers of Hanoi

- Can be very hard to find a direct – brute force – solution to the problem of size n
- However, there is a very simple and elegant recursive solution:
  - Assume that we can solve the problem of size n-1, i.e., we can move n-1 disks from one rod to another using a third rod as auxiliary
  - To move n disks from A to B:
    - Move the top n-1 disks from A to C using B (we know how to do this)
    - Move the remaining disk on A to rod B
    - Move the n-1 disks from C to B using A (we know how to do this)
- Total number of moves: $T(n) = 2T(n - 1) + 1$

Towers of Hanoi

- Recurrence relation:

$$T(n) = 2\ T(n - 1) + 1$$
$$T(1) = 1$$

- Solution by unfolding:

$$\begin{aligned} T(n) &= 2\ (2\ T(n - 2) + 1) + 1 = \\ &= 4\ T(n - 2) + 2 + 1 = \\ &= 4\ (2\ T(n - 3) + 1) + 2 + 1 = \\ &= 8\ T(n - 3) + 4 + 2 + 1 = \ldots \\ &= 2^{i}\ T(n - i) + 2^{i-1} + 2^{i-2} + \ldots + 2^{1} + 2^{0} \end{aligned}$$

- the expansion stops when $i = n - 1$

$$T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^{1} + 2^{0}$$

Towers of Hanoi

- This is a **geometric sum**, so that we have

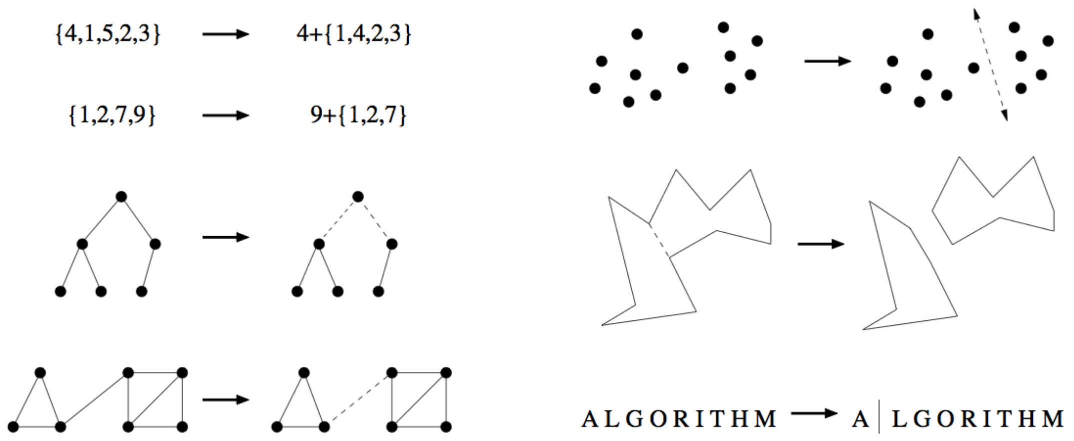
  $T(n) = 2^n - 1 = O(2^n)$

- The running time of this algorithm is exponential (k^n) rather than polynomial (n^k)
- Good or bad news?
  - the Tibetan monks were confronted with a tower of 64 rings...
  - assuming one could move **1 million rings per second**, it would take **half a million years** to complete the process...

Aside: Recursive Programming

Recursion and Recursive Objects

- Many problems can be elegantly described using recursion
- "Learning to think recursively is learning to look for big things that are made from smaller things of exactly the same type as the big thing"



Aside: Recursive Programming

Recursion and Recursive Objects

- The best strategy for developing a recursive algorithm is often to
    - assume you have an algorithm that can give the solution for part of the problem
    - figure what additional work must be done to solve the full problem
    - combine partial solution and additional processing
    - use this new algorithm in place of the assumed algorithm
- In other words, find the recurrence relationship between the full problem and simper components of the problem
- This is a "divide-and-conquer" strategy

Aside: Recursive Programming

- Divide
  - Break the problem into several problems that are similar to the original problem but smaller in size
- Conquer
  - Solve the sub-problems recursively, or,
  - If they are small enough, solve them directly
- Combine the solutions to the sub-problems into a solution of the original problem

Aside: Recursive Programming

Factorial

$n! = n \times (n\text{-}1) \times (n\text{-}2) \times \ldots \times 1$

Also given by the recurrence formula

$$f_n = n \times f_{n-1} \qquad n>0$$
$$f_0 = 1$$

In other words

$$n! = n \times (n\text{-}1)! \quad n>0$$
$$0! = 1$$

Aside: Recursive Programming

```
int factorial(int n) { // assume n >= 0
   if (n == 0)
      return(1);
   else
      return(n x factorial(n-1));
   }
}
```

Aside: Recursive Programming

Fibonnaci Sequence

Given by the recurrence formula

$f_0 = 1$

$f_1 = 1$

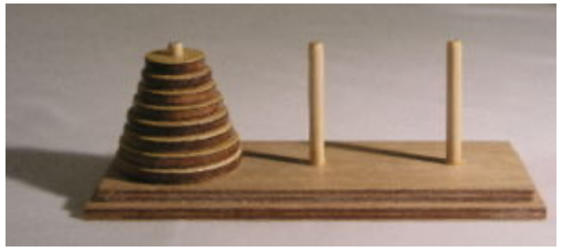
$f_n = f_{n-1} + f_{n-2} \quad n >= 3$

Aside: Recursive Programming

```
int fibonnaci_number(int n) { // assume n >= 0
   if (n == 0 || n == 1)
      return(1);
   else
      return(fibonnaci_number(n-1) + fibonnaci_number(n-2));
   }
}
```

Aside: Recursive Programming

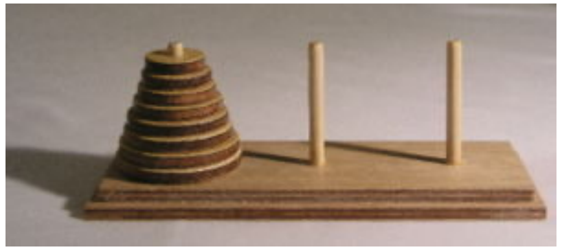
Tower of Hanoi



The objective of the puzzle is to move the entire stack to another peg, obeying the following rules:

- Only one disk may be moved at a time
- Each move consists of taking the upper disk from one of the pegs and sliding it onto another peg, on top of the other disks that may already be present on that peg
- No disk may be placed on top of a smaller disk.

Aside: Recursive Programming



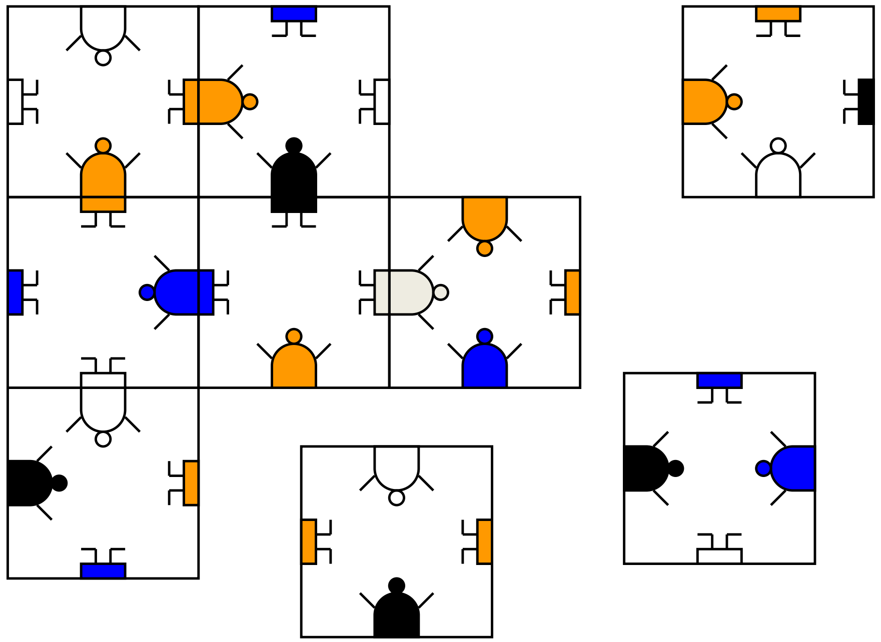
```
void hanoi(int n, char a, char b, char c) {
  if (n > 0) {
      hanoi(n-1, a, c, b);
      printf("Move disk of diameter %d from %c to %c\n", n, a, b);
      hanoi(n-1, c, b, a);
   }
}

…

Hanoi(5, ‘A’, ‘B’, ‘C’);
```

Monkey Puzzle

Are such long running times linked to the size of the solution of an algorithm?

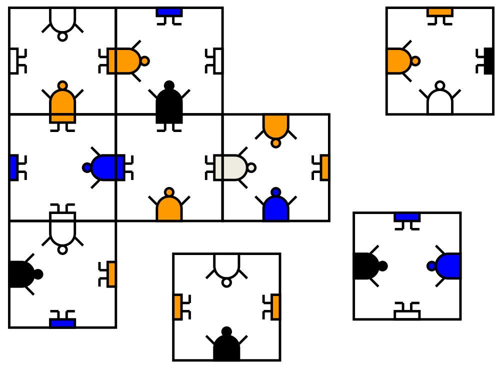
No. To show that, we in the following consider only TRUE/FALSE or yes/no problems – decision problems

- Nine square cards with imprinted "monkey halves"
- The goal is to arrange the cards in 3x3 square with matching halves...



