Polynomial Algorithms

- Problems that can be solved in polynomial (of input size) time.
- Problems that cannot be solved in polynomial time are known as **intractable**.
- Those requiring O(n^{100}) cannot be called tractable, though theoretically **tractable**.
- But there are few such algorithms.
- Algorithms requiring polynomial time on a reasonable model of computaional would also require polynomial time on other models.

Polynomial Algorithms

- Since polynomials are closed under addition, multiplication one output of polynomial algorithm can be fed as input to another polynomial algorithm and composite algorithm is polynomial.
- That is if there exists a polynomial reduction of one problem to another and the latter can be solved in polynomial time then the former has a polynomial algorithm.

Intractable

- Many instances where algorithms employ guided exhaustive searches for solution are not polynomial. These are **intractable**.
- Causes of intractability could be
  - Guided exhaustive search: TSP, n-queen problem, or tiles puzzle, etc.
  - Output is too big: For example, we may need to generate all permutations of n elements or all spanning trees of a graph.
  - Undecidable problems: problems which cannot be solved. For example, given an input and the program, some one asks question like whether the program will terminate.

Abstract Problem

- An abstract problem Q is a binary relation on the set I of problem instances and the set S of solutions.
- For example, consider the problem of shortest path in a graph G.
- Given two vertices, say s, d in a graph G,
  - it is a triple (G, s, d), and
  - a solution to it is sequence of vertices of G that is the shortest path, the sequence is empty if there is no path between s and d.
- Thus `Shortest Path` associates each instance of problem to an member of the solution set which consists of sequences of vertices from G.
- Since the shortest path is not unique, the association is a relation.

Decision Problem

- We require a more concretized version of the above definition called **decision problem**.
- A decision problem – can be answered **YES/NO**.
- In this version $Q : I \longrightarrow \{0, 1\}$.
- The shortest path problem can be restated as a decision problem as follows.

$$I = \{(G, s, d, k) | \ G = (V, E) \text{ is an unweighted graph } s \in V \text{ is the source } d \in V \text{ is the destination, } k \text{ is the upper bound on the shortest path}\}$$

- Let $S = \{0, 1\}$.
- Let $a = (G, s, d, k) \in I$

$$\text{Path}(a) = \begin{cases} 1 & \text{if the shortest path length } \leq k \\ 0 & \text{if the shortest path length } > k \end{cases}$$

Optimization Problem

- Many abstract problems are not decision problems but **optimization** problems.
- In such problems some objective value is either maximized or minimized.
- In order to study these problems for $\mathcal{NP}$ completeness they must be recast as decision problems.
- Typically, a bound will on objective value is fixed, and the optimization is recast as whether the value exceeds the specified bound.

Optimization Problem

- We have just seen how shortest path (which minimizes the distance) is translated into a decision problem by requiring that the length of the path to have value less than k.
- The problem recasting does not have much impact on the theory of $\mathcal{NP}$ completeness.
- Because if we could solve the optimization problem then we can quickly solve the corresponding decision problem.
- We have already seen that the optimization problems are also hard.

Encoding

- Encoding of a set I of abstract objects is a mapping $\mathbf{e}$ from I to a set of alphabets Σ (which have at least 2 symbols).
- For example, the standard binary encoding for the set of natural number described as follows:
  - Instance set: $I = N = \{0, 1, 2, \ldots\}$,
  - Alpahbets: $\Sigma^* = \{0, 1\}^*$.
  - Encoding:$\mathbf{e} : \mathbf{N} \longrightarrow \Sigma^*$, given by

    $\mathbf{e}(0) = 0, \mathbf{e}(1) = 1, \mathbf{e}(2) = 10, \mathbf{e}(3) = 11,$
    $\mathbf{e}(4) = 100, \ldots,$ etc.

Encoding

- Compound objects can be encoded as bit strings by combining the encoding of the constituent parts.
- An algorithm takes an encoding of the problem instance as input.
- A **concrete problem** is the one whose instance set I is the set of strings over an alphabet set Σ.
- That is an encoding maps an abstract problem into a concrete problem.
- In other words, $\mathbf{e}(Q)$ is the concrete decision problem for the corresponding abstract decision problem Q.

Encoding

- Clearly, there are strings in Σ^* that may not represent any meaningful abstract problem instance.
- All such strings will be assumed to be mapped to 0.
- Thus, if the solution to an abstract instance $i \in I$ is $Q(i) \in \{0, 1\}$, then the solution to the concrete instance $\mathbf{e}(i) \in \{0, 1\}^*$ is also $Q(i)$.

Encoding

- An algorithm solves a concrete problem in O($T(n)$), if it solves any instance i of length $|i| = n$ in time of at most O($T(n)$).
- This means encoding is a bridge between the concrete and the abstract problem instances.
- An abstract problem is solvable in p-time if corresponding concrete problem is solvable in p-time.
- So, we can not talk of solving an abstract problem without first specifying the encoding.
- Encoding may play a big role in running time.

