**Nondeterministic Turing Machines**

All the complexity classes we have encountered so far are predicated on the natural assumption that a computation is deterministic: for every possible configuration, there is at most one next configuration.

Nondeterministic finite state machines indicate that this may not be the right way of looking at computation: it does make sense to have multiple potential next configurations. But, as we have seen, nondeterminism can be eliminated in the context of finite state machines, albeit at a possibly exponential cost.

So how about Turing machines?

---

**Details**

We want to define a nondeterministic Turing machine acceptor.

Classical Turing machines have transition functions of the form

$$\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{\pm 1, 0\}.$$  

To introduce nondeterminism, we switch to a more general transition relation

$$\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{\pm 1, 0\}.$$  

The idea being that \((p, a, q, b, d) \in \delta\) means that the machine could go to state \(q\), write symbol \(b\) and move the head by \(d\), but could possibly also do something else.

---

**Computation Trees**

In a nondeterministic Turing machine the one-step relation \(C \vdash_{1} C'\) is no longer single-valued: there may be multiple configurations \(C'\) at the next time step.

For each input \(x\), we obtain a computation tree \(T_x\), rather than just a linear sequence.

---

**Transducers versus Acceptors**

The critical question: how does this kind of machine compute?

If we are interested in computing a function \(f : \Sigma^* \rightarrow \Sigma^*\) there is a problem: what if different branches come up with different answers? This is quite tricky.

But there is one easy scenario: if we want a decision algorithm, a function \(f : \Sigma^* \rightarrow \mathbb{2}\), we can use the following definition:

The machine accepts \(x\) if there is at least one branch in \(T_x\) ending in the accept state.

This is a direct copy of the definition for nondeterministic finite state machines, which definition we already know to make perfect sense.
How should one define running time in the setting?

\[ T_M(x) = \min \{ |\beta| \mid \beta \text{ accepting branch in } T_x \} \]

and similarly for \( T_M(n) \).

The min really is justified here: we could make the right guesses and choose a short accepting path. Of course, there could be longer accepting paths, but they really don’t matter.

By definition, we say that a nondeterministic Turing machine accepts input \( x \) if there is at least one accepting branch in the computation tree \( T_x \).

Of course, there might be lots of non-accepting branches in the same tree. In fact, in most natural examples this will be the case: wrong nondeterministic choices lead to rejection, only the “right” sequence of choices ultimately leads to acceptance.

Note that we are not insisting on a certain fraction of all branches being accepting, the idea behind probabilistic computation; a single one is enough. More on probabilistic algorithms later.

If you like Sci-Fi you can think about a nondeterministic machine as running in a multiverse, an idea stolen from quantum physics.

Every time the machine makes a nondeterministic choice, the world splits into two strands, two branches that evolve in parallel.

After \( n \) choices, we have \( 2^n \) strands, \( 2^n \) copies of the machine running in parallel.

When they are all done, something collapses and the whole ghostly affair ends in acceptance if at least one of the branches did, and rejection otherwise.

We can clean up nondeterministic Turing machines a bit. First, it is easy to see that the number of choices at each step can be limited to two. Second, it is safe to assume that there are exactly two choices at each step. Yes?

In this situation, every branch in the computation tree is determined by a choice sequence, a bit-sequence \( S \in 2^* \). The accepting branch (if any) has length \( T_M(x) \).

Is there anything a nondeterministic acceptor can do that a deterministic one cannot? Yes and no.

**Theorem (Deterministic Simulation)**

Every nondeterministic Turing machine acceptor can be simulated by a deterministic one.

*Proof.*

Use the modified binary-choice machine from above. Systematically check all possible choice sequences \( S \in 2^* \). If one of these simulated computations ends in acceptance, return Yes; otherwise keep on searching.

Clearly, all of this can be handled by a deterministic Turing machine.

If we have a time or space bound, the search can be terminated at some point.

How much does it cost to eliminate the nondeterministic choice? Suppose \( f \) is the running time of the nondeterministic machine.

The running time of the deterministic simulator machine is \( O(2^c f(n)) \)

where \( c > 0 \) is some constant depending on the machine.

In other words, brute force deterministic simulation produces an exponential blow-up in compute time. For abstract computations (say, primitive recursive functions) this exponential slow-down is trifling. But for real algorithms it makes a huge difference.