Monkey Puzzle

- Assumption: orientation is fixed
- Does any M x M arrangement exist that fulfills the matching criterion?
- Brute-force algorithm would take $n!$ times to verify whether a solution exists (why?)
  - assuming n = 25, it would take 490 billion years on a one-million-per- second arrangements computer to verify whether a solution exists



Monkey Puzzle

- Assume n, the number of cards, is 25
- The size of the final square is 5 x 5

Monkey Puzzle

Brute force solution:

- Go through all possible arrangements of the cards
- pick a card and place it - there are 25 possibilities for the first placement
- pick the next card and place it - there are 24 possibilities
- Pick the next card, there are 23 possibilities …

Monkey Puzzle

- There are 25 x 24 x 23 x 22 x … x 2 x 1 possible arrangements
- That is, there are factorial 25 possible arrangements (25!)
- 25! contains 26 digits
- If we make 1000000 arrangements per second, the algorithm will take 490 000 000 000 years to complete

Monkey Puzzle

- Improving the algorithm
  - discarding partial arrangements (backtracking & pruning)
  - etc.
- A smart algorithm would still take a couple of thousand years in the worst case
- Is there an easier way to find solutions?
  Perhaps, but nobody has found them, yet …

Complexity and Intractability

- We classify functions as ‘good’ and ‘bad’
- Polynomial functions are good
- Super-polynomial (or exponential) functions are bad

Complexity and Intractability

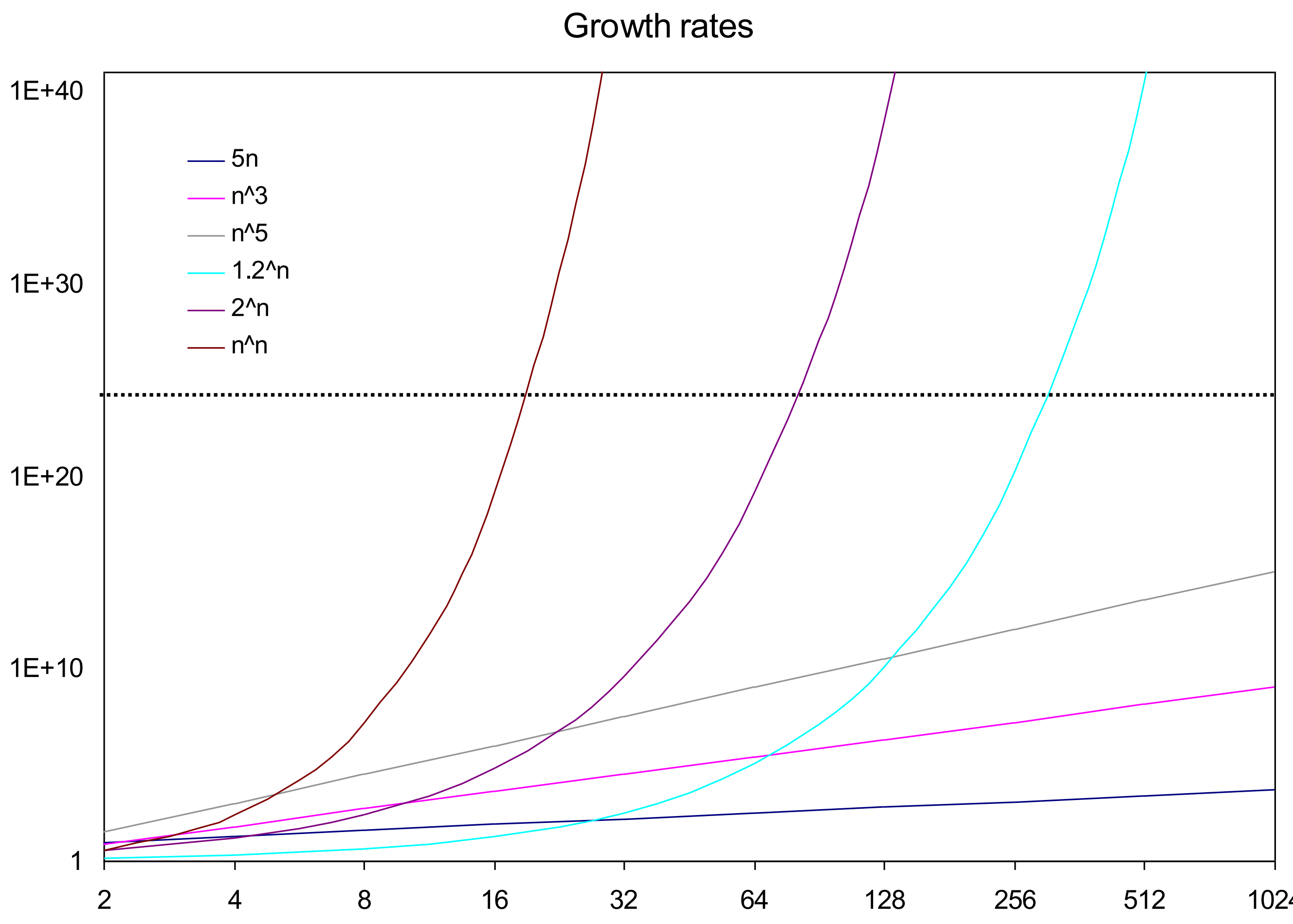
- The order of complexity of this algorithm is $O(n!)$
- $n!$ grows at a rate which is orders of magnitude larger than the growth rate of the other functions we mentioned before

Complexity and Intractability

- Other functions exist that grow even faster,
  e.g. n^n (super-exponential)

- Even functions like 2^n exhibit unacceptable sizes even for modest values of n

Reasonable vs. Unreasonable



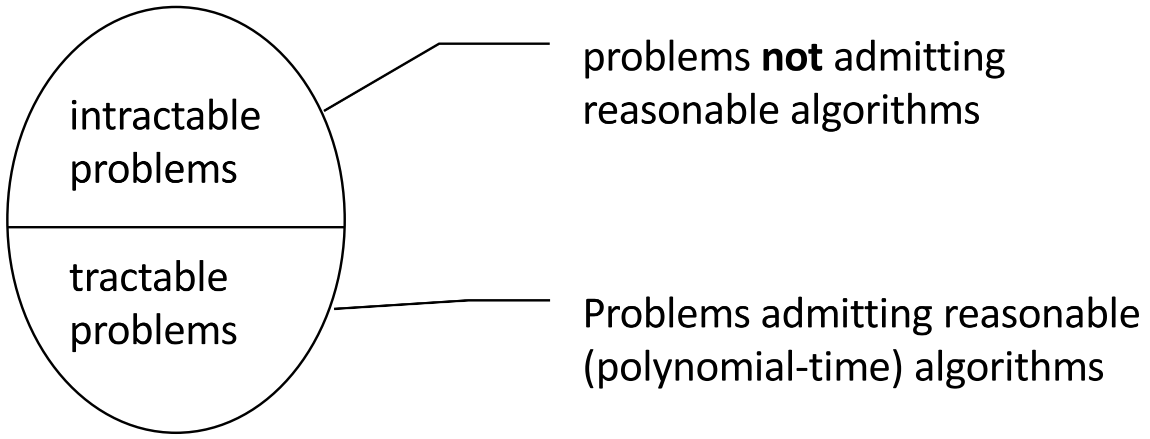
Number of microseconds since “Big-Bang”

Reasonable vs. Unreasonable

	function/ n	10	20	50	100	300
Polynomial	n^2	1/10,000 second	1/2,500 second	1/400 second	1/100 second	9/100 second
	n^5	1/10 second	3.2 seconds	5.2 minutes	2.8 hours	28.1 days
Exponential	2^n	1/1000 second	1 second	35.7 years	400 trillion centuries	a 75 digit-number of centuries
	n^n	2.8 hours	3.3 trillion years	a 70 digit-number of centuries	a 185 digit-number of centuries	a 728 digit-number of centuries

Reasonable vs. Unreasonable

- ”Good”, reasonable algorithms
  - Algorithms bound by a polynomial function n^k
  - Tractable problems
- ”Bad”, unreasonable algorithms
  - Algorithms whose running time is above n^k
  - Intractable problems

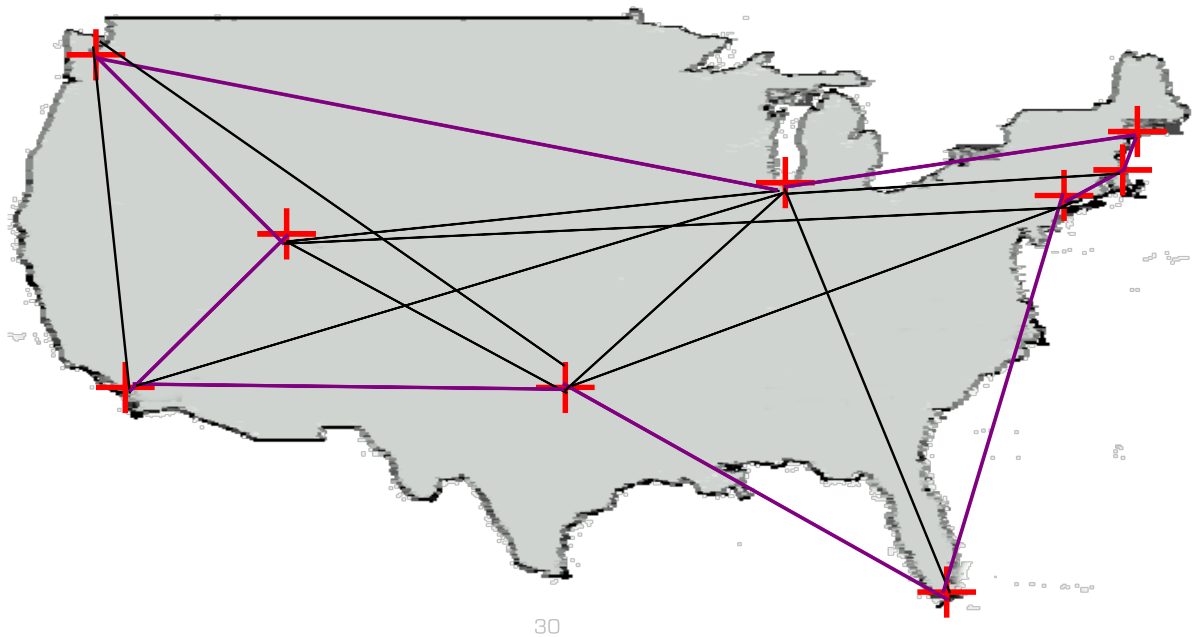

intractable problems
tractable problems
problems **not** admitting reasonable algorithms
Problems admitting reasonable (polynomial-time) algorithms