Examples of Encoding

Encoding	Example	Length of k using the encoding
Unary	$\mathbf{e}(7) = 1111111$	$\lvert k\rvert_{e_1} = k$
Binary	$\mathbf{e}(7) = 111$	$\lvert k\rvert_{e_2} = \lfloor \log_2 k \rfloor + 1$
Ternary	$\mathbf{e}(7) = 21$	$\lvert k\rvert_{e_3} = \lfloor \log_3 k \rfloor + 1$
Base b	–	$\lvert k\rvert_{e_b} = \lfloor \log_b k \rfloor + 1$

Examples of Encoding

- Except for the unary encoding for every other encoding, the natural numbers have lengths related polynomially, since
$$\log_{b_1} k = \frac{\log_{b_2} k}{\log_{b_2} b_1} = c \log_{b_2} k.$$
- The unary encoding has the length exponential on the size of any other encoding, since $|k|_{e_1} = k = b^{\log_b k} \approx b^{|k|_{e_b}}$, for any $b \geq 2$.

Encoding

- For example, suppose the unary encoding used and
  - Suppose an integer k is to be provided as input to the algorithm that has a running time is $\Theta(k)$.
  - Then for a length-n input, the running time of algorithm is $\mathsf{O}(n)$.
- If the binary encoding is used then string for input k has length $n = \lceil \log k \rceil$.
- In this case running time of algorithm is $\Theta(2^n)$, thus exponential in the size of input.
- This means depending on encoding algorithms can either run in p-time or super p-time.

Encoding

- We will assume a *reasonable* encoding scheme, not an expensive encoding scheme like unary.
- The meaning of reasonable cannot be formalized but at least the encoding should capture the following notions
  - Encoding of a problem instance must be concise and should not be padded with extra or unnecessary symbols.
  - The numbers occurring in the problem instance should be represented in binary, or octal, or in any fixed base other than unary.

Encoding

- For example, a graph $G = (V, E)$ where $V = \{V_1, V_2, V_3, V_3\}$ and $E = \{(V_1, V_2), (V_2, V_3)\}$ undere different encoding schemes:

Scheme	String	Length
Vertex & edge list	$V[1]V[2]V[3]$ $V[4](V[1]V[2])$ $(V[2]V[3])$	36
Adjancy list	$(V[2])(V[1]V[3])$ $(V[2])()$	24
Adjancey matrix	0100/1010/ 0100/0000	19

Encoding

- Bounds on input lengths using three encodings will be as follows:

Scheme	Lower bound	Upper bound
Vertex & edge list	$4\|V\| + 10\|E\|$	$4\|V\| + 10\|E\| + (\|V\| + 2\|E\|) \cdot \lceil \log_{10} \|V\| \rceil$
Adjancy list	$2\|V\| + 8\|E\|$	$2\|V\| + 8\|E\| + 2\|E\| . \lceil \log_{10} \|V\| \rceil$
Adjancey matrix	$\|V\|^2 + \|V\| - 1$	$\|V\|^2 + \|V\| - 1$

Formal Language Framework

- It allows us to define algorithms for concrete decision problems as **machines** that operate on languages.
- Consider, the problem of primality test.
- Use binary encoding $\Sigma = \{0, 1\}$, so the concrete problem is
  - PRIME: $\{0, 1\}^* \longrightarrow \{0, 1\}$ with
  - PRIME(10) = PRIME(11)= PRIME(101) = PRIME(111) = PRIME(1011) = $\ldots$ = 1,
  - PRIME(0) = PRIME(1)= PRIME(100) = PRIME(110) = PRIME(1010) = $\ldots$ = 0,

Formal Language Framework

- We associate with PRIME a language L_{prime} corresponding to all string s over {0, 1} with PRIME(s)=1:

$$L_{prime} = \{10, 11, 101, 111, 1011, \ldots\}$$

- The algorithm for primality test can then be interpreted as a machine that accepts only valid strings (that are primes) of the formal language L_{prime} and rejects all other strings.
- Empty string is denoted by ϵ, and the empty language by ϕ.
- The language of all strings over $\Sigma = \{0, 1\}$ is

$$\Sigma^* = \{\epsilon, 0, 1, 00, 01, 11, 000, \ldots\}$$

- Every language L over Σ is a subset of Σ^*.

Formal Language Framework

- Operations on languages:
  - `Union`: $L_1 \cup L_2$, `intersection`: $L_1 \cap L_2$, `complement`: $\Sigma^* - L$.
  - `Concatenation`: $\{x_1 x_2 | x_1 \in L_1$ and $x_2 \in L_2\}$
  - `Closure`: $L^* = \{\epsilon\} \cup L \cup L^2 \cup \ldots$, where L^k is the language obtained by k concatenation of L.
- From language theory framework, the set of instances for any decision problem Q is simply the set Σ^* where $\Sigma = \{0, 1\}$.
- So Q is a language

$$L = \{x \in \Sigma^* | Q(x) = 1\}$$

Formal Language Framework

- For example, the language for the decision problem shortest path is

$$\text{PATH} = \{(G, s, d, k) \mid G = (V, E) \text{ is an undirected graph } s, d \in V, k \geq 0 \text{ is an integer, and there exists a path from } s \text{ to } d \text{ in } G \text{ whose length is at most } k\}$$

Formal Language Framework

- An algorithm $\mathbf{A}$ **accepts** a string $x \in \{0,1\}^*$, if given input x the algorithm outputs $\mathbf{A}(x) = 1$.
- The language **accepted** by $\mathbf{A}$ is the set

$$L = \{x \in \{0,1\}^* | \mathbf{A}(x) = 1\}$$

- $\mathbf{A}$ **rejects** a string x if $\mathbf{A}(x) = 0$.
- Note that if $\mathbf{A}$ **accepts** a language L, then it does not mean that $\mathbf{A}$ **rejects** $x \notin L$
- For example, $\mathbf{A}$ may loop for ever with $x \notin L$.