**Warning:** the fact that the obvious deterministic simulation method is exponential does not mean that there might not be a better algorithm. This is most emphatically not a lower bound result.
A clear understanding of the difference between decidability and tractability first arose in the 1960s as a result of the increasing use of computers. Since then many algorithms have been developed that solve problems in polynomial time.

In some cases, finding the right algorithm may involve decades of research. The latest example is the polynomial time primality testing algorithm by Agrawal, Kayal and Saxena.

As it turns out, there is a rather large class of problems that are very nearly tractable but have resisted all efforts to find polynomial time algorithms. There are trivial exponential time algorithms, but the step down to polynomial time seems exceedingly difficult – though no one has been able to establish appropriate lower bounds.

Note that one can sometimes trade space for time: a computation can be made to use less memory by re-computing values rather than storing them.

For example, memoizing is a trick that goes in the opposite direction: avoid recomputation for the cost of an additional hash table.

Or space measure ignores these issues, we go for the least memory-intensive branch, no matter how long the computation is. Obviously, it would be of interest to combine both measures.

We can handle space complexity $S_M(x)$ in a similar manner:

$$S_M(x) = \min (|\beta| \mid \beta \text{ accepting branch in } T_x)$$

Here $|\beta|$ stands for the largest amount of memory used along branch $\beta$. Note that this does not necessarily mean the branch shorter than all others.

Similarly define $S_M(n)$.

We can now lift the definitions of deterministic time/space complexity classes to nondeterministic ones.

**Definition**

Let $f : \mathbb{N} \to \mathbb{N}$ be a (reasonable) function.

$$\mathrm{NTIME}(f) = \{ L(M) \mid M \text{ a nondeterministic TM, } T_M(n) = O(f(n)) \}$$

$$\mathrm{NSPACE}(f) = \{ L(M) \mid M \text{ a nondeterministic TM, } S_M(n) = O(f(n)) \}$$

As before, this generalizes easily to $\mathrm{NTIME}(F)$ and $\mathrm{NSPACE}(F)$.

More interesting is the question how much deterministic time/space is required to capture a nondeterministic class.

**Theorem**

Assume that $\log n \leq g(n)$. Then

$$\mathrm{NTIME}(g(n)) \subseteq \mathrm{SPACE}(g(n)^2).$$

**Sketch of proof**.

Again this is a simulation result.

The trick is to use a divide-and-conquer approach: a computation of length $t$ is broken up into two subcomputations of length $t/2$.

Since the target is a space class and space, unlike time, can be reused, one can show that the divide-and-conquer algorithm requires no more than $g(n)^2$ space.
It is generally agreed that the class of problems solvable in polynomial time is a reasonable formalization of feasible problem.

\[ P = \text{TIME}(\text{poly}), \]

If we allow nondeterminism we get

\[ \text{NP} = \text{NTIME}(\text{poly}) \]

The question is: is this definition at all useful? Clearly \(\mathbb{P} \subseteq \text{NP}\), but what is the difference?

Key rant on definitions.

**Déjà Vu All Over Again**

This is another example of projection: recall that for any semidecidable set \(A \subseteq \mathbb{N}\) there is a decidable set \(R \subseteq \text{NP}\) such that

\[ x \in A \iff \exists y \in \text{R}(x, y) \]

In general, \(A\) is not decidable since there is no way to bound the search for \(y\): the algorithm for \(R\) yields a semi-algorithm for \(A\).

In the \(\text{NP}\) scenario the search is trivially bounded, but it is entirely unclear that the bound could be made polynomial in the size of the input.

**Guess and Verify**

One interpretation of this definition is that a decision problem is in \(\text{NP}\) if we can “solve” it as follows. Given an instance \(x\):

- Guess a witness \(y\), a bit-string of length polynomial in the length of \(x\).
- Verify some property \(R(x, y)\) – which property can be tested in polynomial time.

Witness \(y\) corresponds to a particular branch in \(T_r\), and \(R(x, y)\) verifies that \(y\) really is a branch, and ends in the accept state. So for all Yes-instances there is an appropriate witness, but for No-instances no such witness exists.

So if we only could clairvoyantly produce the right witness, the whole decision procedure would take polynomial time. But a brute-force search over all potential witnesses takes exponential time.