Just Get a Faster Computer

- Computers become faster every day
  - Doesn’t matter: insignificant (a constant) compared to exp. running time
- Maybe the Monkey puzzle is just one specific one we could simply ignore
  - the monkey puzzle falls into a category of problems called NPC (NP complete) problems (~1000 problems)
  - all admit unreasonable solutions
  - not known to admit reasonable ones…

Travelling Salesman Problem (TSP)

TSP is the problem of a salesman who wants to find, starting from his hometown, a shortest possible trip through a given set of customer cities and to return to its hometown; visiting exactly once each city



Travelling Salesman Problem (TSP)

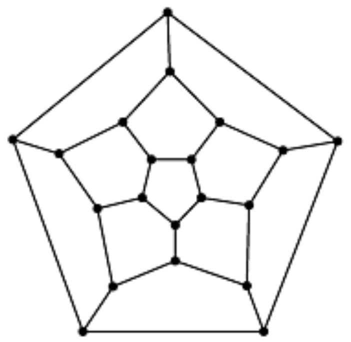
- Naive solutions take $n!$ time in worst-case, where n is the number of edges of the graph

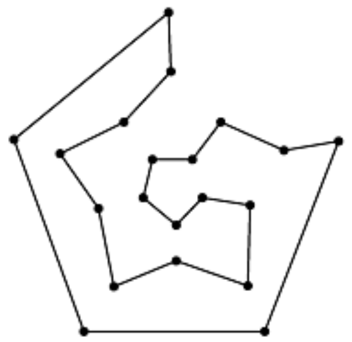
- No polynomial-time algorithms are known
  - TSP is an NP-complete problem

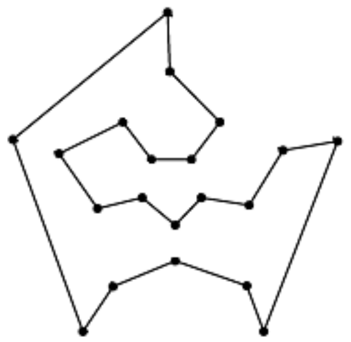
- Longest Path problem between A and B in a weighted graph is also NP-complete

TSP & Hamiltonian

A Hamiltonian circuit for a given graph $G=(V, E)$ consists of finding an ordering of the vertices of the graph G such that each vertex is visited exactly once


Typical Input for HCP


Hamiltonian cycle for the graph

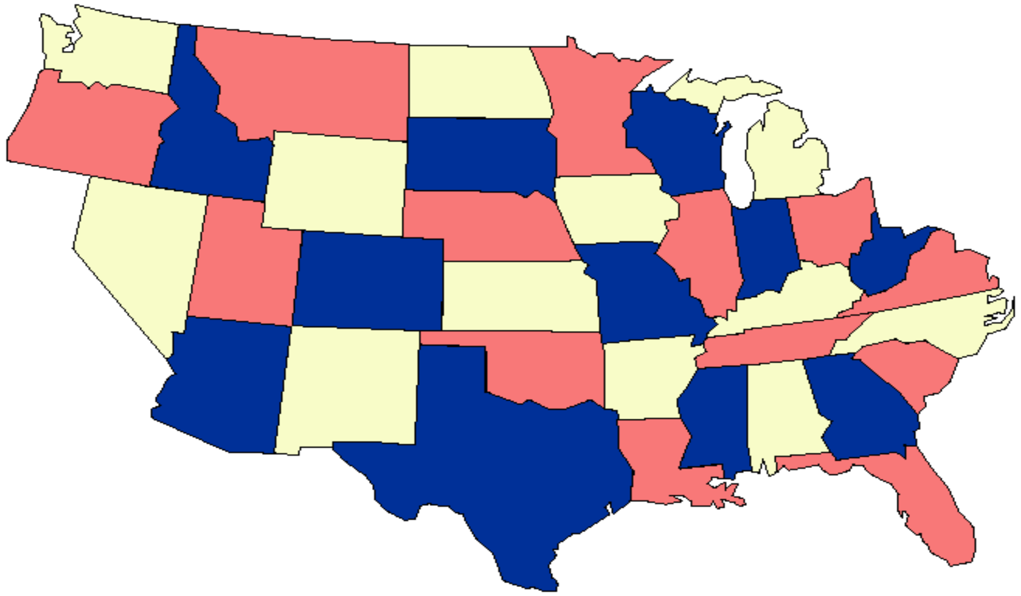

Another Hamiltonian cycle for the same graph in

Coloring Problem

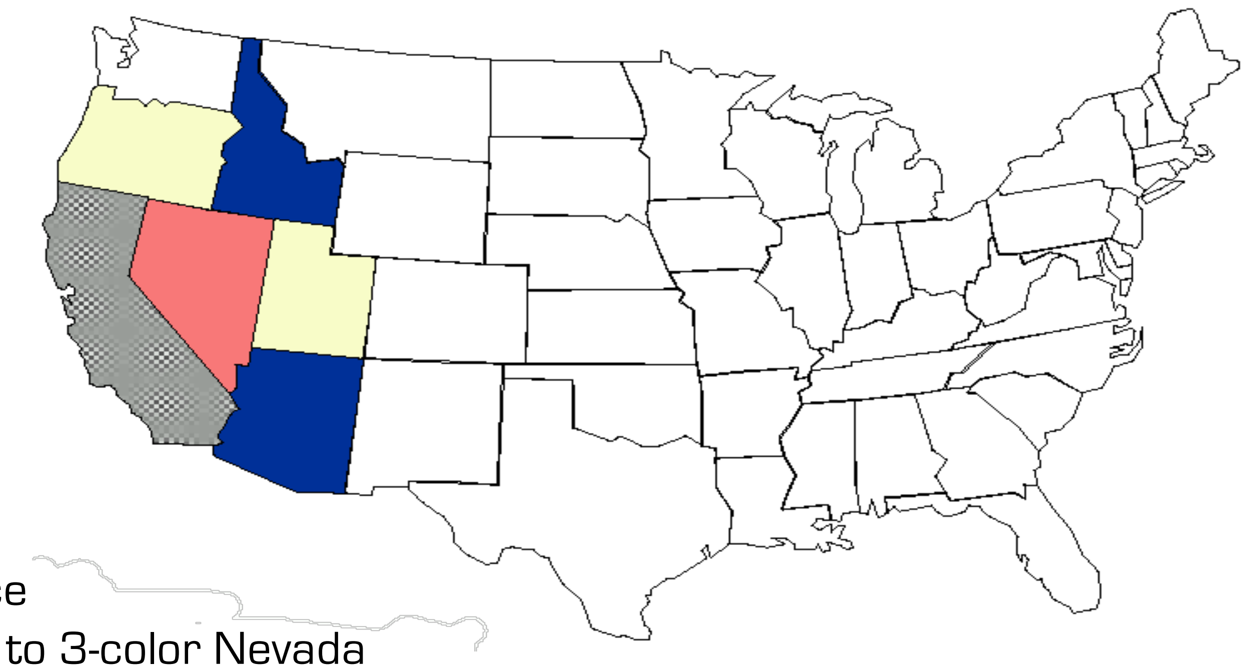
3-colour

- given a planar map, can it be colored using 3 colors so that no adjacent regions have the same color

YES instance



Coloring Problem



NO instance
Impossible to 3-color Nevada
and bordering states

Coloring Problem

- Any map can be 4-colored
- Maps that contain no points that are the junctions of an odd number of states can be 2-colored
- No polynomial algorithms are known to determine whether a map can be 3-colored – it's an NP-complete problem

Satisifiability (SAT)

- Determine the truth or falsity of formulae in Boolean algebra (or, equivalently, in propositional calculus)

- Using Boolean variables and operators

  $\wedge$ (and)
  $\vee$ (or)
  $\sim$ (not)

  we compose formula such as the following

$$\phi = (\sim x \wedge y) \vee (x \wedge \sim z)$$

Satisifiability (SAT)

- The algorithmic problem calls for determining the satisfiability of such formulae

  Is there some assignment of value to x, y, and z for which ϕ evaluates to 1 (TRUE)

  $x = 0, y = 1, z = 0$ makes $\phi = (\sim x \land y) \lor (x \land \sim z)$ evaluate to 1