Formal Langauge Framework

- L is **decided** by **A** if every string is either **accepted** or **rejected** by it.
- L is **decided** in **p-time** if for any length-n string $x \in \Sigma^*$, **A decides** x in $O(n^k)$ time.
- So **accept** and **decide** are not exactly the same.
- For **deciding** L, **A** needs to **accept** all strings in L and also **reject** the strings not in L.
- But for **accepting** L, **A** needs to only **accept** all strings in L.

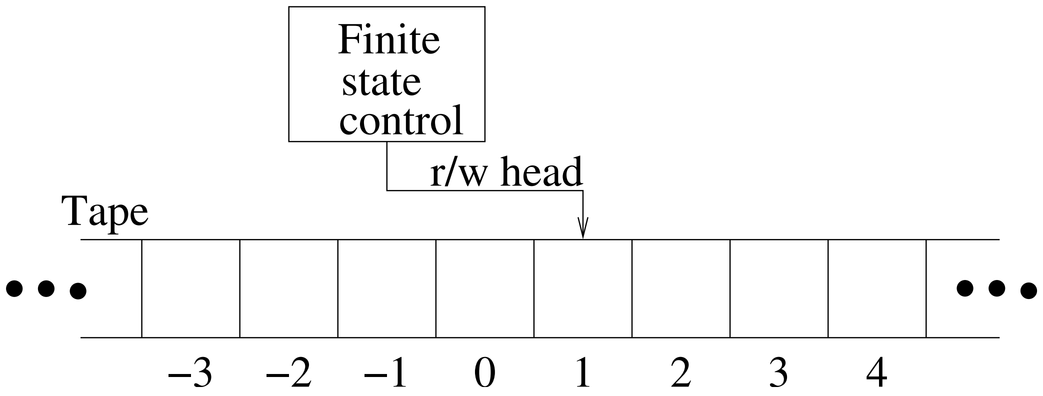
Formal Language Framwork

- For example, if $\mathbf{A}$ computes shortest path from s to d using BFS and then compares the distance with k. It outputs 1 if the distance is $\leq k$ and halts.
- But $\mathbf{A}$ loops for ever if distance $> k$.
- Then $\mathbf{A}$ does not decide PATH, however $\mathbf{A}$ accepts PATH.
- Similarly, for halting problem there is an accepting algorithm but no decision algorithm.
- **Complexity class** is a set of languages, membership of which is determined by a **complexity measure**.
- Thus complexity class $\mathcal{P}$ is

$$\mathcal{P} = \{L \subseteq \{0,1\}^* |\ \exists \text{ an algo } \mathbf{A} \text{ which decides } L \text{ in polynomial time}\}$$

Deterministic Model

- The deterministic computational model that would allow algorithm to behave as a machine (1-tape Turing machine).
- The one-tape Turing machine is pictured as follows.



Deterministic Model

- A program for a DTM specifies following information.
  - A finite set Γ of tape symbols, $\Gamma \subset \Sigma$.
  - A distinguished blank symbol $\in \Gamma - \Sigma$
  - A finite set Q of states, including the distinguished start state q_0 and two distingusihed halt states q_N and q_Y.
  - A transition function

    $$\delta : (Q - \{q_N, q_Y\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{+1, -1\}$$
- The input to DTM is a string $x \in \Sigma^*$.
- The string is placed in squares 1 through $|x|$ with one symbol per square on the tape.
- Other squares contain the blank symbol.

Deterministic Model

- The program starts its operation in q_0 state with r/w head at square 1.
- Computation proceeds in a step-by-step manner.
- If the current state is either q_Y or q_N the computation ends.
  - The answer is $\mathbf{YES}$ if the state of termination is q_Y.
  - The answer is $\mathbf{NO}$ if the state of termination is q_N.
- If the current state is $q \in Q - \{q_Y, q_N\}$ and some symbol $s \in \Gamma$ being scanned, and the value $\delta(q, s) = (q', s', \Delta)$ is defined.
  - The r/w head erases s and writes s' in its place and moves 1 step either to left (if $\Delta = -1$) or to right (if $\Delta = +1$).

