Or, how to make $1,000,000.

http://www.claymath.org/millennium-problems/p-vs-np-problem
Time Complexity

The Practical and Impractical

Review:
- **Polynomial** time difference between single-tape and multi-tape TMs
- **Exponential** time difference between deterministic and non-deterministic TMs.

**Polynomial** considered small.
- All reasonable, deterministic programming models are polynomially equivalent.
- “Reasonable” is loosely defined, but includes models that closely resemble running time on actual computers.

**Exponential** considered big.
- Common in brute-force searches.
The class P plays a central role in our theory and is important because

1. P is invariant for all models of computation that are polynomially equivalent to the deterministic single-tape Turing machine, and
   a. E.g., mathematically robust, unaffected by computation model.

2. P roughly corresponds to the class of problems that are realistically solvable on a computer.
   a. E.g., is practically relevant.
General Approach

Guidelines for proving that some problem/algorithm is in P

1. Give algorithms in numbered stages.
2. Give a polynomial upper bound (e.g., big-O) on the # of stages for input of length $n$.
3. Ensure that each stage can be completed in polynomial time on a reasonable deterministic model.
4. Input must be reasonably encoded.
Example: The **PATH** problem

\[ \text{PATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]

**Brute force:** Examine all possible paths and accept if one path is from \( s \) to \( t \).

**Too slow!!!**

**PROOF:**

M = “On input \( \langle G, s, t \rangle \), where \( G \) is a directed graph with nodes \( s \) and \( t \):

1. Place a mark on node \( s \).
2. Repeat the following until no additional nodes are marked:
   3. Scan all the edges of \( G \). If an edge \((a, b)\) is found going from a marked node \( a \) to an unmarked node \( b \), mark node \( b \).
   4. If \( t \) is marked, accept. Otherwise, reject.”

What is the total number of stages?

Can each stage be executed in polynomial time?
Example: Relative Primes (i.e., GCD is 1)

\[ \text{RELPRIME} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \} \]

\[ E = \text{“On input } \langle x, y \rangle, \text{ where } x \text{ and } y \text{ are natural numbers in binary:} \]
1. Repeat until \( y = 0 \):
2. Assign \( x \leftarrow x \mod y \).
3. Exchange \( x \) and \( y \).
4. Output \( x \).”

\[ R = \text{“On input } \langle x, y \rangle, \text{ where } x \text{ and } y \text{ are natural numbers in binary:} \]
1. Run \( E \) on \( \langle x, y \rangle \).
2. If the result is 1, accept. Otherwise, reject.”

If \( E \) runs in polynomial time, then so will \( R \).

How many stages are run by \( E \)?
- If \( x / 2 \geq y \), then \( x \mod y < y \leq x / 2 \) and \( x \) drops by at least half.
- If \( x / 2 < y \), then \( x \mod y = x - y < x / 2 \) and \( x \) drops by at least half.
- \( x \) and \( y \) are exchanged at end of loop.
- Max executions of 2 & 3 is \( \min(2 \log_2 x, 2 \log_2 y) \).

\# of stages is \( O(n) \).

Each stage of \( E \) is polynomial, therefore algorithm is polynomial.
Theorem: Every context-free language is a member of P.

Theorem 4.9 proved that every CFL is decidable.

Unfortunately, that algorithm tried all possible derivations, and is therefore exponential. :

Requires dynamic programming.

- Overlapping subproblems, optimal substructure.
- Often uses a matrix for memoization.

Create a $n \times n$ table. For $i \leq j$, the $(i,j)$th entry of the table contains the collection of variables that generate the substring $w_iw_{i+1}\cdots w_j$.

Algorithm is $O(n^3)$.