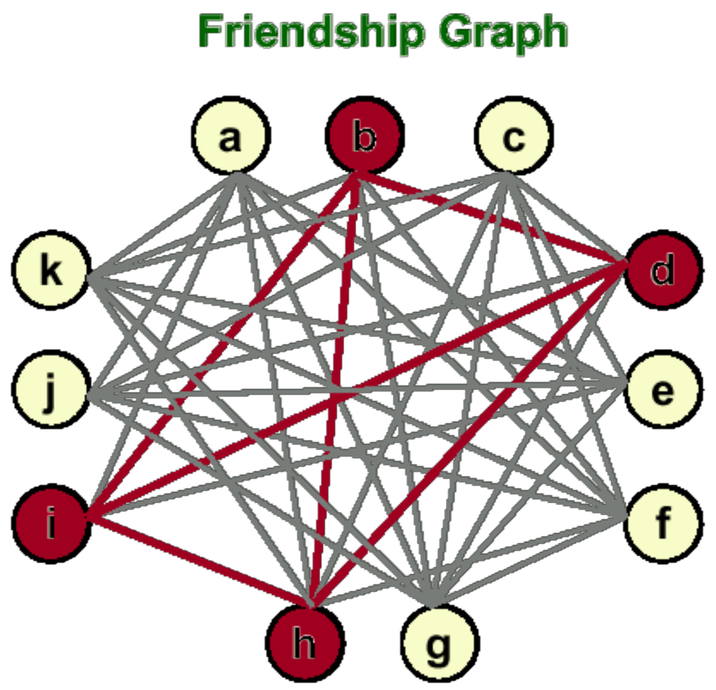
- Exponential time algorithm on n = the number of distinct elementary assertions ($O(2^n)$)

- Best known solution, problem is in NP-complete class

CLIQUE

Given n people and their pairwise relationships, is there a group of s people such that every pair in the group knows each other

- people: a, b, c, ..., k
- friendships: (a,e), (a,f),...
- clique size: s = 4?
- YES, {b, d, i, h} is a certificate



P

Definition of P

- The set of all decision problems solvable in polynomial time on a deterministic Turing machine (i.e., a real computer)

Examples

- MULTIPLE: Is the integer y a multiple of x?
  - YES: (x, y) = (17, 51)
- RELPRIME: Are the integers x and y relatively prime, i.e., is GCD(x, y) = 1?
  - YES: (x, y) = (34, 39)
- MEDIAN: Given integers x_1 , …, x_n , is the median value < M?
  - YES: (M, x_1 , x_2 , x_3 , x_4 , x_5) = (17, 2, 5, 17, 22, 104)

NP

Definition of NP

- The set of all decision problems solvable in polynomial time on a nondeterministic Turing machine

- Important definition because it links many fundamental problems

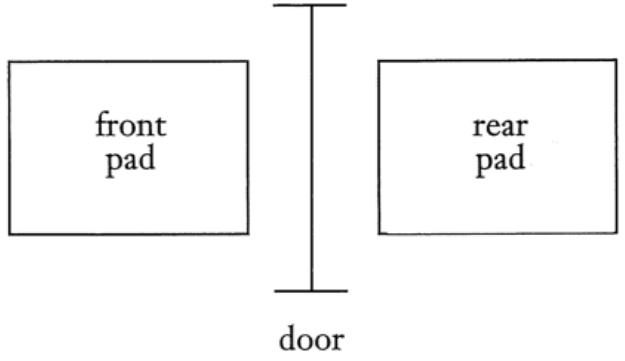
- There are no known polynomial time solutions to NP problems

Aside: Determinism & Non-determinism

Finite Automata

Example: controller for an automatic door

- Front pad to detect presence of a person about to walk though
- Rear pad to make sure it stays open long enough (and avoid hitting someone)

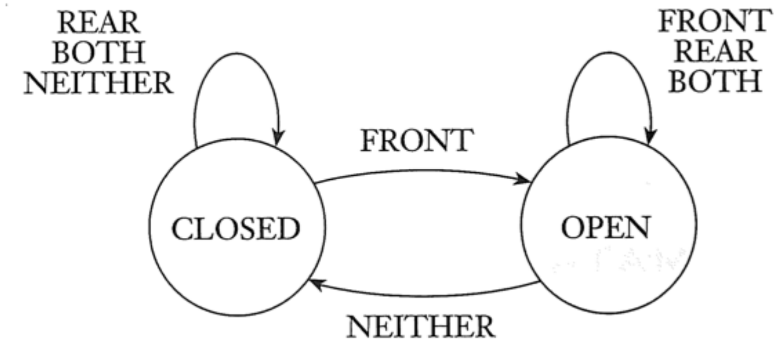


Aside: Determinism & Non-determinism

Finite Automata

Example: controller for an automatic door

- Controller states: OPEN, CLOSED
- Input conditions: FRONT, REAR, BOTH, NEITHER

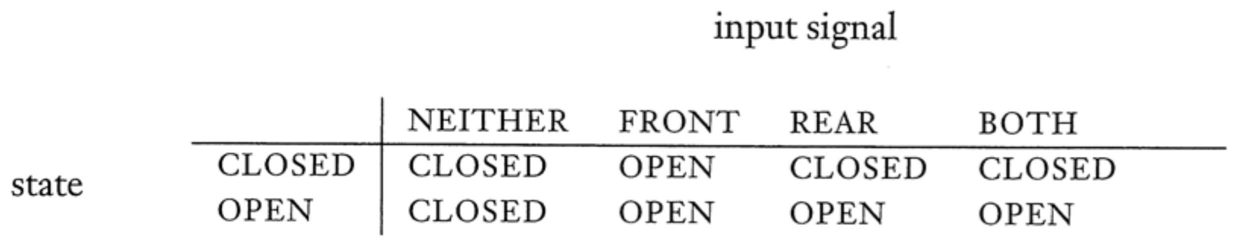


Aside: Determinism & Non-determinism

Finite Automata

Example: controller for an automatic door

- Controller states: OPEN, CLOSED
- Input conditions: FRONT, REAR, BOTH, NEITHER

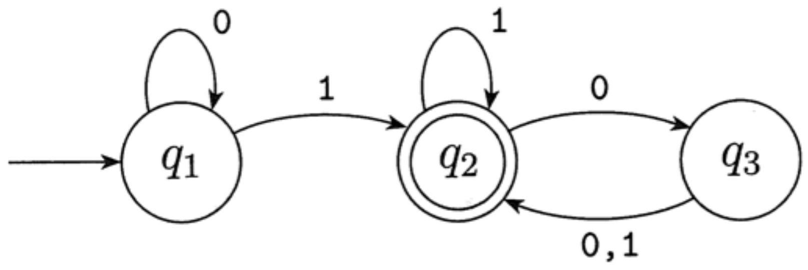


		input signal			
		NEITHER	FRONT	REAR	BOTH
state	CLOSED	CLOSED	OPEN	CLOSED	CLOSED
	OPEN	CLOSED	OPEN	OPEN	OPEN

Aside: Determinism & Non-determinism

Finite Automata

- Automaton receives an input string, e.g., 1101

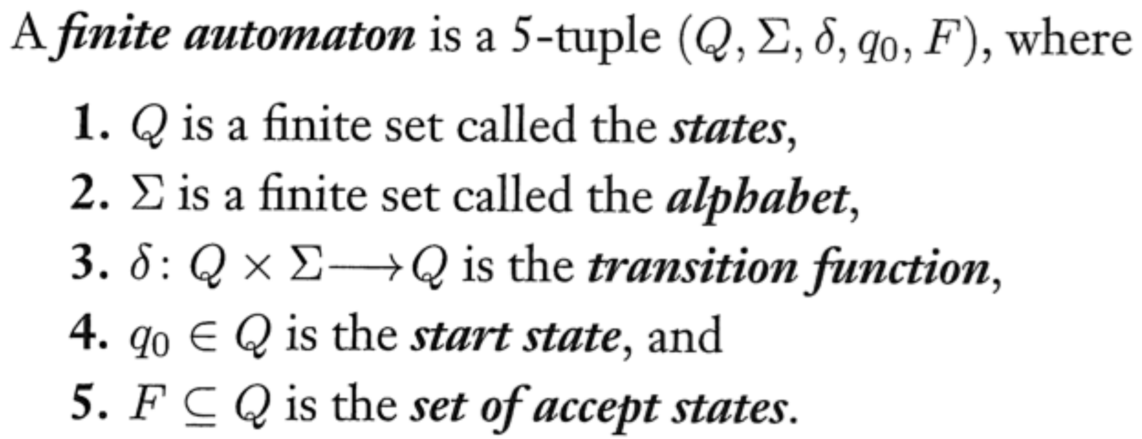


- Output is either **accept** or **reject**
  - accept if in accept state at the end of the input string
  - reject otherwise

Aside: Determinism & Non-determinism

Finite Automata

Formal definition of a finite automaton



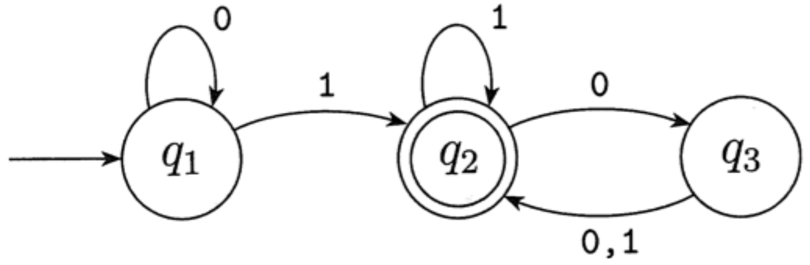
A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

The transition function specifies exactly one next state for each possible combination of a state and an input symbol

Aside: Determinism & Non-determinism

Finite Automata



1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

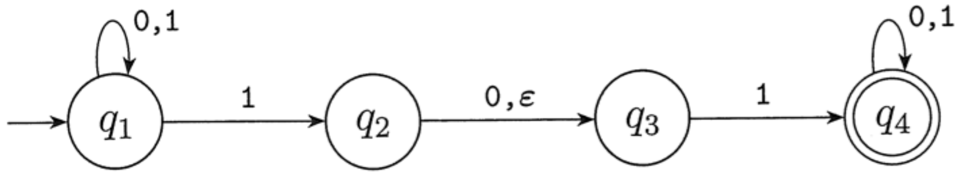
4. q_1 is the start state, and
5. $F = \{q_2\}$.