Deterministic Model

- Consider a DTM program with

$$\Gamma = \{0, 1, b\},\ \Sigma = \{0, 1\} \text{ and}$$

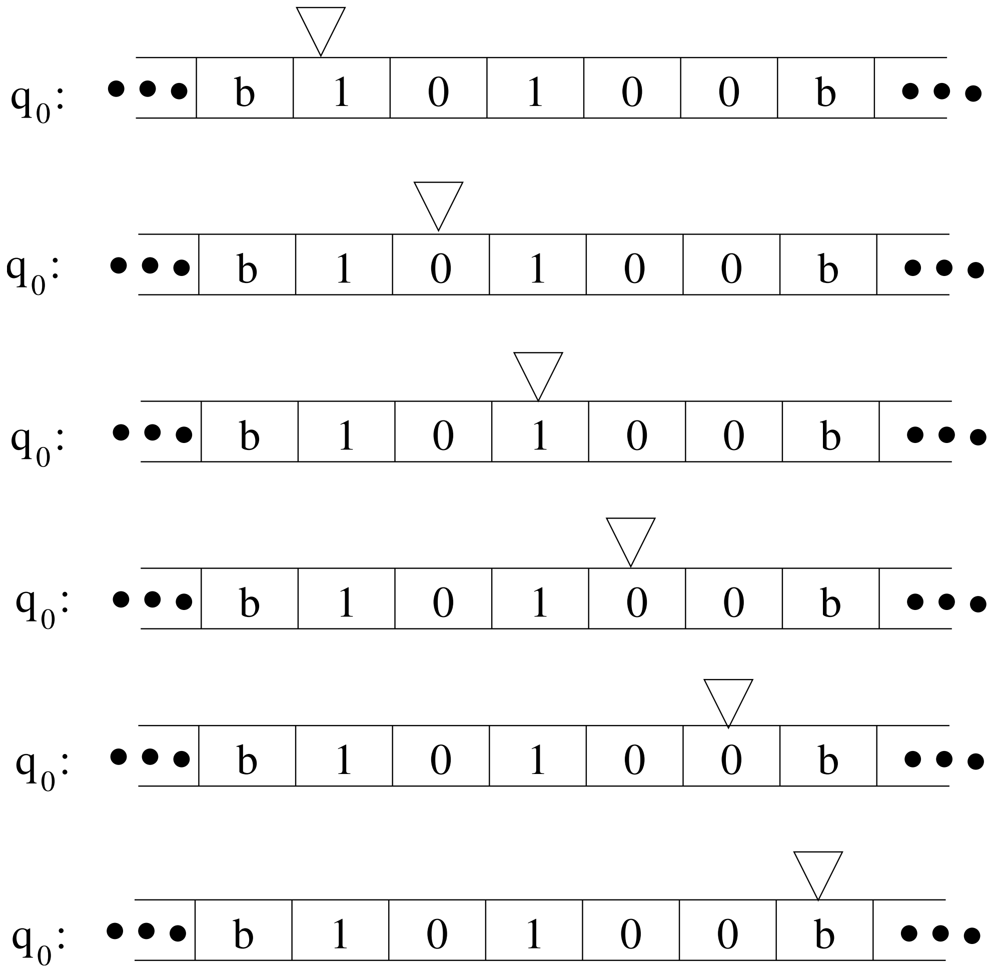
$$Q = \{q_0, q_1, q_2, q_3, q_N, q_Y\}$$

- The function $\delta(q, s)$ is provided below for an input symbol from Σ.

q	0	1	b
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(q_1, b, -1)$
q_1	$(q_2, b, -1)$	$(q_3, b, -1)$	$(q_N, b, -1)$
q_2	$(q_Y, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$
q_3	$(q_N, b, -1)$	$(q_N, b, -1)$	$(q_N, b, -1)$

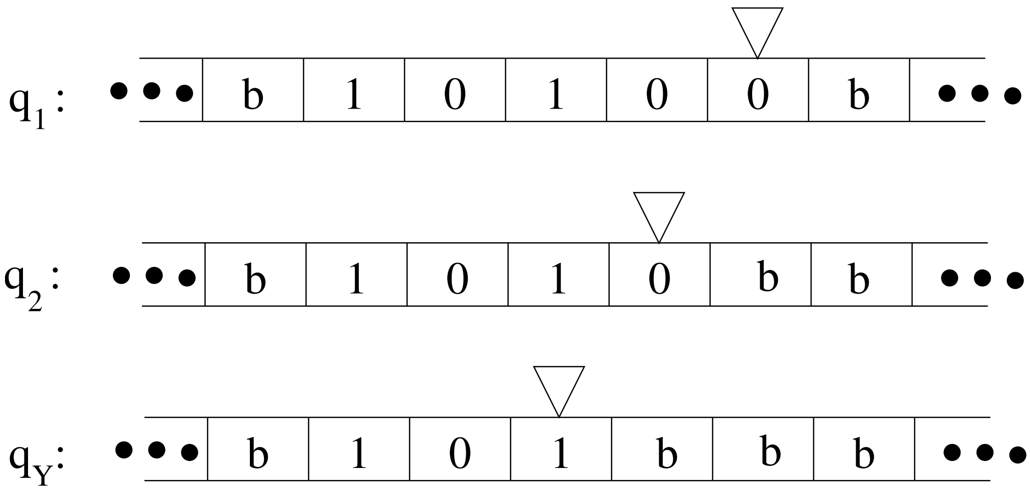
Deterministic Model

- Step-by-step execution of program for input string $x = 10100$



Deterministic Model

- The program terminates in q_Y state for the given string.