Let $G$ be a CFG in Chomsky normal form generating the CFL $L$.

$$D = \text{“On input } w = w_1 \cdots w_n:\n1. \text{For } w = \varepsilon, \text{if } S \rightarrow \varepsilon \text{ is a rule, accept; else, reject. [ } w = \varepsilon \text{ case] }\n2. \text{For } i = 1 \text{ to } n: \text{ [examine each substring of length 1] }\n3. \text{For each variable } A:\n4. \text{Test whether } A \rightarrow b \text{ is a rule, where } b = w_i.\n5. \text{If so, place } A \text{ in } \text{table}(i, i).\n6. \text{For } l = 2 \text{ to } n: \text{ [ } l \text{ is the length of the substring] }\n7. \text{For } i = 1 \text{ to } n - l + 1: \text{ [ } i \text{ is the start position of the substring] }\n8. \text{Let } j = i + l - 1. \text{ [ } j \text{ is the end position of the substring] }\n9. \text{For } k = i \text{ to } j - 1: \text{ [ } k \text{ is the split position] }\n10. \text{For each rule } A \rightarrow BC:\n11. \text{If } \text{table}(i, k) \text{ contains } B \text{ and } \text{table}(k + 1, j) \text{ contains } C, \text{ put } A \text{ in } \text{table}(i, j).\n12. \text{If } S \text{ is in } \text{table}(1, n), \text{ accept; else, reject.”}$$
The Class NP

Making complexity more complex.

There are problems for which we have not found polynomial algorithms.

- Maybe they don’t exist.
- Maybe we’re just not smart enough yet.
A Hamiltonian path in a directed graph $G$ is a directed path that goes through each node exactly once.

\[
\text{HAMPATH} = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t \} \]

What algorithm can we use to decide HAMPATH?

Is the algorithm polynomial?

Nobody has found a polynomial solution for HAMPATH. However, HAMPATH exhibits polynomial verifiability, which means that if a solution (path) were proposed, then it’s membership could be decided in polynomial time.

- I.e., verifying existence may be easier than determining existence.

HAMPATH is not polynomial, neither is it even polynomial verifiable.

- Verification would be no simpler than the original computation!
Verifiers

**DEFINITION:**
“A verifier for a language $A$ is an algorithm $V$, where $A = \{w \mid V$ accepts $\langle w, c \rangle$ for some string $c\}$. We measure the time of a verifier only in terms of the length of $w$, so a polynomial time verifier runs in polynomial time in the length of $w$. A language $A$ is polynomially verifiable if it has a polynomial time verifier.”

The symbol $c$ represents a **certificate**, or **proof**, of $w$’s membership in $A$.

In HAMPATH, $c$ would be the **path to verify** between $s$ and $t$.

**EXAMPLE:**
Composite numbers are numbers which are not prime.

$\text{COMPOSITES} = \{x \mid x = pq, \text{ for integers } p, q > 1\}$.

COMPOSITES is polynomially verifiable, because a divisor can serve as $c$.

(Note: It is not trivial to assume that COMPOSITES is in P. Why?)
Definition: NP

“NP is the class of languages that have polynomial time verifiers.”

NP does not mean Not-P. It is an unfortunate coincidence that causes confusion.

NP comes from Non-deterministic Polynomial, as related to non-deterministic, polynomial-time Turing Machines.

\[ P \subseteq NP \]

HAMPATH on a NTM:

\[ N_1 = \text{"On input } \langle G, s, t \rangle, \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t: \]

1. Write a list of \( m \) numbers, \( p_1, \ldots, p_m \), where \( m \) is the number of nodes in \( G \). Each number in the list is non-deterministically selected to be between 1 and \( m \).
2. Check for repetitions in the list. If any are found, reject.
3. Check whether \( s = p_1 \) and \( t = p_m \). If either fail, reject.
4. For each \( i \) between 1 and \( m-1 \), check whether \( (p_i, p_{i+1}) \) is an edge of \( G \). If any are not, reject. Otherwise, all tests have been passed, so accept.”
Theorem: NP Languages

“A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.”

IDEA: Show how to convert a polynomial time verifier to an equivalent polynomial time NTM and vice versa.

The NTM simulates the verifier by guessing the certificate.

The verifier simulates the NTM by using the accepting branch as the certificate.

Let \( A \in NP \) and show that \( A \) is decided by a polynomial time NTM \( N \). Let \( V \) be the polynomial time verifier for \( A \) that exists by the definition of NP. Assume that \( V \) is a TM that runs in time \( n^k \) and construct \( N \) as follows.