**A Technical Detail**

One has to be a bit careful formalizing the “guess and verify” approach:

An \(A \subseteq \Sigma^*\) is in \(\text{NP}\), if there is a polynomial time decidable relation \(R\) and a polynomial \(p\) such that

\[ x \in A \iff \exists w \ ((|w| \leq p(|x|)) \land R(w, x)) \]

Thus, the witness must be relatively short.

Of course, the number of potential witnesses is still exponential and brute-force search is out.

**Quoi?**

The key here is that the running time of the algorithm for \(R\) is polynomial in \(|w| + |x|\), but the input for our decision problem is just \(x\).

In order to avoid cheating by witnesses of the form

\[ w = 00 \ldots 000 W \]

where \(W\) is the real witness we have to require short witnesses: \(|w| \leq \text{p}(|x|)\) for some polynomial \(p\).

**Exercise**

What would happen if we dropped the bound on witness length in the definition of \(\text{NP}\)? What class of problems would we describe?

**Asymmetry**

At any rate, we have the same asymmetry in membership versus non-membership that we saw in semidecidable sets. And we get similar closure properties. Translating this into resource-bound computation theory we would like:

**Lemma**

The polynomial time decidable sets are closed under intersection, union and complements. The nondeterministic polynomial time decidable sets are only closed under intersection and union.

More precisely, we don’t know whether nondeterministic polynomial time is closed under complements: it still could happen that \(\mathbb{P} = \text{NP}\) in which case we would trivially have closure.
Here is the easy part.

**Lemma**
The class $NP$ is closed under intersection and union.

**Proof.**
One can simply guess the two witnesses, and the run the two verification procedures. In the end, perform a simple Boolean $\lor$ or $\land$ operation on their outputs.

The whole operation is still polynomial time since polynomials are closed under addition.

This argument is actually a bit easier than for semidecidable sets since there is no problem with halting (union for semidecidable sets requires to run the machines in parallel by interleaving computations, here we can go sequentially).

---

**And Complement?**

But complementation runs into difficulties:

$$x \notin A \iff \forall w, |w| \leq p(|x|) (\neg R(w,x))$$

Of course, $\neg R$ is still polynomial time decidable.

But, there is no obvious reason why it should be possible to replace the universal quantifier by an existential one.

The class of problems with this characterization is called co-$NP$. A typical example is testing whether a Boolean formula is a tautology.

Of course, if $P = NP$ then $P = NP = \text{co-NP}$.

---

**Actual Computability is Hard**

Note that the analogous problem in classical recursion theory is much easier:

The classes of decidable, semidecidable and co-semidecidable sets are all distinct: the Halting set $K$ is semidecidable but not decidable, and therefore its complement is co-semidecidable but not semidecidable.

But the additional constraint of performing the computations in polynomial time makes things significantly more complicated. In particular, the classical diagonalization proof for the undecidability of $K$ simply does not carry over.

**Exercise**

Try to prove $P \neq NP$ by diagonalization and discuss what goes wrong.

---

**Gödel**

Here is a truly amazing fact: Kurt Gödel anticipated the $P$ versus $NP$ problem in 1956 in a letter to von Neumann (who was dying of cancer at the time). And he never bothered to publish.

See http://rjlipton.wordpress.com/the-gdel-letter/

---

**Vertex Cover**

Here is a classical example of an $NP$ problem in graph theory.

**Definition**
Let $G = (V, E)$ be an undirected graph.

A vertex cover for $G$ is a set $C \subseteq V$ such that $\forall \{v, u\} \in E (v \in C \lor u \in C)$.

Note that $V$ is always a vertex cover, so the interesting problem is to find a small cover.

You can think of this as a problem of trying to eavesdrop on all conversations in a network: we would like to place the minimal number of listening devices on some of the nodes of the network.
## Example

Nodes: 32, edges: 140, minimal cover size: 25

## Versions

There are several variants of the problem:

- **Search Version**: given $G$, compute a cover of minimal size.
- **Function Version**: given $G$, compute the lexicographically first cover of minimal size.
- **Decision Version**: given $G$ and a bound $k$, is there a cover of size $k$?

Arguable, the search version is the most natural one. But for complexity analysis, the decision version is much easier to handle.

## Oracles

Note that an oracle that solves the decision version can be used to solve the search version. Given a graph $G$:

- Do a binary search to determine the minimal size $k$ of a vertex cover.
- Find a vertex $v$ such that $G - v$ has a cover of size $k - 1$.
- Recurse.