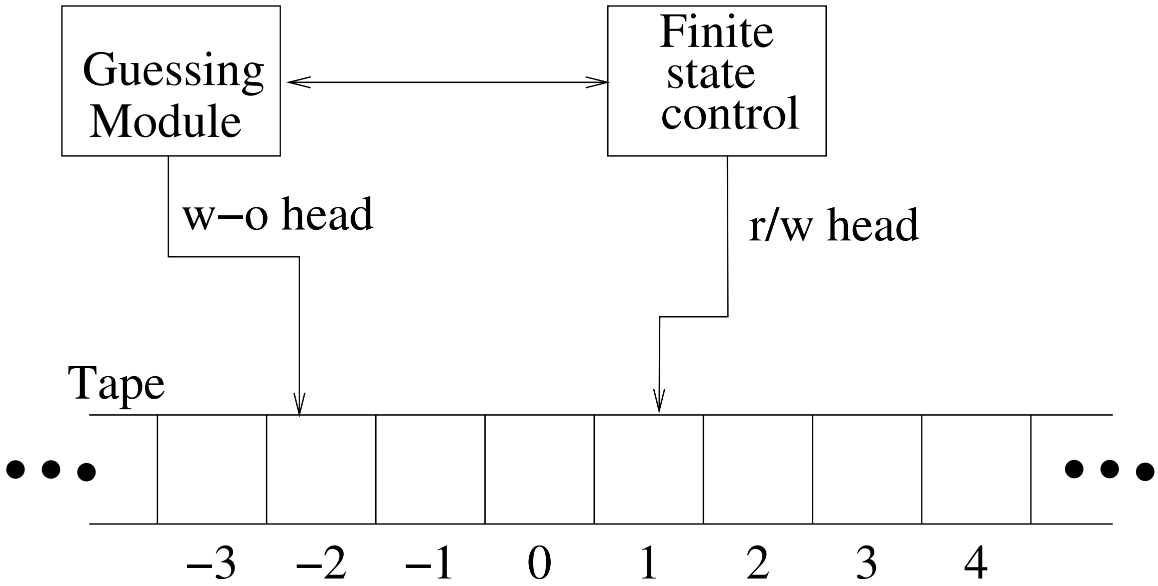
Deterministic Model

- The program is said to recognize a language if it accepts all strings from of Σ^* for a given set Σ.
- For recognition problem it is not necessary that the program should terminate for every string, it may just terminate in acceptance state for all valid strings of the input language.
- But for the notion of a DTM algorithm we need additional constraint that the DTM program should halt on all strings.
- The correspondence between language recognition and the solution of a decision problem is:

  A DTM program M solves a decision problem Π under an encoding scheme e if M halts for all input string over its input alphabet and $L_M = L\{\Pi, e\}$.

Nondeterministic Model

- NDTM also can be defined in the same way.
- It has an additional *write-only* head for **guessing module**
- The picture is as follow:



Nondeterministic Model

- NDTM program includes the tape alphabet Γ, input alphabet Σ, blank symbol, state set Q, initial state q_0, final states q_Y, q_N, and the transition function

  $$\delta : (Q - \{q_N, q_Y\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, +1\}$$

- The computation of an NDTM differs from that of DTM in that there are two stages: **Guessing** and **Checking**.
- In guessing state guessing module will direct w-o head to either to write one symbol in the current square of the tape or to move one square to left.
- Initially the input string x is placed on the tape from square 1 through $|x|$, and the w-o head is at -1.
- After guessing module becomes inactive the finite state control become active in state q_0.

NDTM

- The guessing module work totally arbitrary
  - while selecting symbols of Γ that should be written on the tape.
  - whether to remain active or not.
- It means it can write any string from Γ^* before it halts and it may as well never halt.
- The checking stage beigns when the finite state control is activated in state q_0.
- From this state onward the NDTM program takes control and works exactly in the similar way as a DTM program.
- The guessing module and w-o head cease to play any role.

Nondeterministic Model

- Tractability of a problem is unaffected as long as the computational model is restricted to a model of a realistic computer.
- The objective of choosing a model is purely for convenience, but it should be done without sacrificing the applicability of the results.
- For studying sequential algorithms, RAM model was used instead of the ideal abstract model of a Turing machine
- The choice was justified due to following result regarding simulation.

Simulated machine	Simulating machine	
	1-TM	RAM
1-TM	–	$O(T(n) \log T(n))$
RAM	$O(T(n))$	–

Nondeterministic Model

- We will use an augemnted RAM like model as nondeterministic computation model to present the concept of nondeterministic algorithms.
- Three new instructions:
  1. `choice(S)`: arbitrarily chooses one element from a set `S`.
  2. `failure()`: signals unsuccessful completion of the current thread.
  3. `success()`: Signals successful completion of the current thread and forces termination of other running threads.
- For example, `x = choice({1, ..., n})` results in `x` being assigned one of the values in the range `[1, n]`. But no rule specifying how the choice to be made.
- All three instructions take $O(1)$ time.