$\delta: Q \times \Sigma \to Q$

Aside: Determinism & Non-determinism

Non-determinism

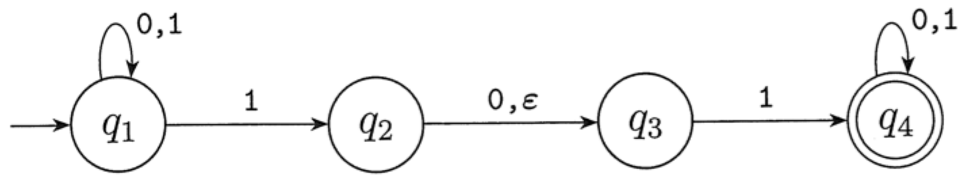
- Deterministic computation:
  - When a machine is in a given state and reads the next input symbol, we know what the next state will be
- Nondeterministic computation:
  - Several choices may exist for the next state
- DFA: deterministic finite automata
- NFA: nondeterministic finite automata



Aside: Determinism & Non-determinism

Nondeterminism

- DFA: deterministic finite automata
  - Exactly one exiting transition arrow for each symbol in the alphabet
- NFA: nondeterministic finite automata
  - Zero, one, or many exiting arrows for each alphabet symbol
  - Transition may also be labelled ε (transition on no symbol)
    zero, one, or many arrows may exit from a state with the label ε

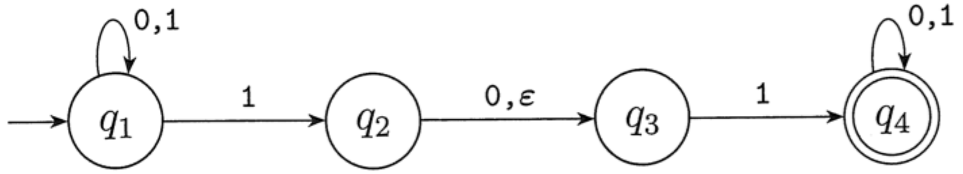


Aside: Determinism & Non-determinism

Nondeterminism

How does an NFA compute?

- If you read a symbol for which there is more than one transition (arrow)
- **The machine splits into multiple copies of itself and follows all of the possibilities in parallel**
- If there a subsequent choices, the machine splits again
- **If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy dies**

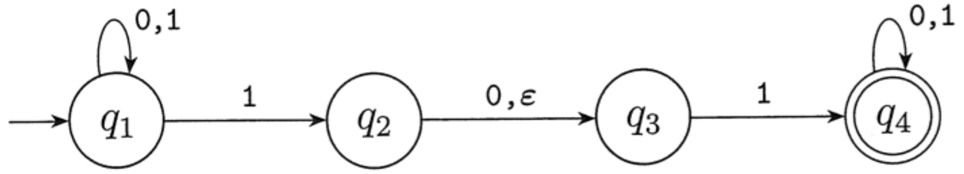


Regular Languages

Nondeterminism

How does an NFA compute?

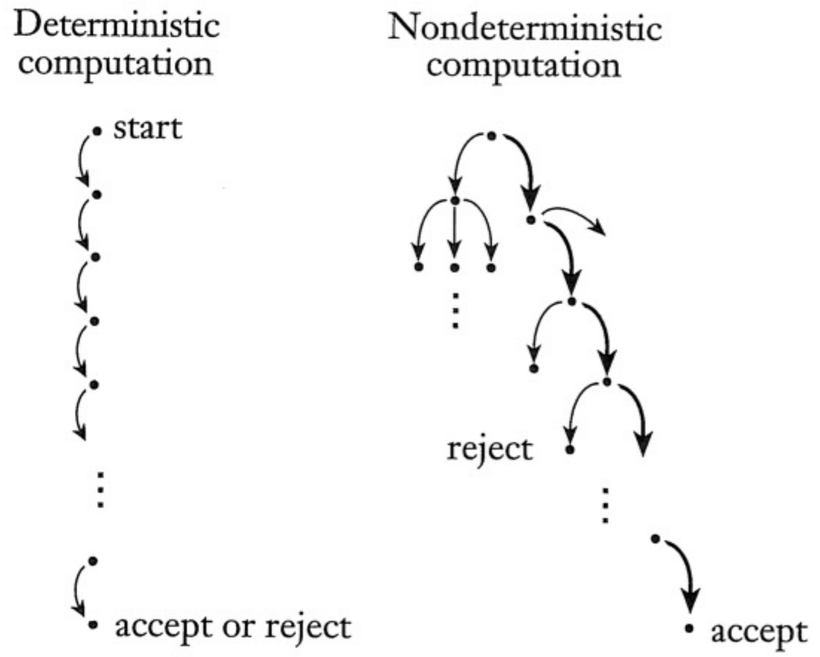
- If a state with an ε is encountered, **without reading any input**, the machine splits into multiple copies, one following the ε–labelled arrows, and one staying at the current state



Aside: Determinism & Non-determinism

Nondeterminism

- How does an NFA compute?
  - Nondeterminism is a kind of parallel computation, with multiple independent processes/threads running concurrently
  - If at least one of the processes/threads accepts, the entire computation accepts



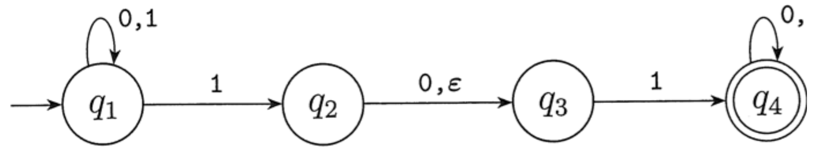
Aside: Determinism & Non-determinism

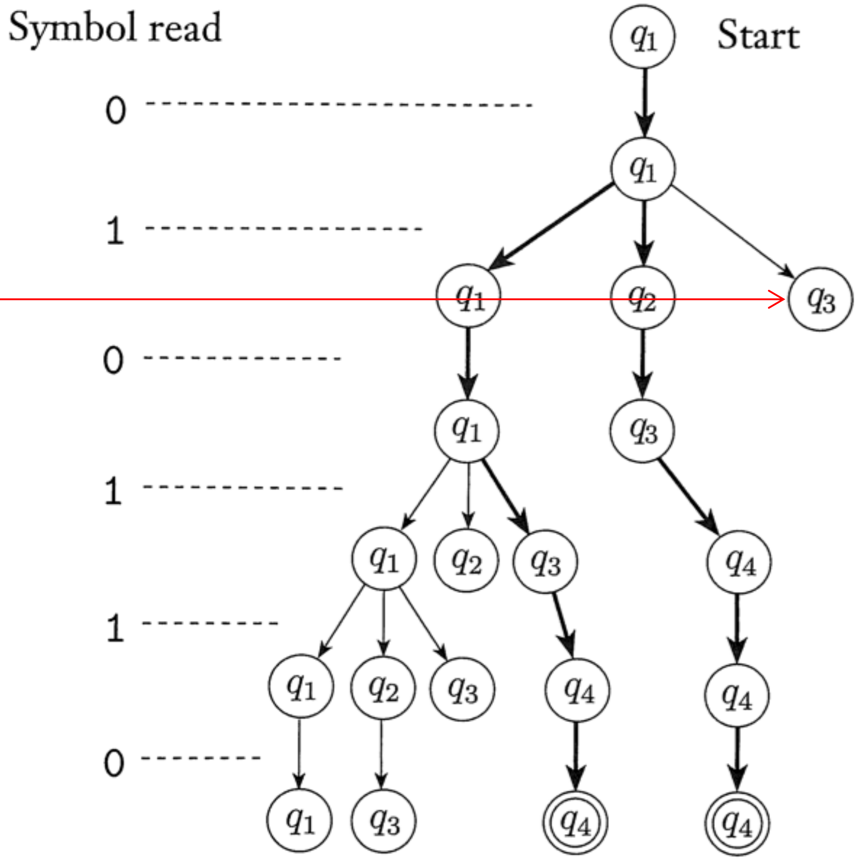
Nondeterminism

Computation for input 010110

Where does this q_3 come from?