\( N = \) “On input \( w \) of length \( n \):
1. Nondeterministically select string \( c \) of length at most \( n^k \).
2. Run \( V \) on input \( \langle w, c \rangle \).
3. If \( V \) accepts, accept; otherwise, reject.”

\( V = \) “On input \( \langle w, c \rangle \), where \( w \) and \( c \) are strings:
1. Simulate \( N \) on input \( w \), treating each symbol of \( c \) as a description of the nondeterministic choice to make at each step (as in the proof of Theorem 3.16).
2. If this branch of \( N \)’s computation accepts, accept; otherwise, reject.”
NP is insensitive to the choice of a reasonable nondeterministic computation model, because they are polynomially equivalent.

Restrictions on algorithm stages are similar to those of class P.

Each stage must have an obvious implementation on a reasonable nondeterministic model.
**Example NP Problem: k-cliques**

**Clique:** fully-connected subgraph.

**K-clique:** clique containing exactly $k$ nodes.

(example picture is a 5-clique)

Problem is to determine if a k-clique exists in a graph.

**CLIQUE** =\{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}

**THEOREM:** CLIQUE is in NP.

**PROOF:**

Clique is the certificate.

$V = “\text{On input } \langle \langle G, k \rangle, c \rangle:\”$

1. Test whether $c$ is a subgraph with $k$ nodes in $G$.
2. Test whether $G$ contains all edges connecting nodes in $c$.
3. If both pass, accept; otherwise, reject.”
Example NP Problem: k-cliques (alternate proof)

Clique: fully-connected subgraph.

K-clique: clique containing exactly \( k \) nodes.  
(example picture is a 5-clique)

Problem is to determine if a k-clique exists in a graph.

\[ \text{CLIQUE =}\{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\} \]

THEOREM: CLIQUE is in NP.

PROOF:
Nondeterministically decide CLIQUE.

\[ V = \text{“On input } \langle G, k \rangle:\]
\[ 1. \text{ Nondeterministically select a subset } c \text{ of } k \text{ nodes of } G. \]
\[ 2. \text{ Test whether } G \text{ contains all edges connecting nodes in } c. \]
\[ 3. \text{ If both pass, accept; otherwise, reject.”} \]
Example NP Problem: SUBSET-SUM

SUBSET-SUM = {⟨S, t⟩ | S = {x₁, ..., xₖ}, and for some \{y₁, ..., yₙ\} ⊆ \{x₁, ..., xₖ\}, we have \(\sum y_i = t\)}.

Simply put: From a group of numbers, is there a subset of them that add up to a target number?

EXAMPLE:
⟨\{4, 11, 16, 21, 27\}, 25⟩ ∈ SUBSET-SUM because \(4 + 21 = 25\).

NOTE: Repetition is allowed, because \{x...\} and \{y...\} are defined as multisets.

PROOF: Verifier with subset as certificate.
\(V = \) “On input ⟨⟨S, t⟩, c⟩:
1. Test whether \(c\) is a collection of numbers that sum to \(t\).
2. Test whether \(S\) contains all the numbers in \(c\).
3. If both pass, accept; otherwise, reject.”

ALTERNATIVE PROOF: Nondeterministic TM.
\(N = \) “On input ⟨\(S, t\⟩:
1. Nondeterministically select a subset \(c\) of the numbers in \(S\).
2. Test whether \(c\) is a collection of numbers that sum to \(t\).
3. If the test passes, accept; otherwise, reject.”
coNP: The dark side.

Like ¬HAMPATH, ¬CLIQUE and ¬SUBSET-SUM are not obviously in NP.

We define coNP as the languages which are the complements of NP.

Problem: Verifying that something is not present seems much more difficult than verifying that something is present.

Reality:
We don’t know if coNP is different from NP.
P vs NP

P = the class of languages for which membership can be decided quickly.

NP = the class of languages for which membership can be verified quickly.

The best deterministic method currently known for deciding languages in NP uses exponential time. E.g.,

$$NP \subseteq \text{EXPTIME} = \bigcup_{k} \text{TIME}(2^{n^k})$$

https://www.win.tue.nl/~gwoegi/P-versus-NP.htm