Nondeterministic Sorting Algo

```
nSort(A, n) {
   for (i=1; i≤ n; i++) B[i]=0;
   for (i=1; i≤ n; i++) {
      j = choice({1, ..., n});
      if (B[j] ≠ 0) failure();
      B[j] = A[i];
   }
   for (i=1; i≤ n; i++)
      if (B[i] > B[i+1]) failure();
   write(B[1..n]);
   success();
}
```

Satisfiability

- Let $x_1, x_2, \ldots, x_n$ be boolean variables. Let $\overline{x_i}$ denote negation of x_i.
- A litral is either a variable or its negation.
- A formula in the propositional calculus is an expression that can be constructed by using literals and logical operators $\wedge$ and $\vee$.
- For example

  $(x_1 \wedge x_2) \vee (x_3 \wedge \overline{x_4})$, and $(x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2})$

  are formulas.

Satisfiability

- A formula is in conjuctive normal form (CNF) iff it is represented as

$$\wedge_{i=1}^{k} C_i, \text{ where the } C_i \text{ are clauses}$$

- A clause is represented as $\vee l_{ij}$, l_{ij} being a literal.
- Similarly we can define disjunctive normal form as

$$\vee_{i=1}^{k} C_i, \text{ where } C_i = \wedge l_{ij}.$$

- **Satisfiability** (SAT) is to determine if a formula evaluates to true for some assignment of the truth values to the variables.

Satisfiability

- Stephen Cook proved that if SAT in $\mathcal{P}$ then all problems in $\mathcal{NP}$ are in P, i.e., $\mathcal{P} = \mathcal{NP}$.
- Equvalently, SAT is the hardest problem in $\mathcal{NP}$ class.
- A problem $\mathbf{A}$ is called $\mathcal{NP}$-hard if and only there exists a polynomial reduction of $\mathbf{A}$ to SAT problem.
- SAT is in $\mathcal{NP}$ – a nondeterministic polynomial algorithm for SAT is given below.

```
evalSAT(E, n) {
   for (i=1; i≤ n; i++)
      x[i] = choice({true, false});
   if (E(x[1,...,n])) success();
   else failure();
}
```

Relationship Between P and NP

- The relationship between $\mathcal{P}$ and $\mathcal{NP}$ is fundamental to study of theory of $\mathcal{NP}$ completeness.
- Clearly $\mathcal{P} \subseteq \mathcal{NP}$.
- To understand it just observe that suppose a problem $\Pi \in \mathcal{P}$ then the deterministic algorithm $\mathbf{A}$ for Π can be merely used for the checking stage of the nondeterministic algorithm for Π and ignore the guessing stage.
- On the other hand, it is easy to understand that there is no such simple way to covert a polynomial time nondeterministic algorithm to a polynomial time deterministic algorithm.
- So the inclusion is proper, or it looks like $\mathcal{P}$ is not equal $\mathcal{NP}$.

Relationship Between P and NP

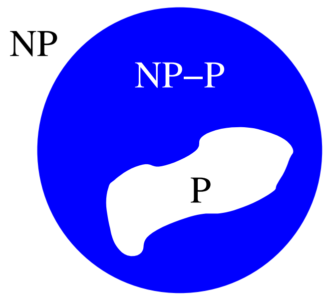
- The best result in the direction of the reverse inclusion is
  If $\Pi \in \mathcal{NP}$ then there exists a polynomial p such that Π can be solved by deterministic algorithm with time complexity O$(2^{p(n)})$.
- Let A be a p-time $\mathcal{NP}$ algorithm for Π. Let $q(n)$ be the bound on time complexity of A.
- This means q can be evaluated in p-time by taking $q(n) = c_1 n^{c_2}$ by taking c_1 and c_2 suitably large integers.
- Also for every accepted input of length n there exists a guess string (over tape alphabet Γ) of length at most $q(n)$ such that checking stage of A responds $\mathbf{YES}$ for that input in no more than $q(n)$ steps.

Relationship Between P and NP

- This implies that number of possible guesses to be considered is at most $k^{q(n)}$ where $k = |\Gamma|$. The shorter guesses can be padded with blank symbols.
- We can deterministically discover whether A has an accepting state for a given input of length n by applying checking stage until it halts or makes $q(n)$ steps on each of the $k^{q(n)}$ possible guesses.
- The simulation responds **YES** if it encounters a guessed string that leads to an accepting computation within the time bound.
- Otherwise it responds **NO**.

Relationship Between P and NP

- This yields a deterministic algorithm for solving Π.
- Further more the complexity is $q(n).k^{q(n)}$, which is O($2^{p(n)}$).
- Using backtracking or branch and bound method we can eliminate a large number of obviously useless guesses from consideration.
- This speeds up arriving at an accepting state.
- But the simulation would still neeed exponential time.
- Thus we picture the relationship as follows:



NP Hard Problems

- A problem Π whether a member of $\mathcal{NP}$ or not called $\mathcal{NP}$ hard if Π can be transformed in polynomial time to a problem that is known to be $\mathcal{NP}$ complete.
- The problem is called $\mathcal{NP}$ hard for the reason that it can not be solved in p-time unless there is proof that $\mathcal{P} = \mathcal{NP}$.
- According to this definition an $\mathcal{NP}$ complete problem is also an $\mathcal{NP}$ hard problem.
- Halting problem for deterministic algorithm is undecidable. It is an extreme example of $\mathcal{NP}$ hard problems.
- Halting problem is described as follows:

NP Hard Problems

Given an arbitrary deterministic algorithem A and an input I halting problem is to determine whether A will ever terminate with the input I.