Split in q_2 because of ε–labelled arrow

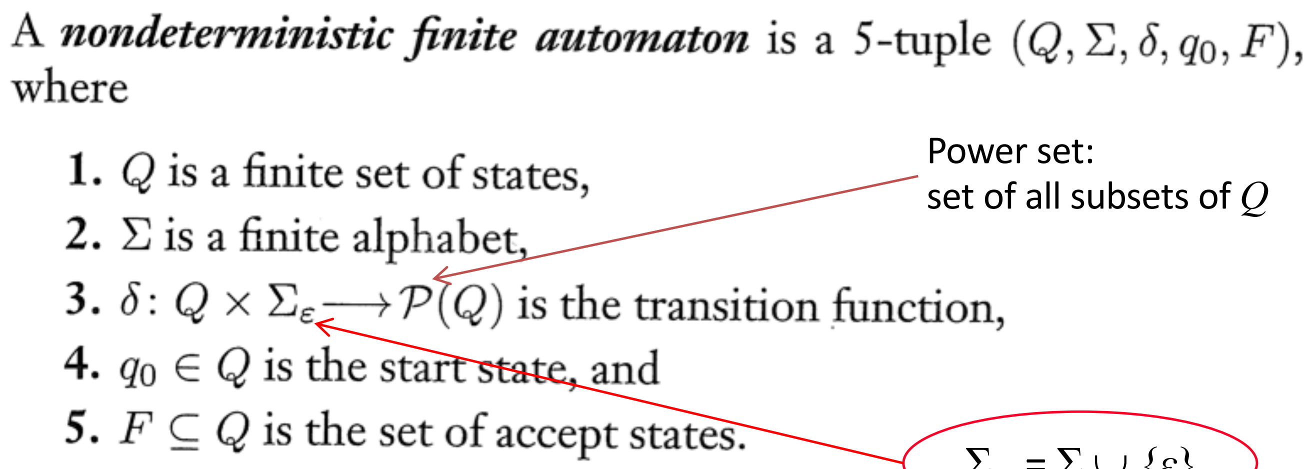




Aside: Determinism & Non-determinism

Nondeterminism

Formal definition of an NFA



A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

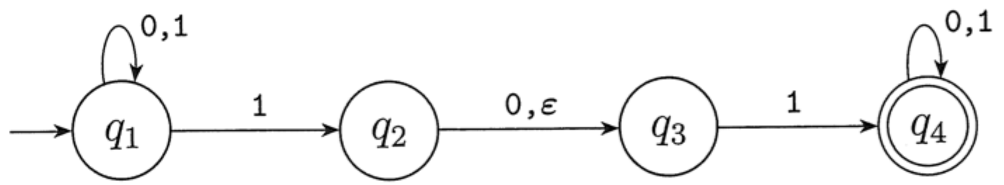
1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta \colon Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Power set:
set of all subsets of Q

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

Aside: Determinism & Non-determinism

Nondeterminism



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

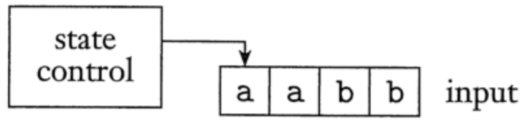
$\delta: Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$

4. q_1 is the start state, and
5. $F = \{q_4\}$.

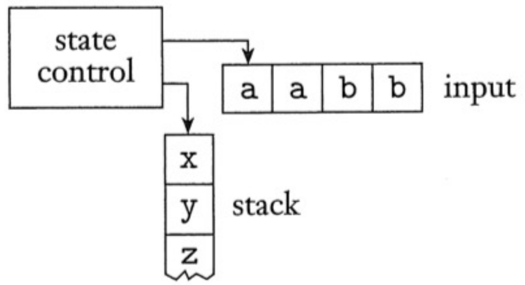
Aside: Determinism & Non-determinism

Push-down Automata (PDA)

- Finite automaton



- Push-down automaton



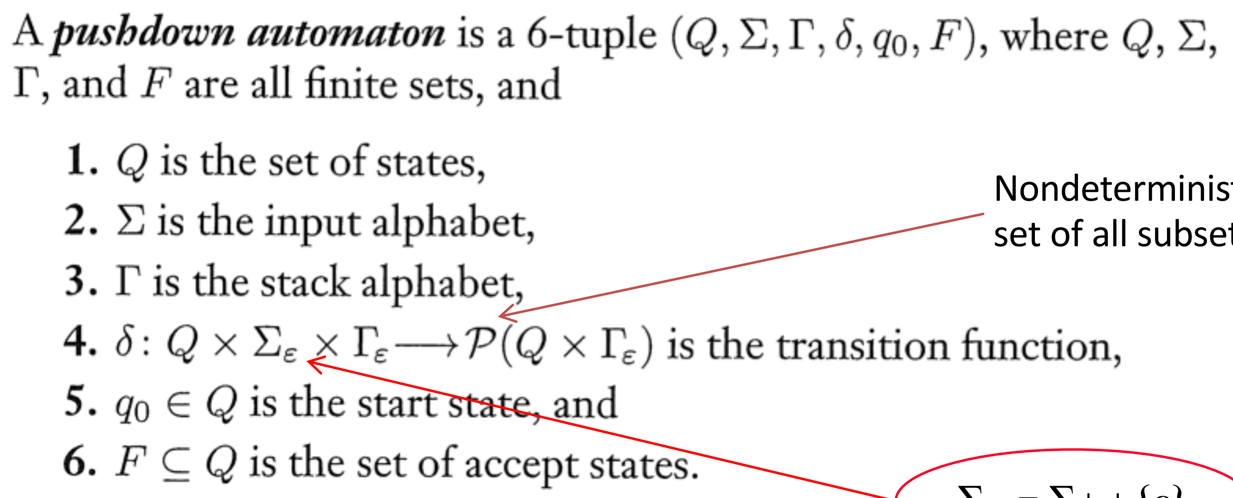
Aside: Determinism & Non-determinism

Push-down Automata (PDA)

- PDA can be deterministic or nondeterministic
- Nondeterministic PDA are more powerful than deterministic PDA
- (different situation to DFA and NFA)
- Nondeterministic PDA can recognize languages that deterministic PDAs cannot

Aside: Determinism & Non-determinism

Push-down Automata (PDA)



A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta \colon Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

Nondeterministic. Power set: set of all subsets of $Q \times \Gamma_\varepsilon$

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$

Transition function: current state, next input symbol read, top symbol on stack determine the next move of PDA (i.e. some combination of new state and stack operation)

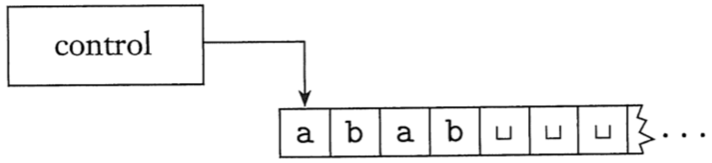
Aside: Determinism & Non-determinism

Turing machines

- Proposed by Alan Turing in 1936
  - Similar to finite automaton but has unlimited and unrestricted memory
  - Can do anything a general-purpose computer can do
  - But … cannot solve some problems (and, so, neither can computers)
    - Beyond the limits of theoretical computation

Aside: Determinism & Non-determinism

Turing machines



– **Infinite tape**: unlimited memory

- **Tape head** (**read** and **write symbols** to the tape, **move** left and right)

- Tape only contains input string initially; blank everywhere else

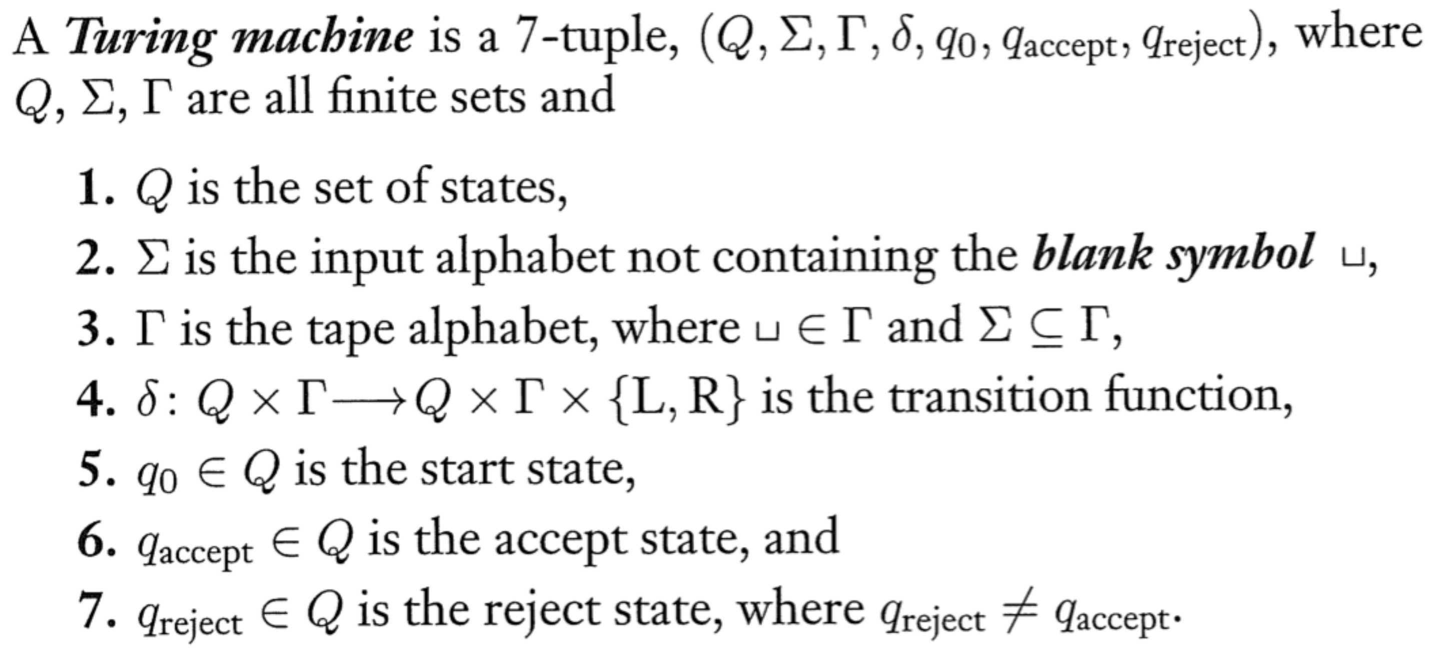
- Computes until it decides to produce an output