- Since the problem is undecidable there is no question of an algorithm of any complexity being able to solve this. Hence there is no $\mathcal{NP}$ algorithm for the halting problem.
- We now need to provide a polynomial reduction of this problem to SAT.

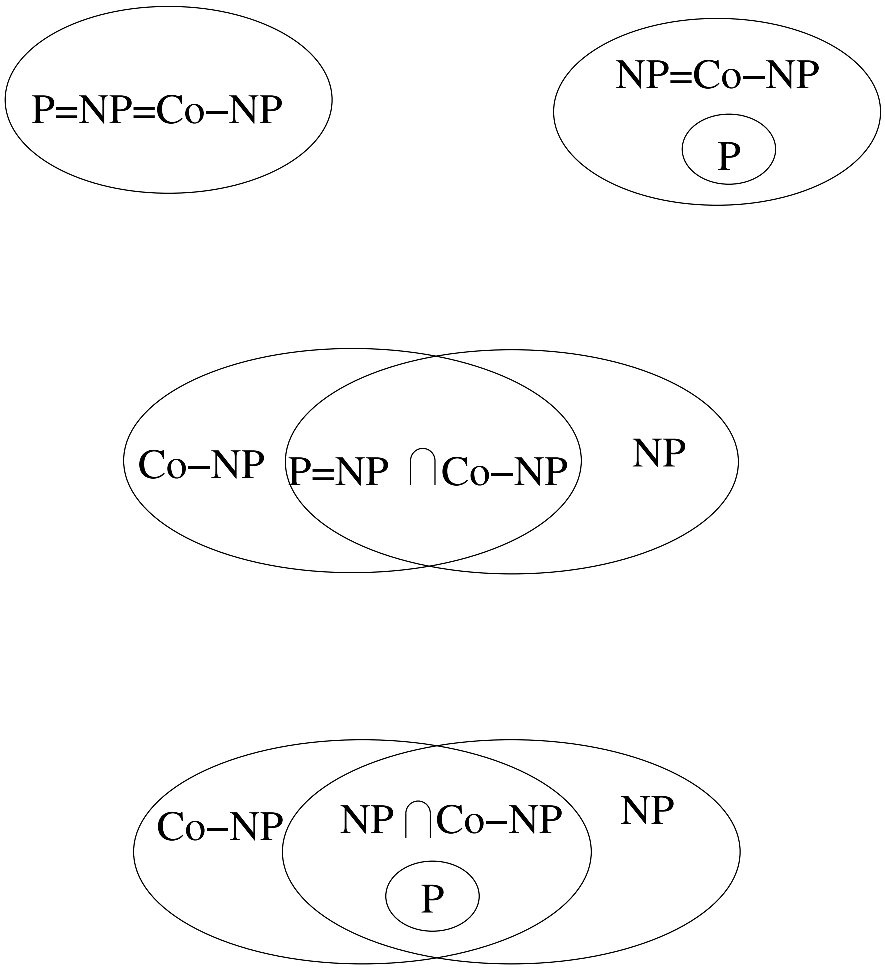
NP Hard Problems

- Construct an algorithm $\mathbf{A}$ whose input is a propositional formula $\mathbf{X}$.
- If $\mathbf{X}$ has n variables then $\mathbf{A}$ tries out all 2^n possible truth assignments and verifies if the $\mathbf{X}$ is satisfiable.
- If so then $\mathbf{A}$ terminates. Otherwise $\mathbf{A}$ goes into an infinite loop.
- This means if $\mathbf{A}$ halts on **Halting Problem** then there is a solution to SAT in polynomial time by using $\mathbf{A}$ and $\mathbf{X}$ as input to the algorithm for the halting problem.
- So, Halting problem is a $\mathcal{NP}$-hard problem that is not in $\mathcal{NP}$.

Cousins of NP Completeness

- $\mathcal{NP}$ hard problems which belong to class $\mathcal{NP}$ are called $\mathcal{NP}$ complete problem.
- Consider the complement $\overline{L}$ of languages $L \in \mathcal{NP}$.
- Does $\overline{L}$ belong to $\mathcal{NP}$?
- We define the set of languages L such $\overline{L}$ belongs to $\mathcal{NP}$, as Co-$\mathcal{NP}$.
- So the question whether the class $\mathcal{NP}$ is closed under **complementation** can be alternatively rephrased as: Is $\mathcal{NP} =$ Co-$\mathcal{NP}$?.
- We know that $\mathbf{P}$ is closed under complementation. (why?)
- Thus $\mathcal{P} \subseteq \mathcal{NP} \cap$ Co-$\mathcal{NP}$.
- But are there more languages in $\mathcal{NP} \cap$ Co-$\mathcal{NP}$.

Cousins of NP Completeness



NP Completeness

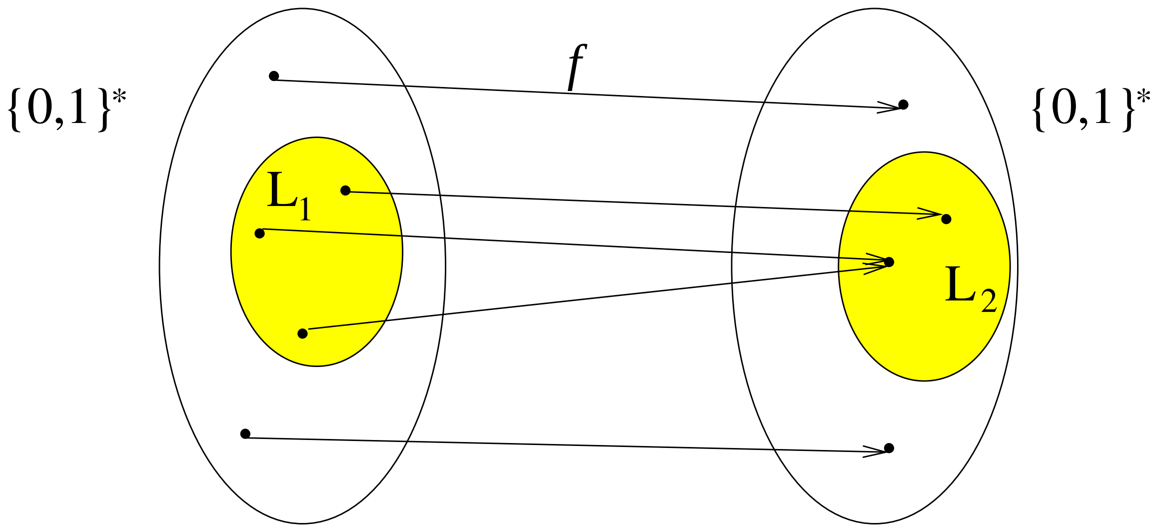
- Most compelling evidence as to why $\mathcal{P} \neq \mathcal{NP}$ comes from the existence of $\mathcal{NP}$ complete problems.
- The $\mathcal{NP}$ complete problem has the property that if any problem in $\mathcal{NP}$ complete class is solvable in p-time so will all problems in the class $\mathcal{NP}$.
- We use the notion of polynomial reducibility to show that all problems in the class $\mathcal{NP}$ complete can be reducible to each other polynomially.
- In terms of language framework $\mathcal{NP}$ complete languages are the hardest languages in $\mathcal{NP}$.

NP Completeness

- A problem Π can be reduce to another problem Π' if any instance of Π can be rephrased as an instance of Π'.
- For example, every instance of solving a linear equation can be reduced to an instance of solving a quadratic equation:
  - Given an instance $ax + b = 0$, we can transform it to an instance of $0.x^2 + ax + b = 0$.
  - So problem is no harder than solving a quadratic equation.

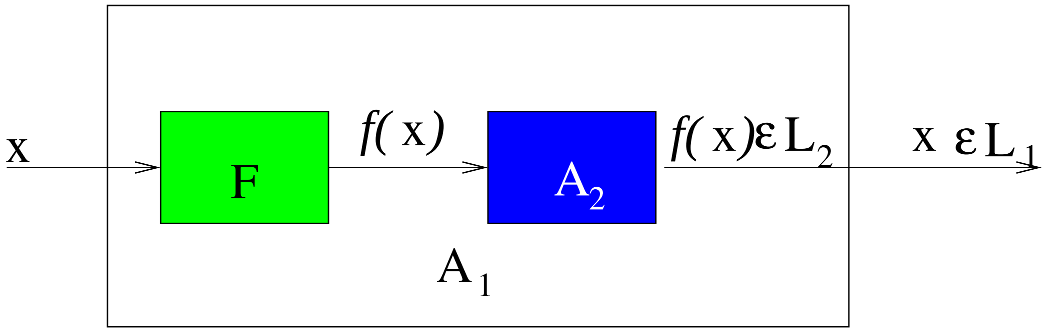
Reducibility

- In formal language framework, a language L_1 is polynomially reducible to a language L_2, denoted by $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.
- The function $f(x)$ is called reduction function, and a polynomial time algorithm that computes f is called a reduction algorithm.



Reducibility

- If $L_1, L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$ then $L_2 \in \mathcal{P}$ implies $L_1 \in \mathcal{P}$.
  - Let A_2 be a p-time algorithm for L_2 and let F be a p-time reduction function f.



  - Then we can transform every string in L_1 to $f(x)$ in p-time by applying F.
  - After that we apply A_2 to decide whether $f(x) \in L_2$.
  - Since both F and A_2 are p-time, the above construction is also p-time.

Reducibility

- A polynomial time reduction provide a formal means for showing that a problem is at least as hard as another to within a p-time factor.
- So $\mathcal{NP}$ complete problems can be defined as the hardest problem in $\mathcal{NP}$.
- Equivalently, a language $L \subseteq \{0,1\}^*$ is $\mathcal{NP}$ complete if
  1. $L \in \mathcal{NP}$, and
  2. $L' \leq_P L$ for every $L' \in \mathcal{NP}$.
- If a language satisfies property 2 but not necessarily 1 then we say the problem is $\mathcal{NP}$ hard.