  Output *accept* if it enters Accepting state;
  Output *reject* if it enters Rejecting state

- If it doesn't enter accepting or rejecting state, it **loops** forever, never **halting**

Aside: Determinism & Non-determinism

Turing machines



A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,

2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,

3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,

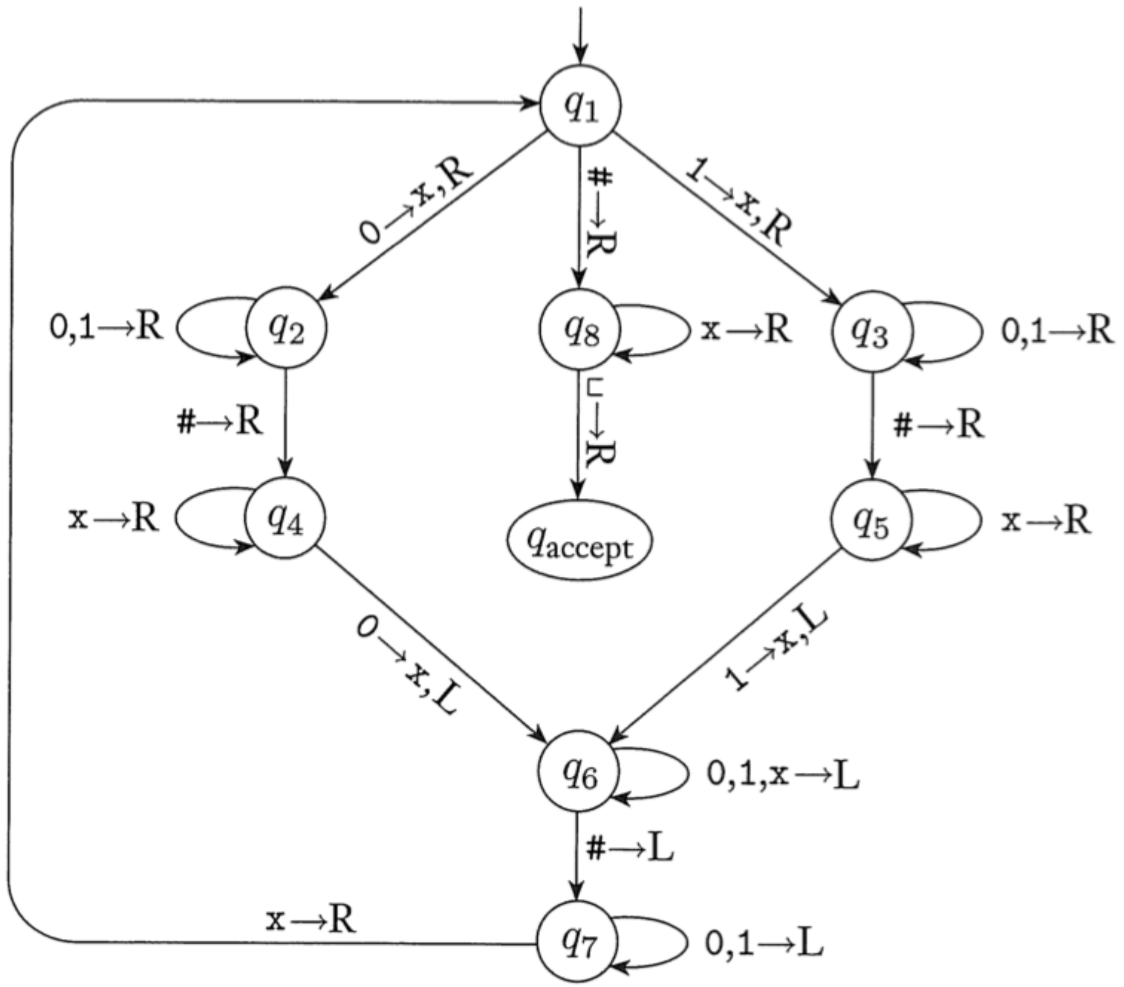
4. $\delta \colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,

5. $q_0 \in Q$ is the start state,

6. $q_{\text{accept}} \in Q$ is the accept state, and

7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

Aside: Determinism & Non-determinism



NP

- Definition of NP
  - Set of all decision problems solvable in polynomial time on a nondeterministic Turing machine
  - Important definition because it links many fundamental problems
- Useful alternative definition
  - Set of all decision problems with efficient verification algorithms

    efficient = polynomial number of steps on deterministic TM

NP

- NP = set of decision problems with efficient (polynomial time) verification algorithms
- Why doesn't this imply that all problems in NP can be solved efficiently?
  - BIG PROBLEM: need to know certificate ahead of time
    - real computers can simulate by guessing all possible certificates and verifying
    - naïve simulation takes exponential time unless you get "lucky"

NP-Completeness

- NP-hard:
  - A problem that is at least as hard as any problem in NP
  - That is, any problem in NP can be reduced to an NP-hard problem in polynomial time
- NP-complete problems are NP problems that are NP-hard
  - “Hardest computational problems” in NP

NP-Completeness

A problem B is NP-complete if it satisfies two conditions

- B is in NP
- Every problem A in NP is polynomial time reducible to B

NP-Completeness

- Each NPC problem's fate is tightly coupled to all the others (complete set of problems)

- Finding a polynomial time algorithm for one NPC problem would automatically yield a polynomial time algorithm for all NP problems

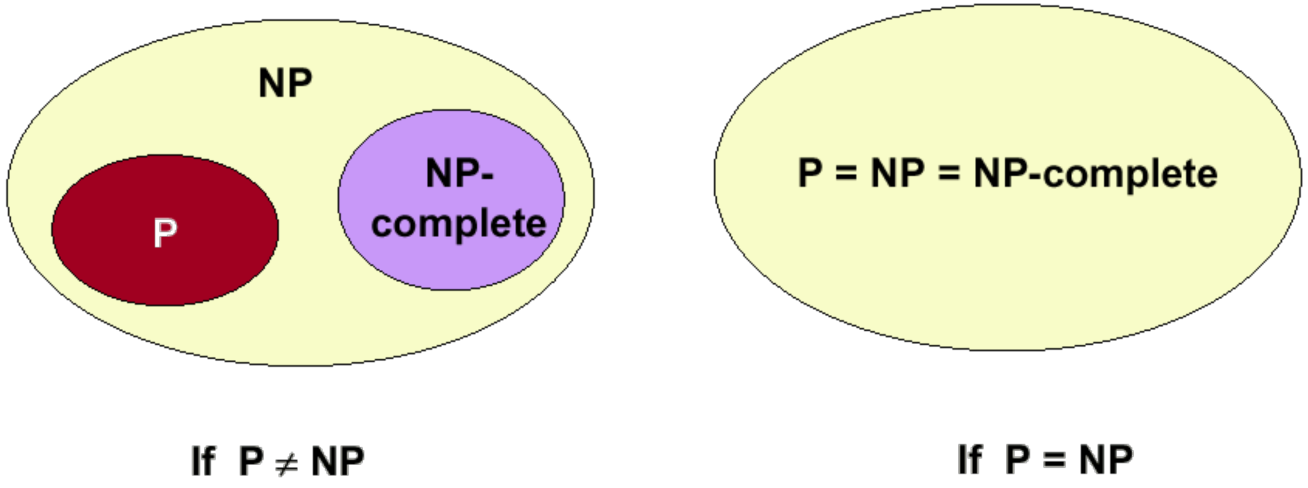
- Proving that one NP-complete problem has an exponential lower bound would automatically prove that all other NP-complete problems have exponential lower bounds

The Big Question

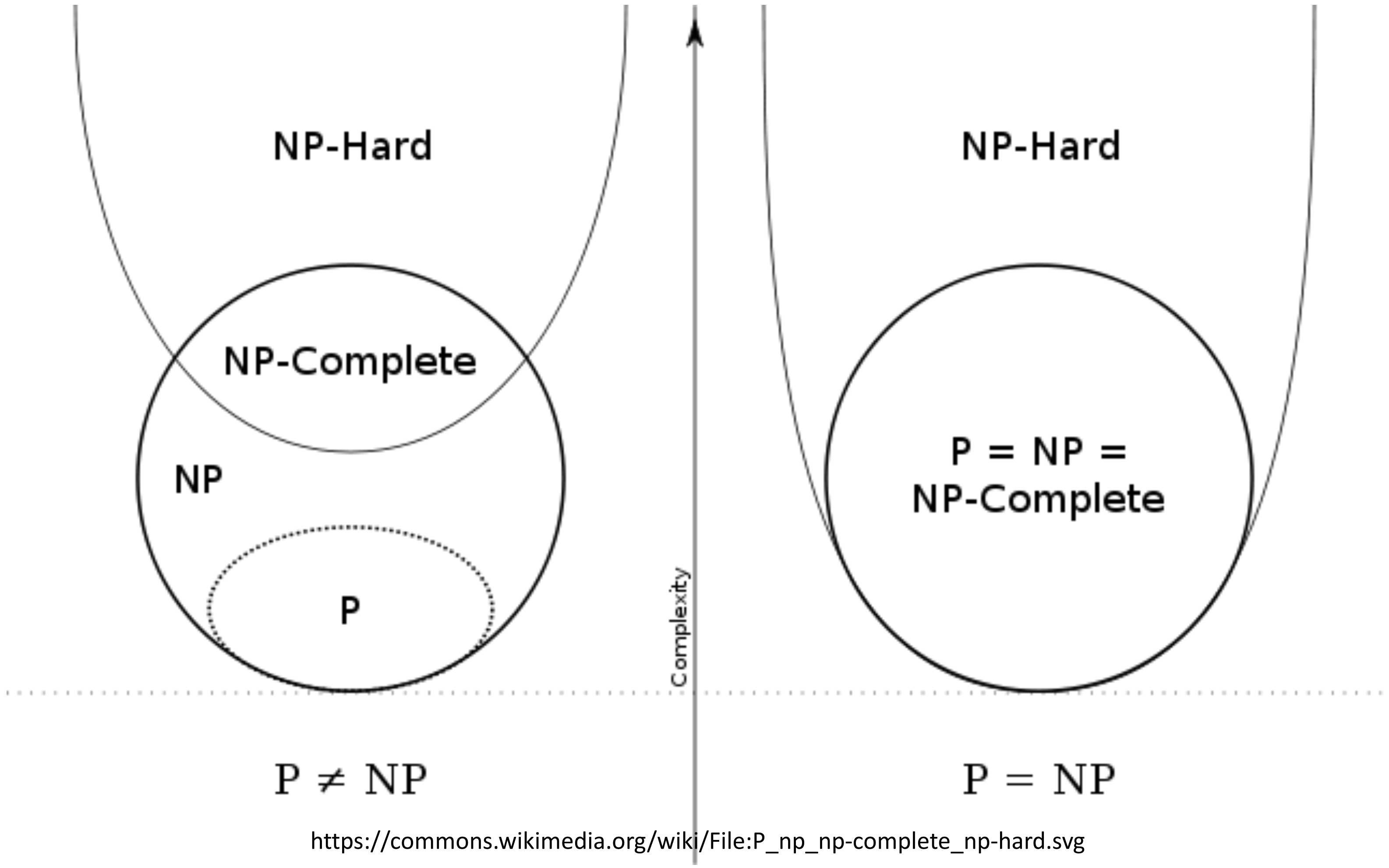
- Does P = NP?

  Is the original DECISION problem as easy as VERIFICATION?

- Most important open problem in theoretical computer science. Clay Institute of Mathematics offers $1m prize



The Big Question



https://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg

The Big Question

- If P=NP, then
  - There are efficient algorithms for TSP and factoring
  - Cryptography is impossible on conventional machines
  - Modern banking system will collapse
- If not, then
  - Can’t hope to write efficient algorithm for TSP
  - But maybe efficient algorithm still exists for testing the primality of a number – i.e., there are some problems that are NP, but not NP-complete

The Answer?

- Probably no, since
  - Thousands of researchers have spent four decades in search of polynomial algorithms for many fundamental NP-complete problems without success
  - Consensus opinion: P ≠ NP
- But maybe yes, since
  - No success in proving P ≠ NP either

Dealing with NP-Completeness

- Hope that a worst case doesn't occur
  - Complexity theory deals with worst case behavior. The instance(s) you want to solve may be "easy"
    - TSP where all points are on a line or circle
    - 13,509 US city TSP problem solved (Cook et. al., 1998)
- Change the problem
  - Develop a heuristic, and hope it produces a good solution.
  - Design an approximation algorithm: algorithm that is guaranteed to find a high- quality solution in polynomial time
    - active area of research, but not always possible
- Keep trying to prove P = NP

Conclusion

- It is not known whether NP problems are tractable or intractable
- But, there exist provably intractable problems
  - Even worse – there exist problems with running times unimaginably worse than exponential
- More bad news: there are provably noncomputable (undecidable) problems
  - There are no (and there will never be) algorithms to solve these problems

Summary

- NP - class of problems which admit non-deterministic polynomial-time algorithms
- P - class of problems which admit (deterministic) polynomial-time algorithms
- NP-Complete - the hardest of the NP problems (every NP problem can be transformed to an NP-Complete problem in polynomial time)
- So, is NP = P or not?

Summary

- We don’t know!
- The NP=P? problem has been open since it was posed in 1971 and is one of the most difficult unresolved problems in computer science

Summary

- A polynomial function is one that is bounded from above by some function n^k for some fixed value of k
  (i.e., $k \neq f(n)$)

- An exponential function is one that is bounded from above by some function k^n for some fixed value of k
  (i.e., $k \neq f(n)$)

- Strictly speaking, n^n is not exponential but super-exponential

Summary

- Polynomial-time algorithm
  - Order-of-magnitude time performance bounded from above by a polynomial function of n
  - Reasonable algorithm
- Super-polynomial / exponential and super-exponential time algorithms
  - Order-of-magnitude time performance bounded from above by a super-polynomial, exponential, or super-exponential function of n
  - Unreasonable algorithm

Summary

- There are many (approx. 1000) important and diverse problems which exhibit the same properties as the monkey puzzle problem (e.g., TSP)
- All admit unreasonable, exponential-time, solutions
- None are known to admit reasonable ones
- But no-one has been able to prove that any of them REQUIRE super-polynomial time

Summary

- Examples of NP-Complete Problems
  - 2-D arrangments (cf. pattern matching / recognition)
  - Path-finding (e.g. travelling salesman TSP; Hamiltonian)
  - Scheduling and matching (e.g. time-tabling)
  - Determining logical truth in the propositional calculus
  - Colouring maps and graphs

Summary

- All NP-Complete problems seem to require
  - construction of partial solutions
  - and then backtracking when we find they are wrong

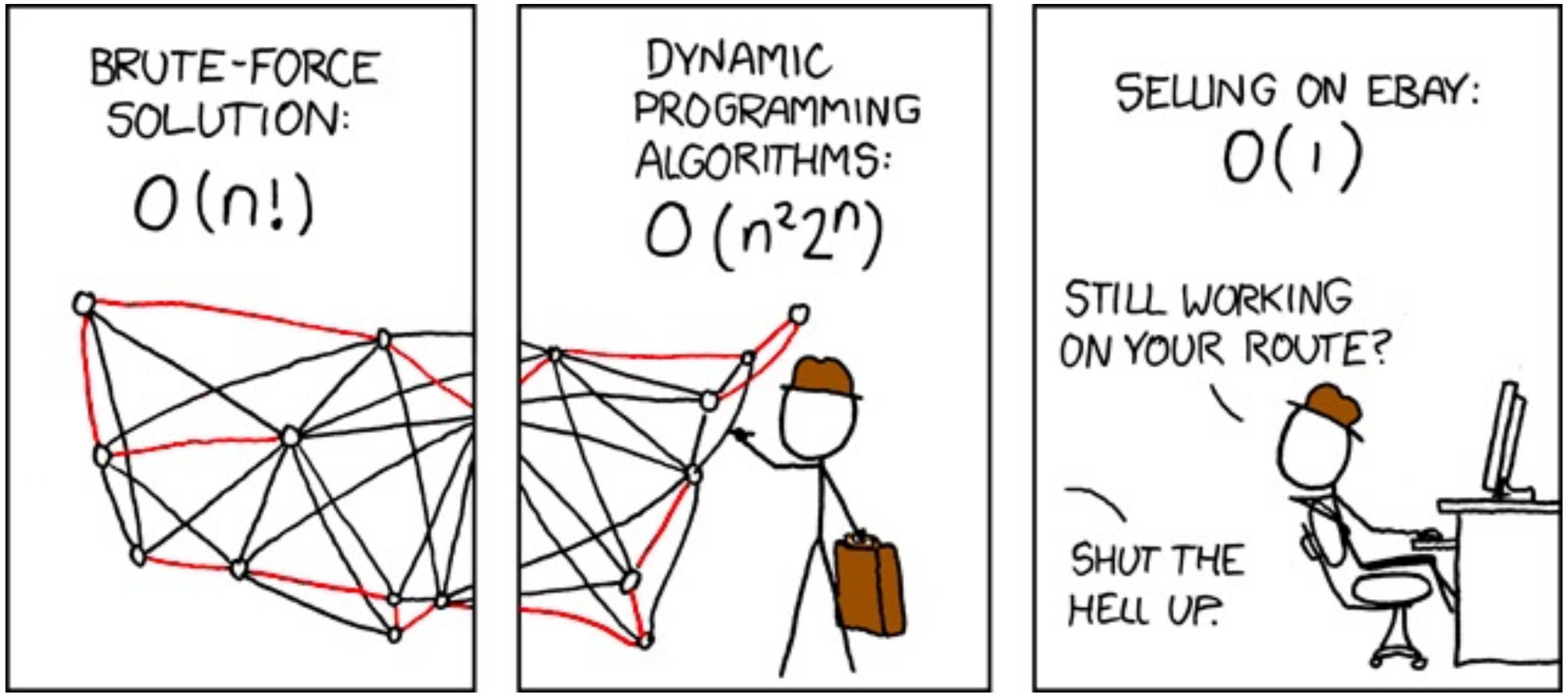
  in the development of the final solution

- However
  - if we could ‘guess’ at each point in the construction which partial solutions were to lead to the ‘right’ answer
  - then we could avoid the construction of these partial solutions and construct only the correct solution

Summary

- Important property of NP-Compete problems
  - Either all NP-Complete problems are tractable or none of them are
  - If there exists a polynomial-time algorithm for any single NP-Complete problem, then there would be necessarily a polynomial-time algorithm for all NP-Complete problems
  - If there is an exponential lower bound for any NP-Complete problem, they all are intractable

Summary


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.