This requires only a polynomial amount of work, plus the calls to the oracle. In particular a polynomial solution of the decision version provides a polynomial time solution of the search version.

## How Bad Can It Be?

We can check all $\binom{n}{k}$ subsets of size $k$. Each check is $O(n^2)$. Unfortunately, $\binom{n}{k}$ is not polynomial in $n$ for variable $k$, so this is not polynomial time ($k$ is a variable).

Of course, there might be smarter ways. E.g., we could start with the highest degree vertices. Surprisingly, none of the natural methods to approach this problem seem to succeed to push it down into polynomial time.

But Vertex Cover is trivially in $\text{NP}$: we guess the vertex cover and verify in polynomial time that is indeed a cover and has the right size.

**Exercise**

*Explain why the highest degree method does not always produce a minimal vertex cover. Try to come up with a better version.*

## Traveling Salesman (TSP)

We are given a $n$ by $n$ matrix of distances $d(i, j) \in \mathbb{N}^+$. Think of $[n]$ as a collection of cities, and $d(i, j)$ as the distance between city $i$ and city $j$.

Formally, a tour is permutation $\pi$ of $[n]$. For simplicity assume $\pi(1) = 1$. Thus we go from city 1 to $\pi(1)$, then $\pi(2)$ and so on till $\pi(n)$, then return to city 1.

The cost of this tour is $\text{cost}(\pi) = \sum_i d(\pi(i), \pi(i + 1)) + d(\pi(n), 1)$.

We want to find a tour of minimal cost.

In the decision version there is a bound $K$ and we ask whether there is a tour of cost at most $K$.

## Variants

In the Triangle TSP the distances are required to be symmetric and conform to the triangle inequality:

$$d(i, j) \leq d(i, k) + d(k, j).$$

In the Euclidean TSP we are given a finite set $S$ of points in the plane with integer coordinates, and the distances are defined to be the standard Euclidean distance between these points.

Thus, Euclidean TSP is a special case of Triangle TSP (we don’t really have to compute square roots!)

All these problems are trivially in $\text{NP}$, and not known to be in $\text{P}$.
A Hamiltonian cycle in an undirected graph $G$ is a cycle that contains every vertex of $G$ exactly once.

Don’t confuse Hamiltonian and Eulerian cycles: an Eulerian cycle is required to use every edge of the graph exactly once. While this may seem to be a minor difference, it most emphatically is not:

- One can test in linear time whether a graph has an Eulerian cycle: the graph only needs to be connected, and every node must have even degree.
- But there is no known polynomial time test for Hamiltonicity.

Exercise
Come up with a reasonable algorithm to test Eulericity (or is it Eulerianess?). Your algorithm should construct an Eulerian cycle if one exists, and return “No” otherwise.

With a little bit of effort, one can even construct an Eulerian cycle in linear time.

Suppose we have a procedure cycle($x$) that, given any vertex $x$, returns some arbitrary cycle anchored at vertex $x$. This is easy to do in time linear in the size of the cycle provided the graph is given in some reasonable representation such as adjacency lists.

Using cycle($x$) as a subroutine we can construct an Eulerian cycle as follows.

- Pick an arbitrary vertex $x$ and use cycle($x$) to obtain a cycle $C$ anchored at $x$.
- Remove all the edges on $C$ from $G$.
- Traverse $C$ until a vertex $y$ is found that still has edges.
- Splice the output of cycle($y$) into $C$.
- Repeat till all edges are used up.

Note that finding a Hamiltonian cycle seems to be a little easier than tackling TSP.

For suppose $G = (V, E)$ is an undirected graph. Think of the vertices as cities, and define the following distances:

$$d(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Then $G$ has a Hamiltonian cycle iff there is a TSP tour of cost $n = |V|$.

Here is a slightly more complicated example.

Consider an directed acyclic graph $G$. You want to place pebbles on all the nodes with out-degree 0 subject to the following rules:

- Initially all nodes are empty (unpebbled).
- Each node is pebbled at most once.
- A new pebble can be placed on node $x$ only if there are pebbles on all nodes $y$ such that $(y, x) \in E$.
- A pebble can be removed at any time.

Of course, there is a catch: you want to use the minimal number of pebbles (removed pebbles can be reused). So we have a decision problem: can $G$ be pebbled with $K$ pebbles?

Suppose we have a system of polynomial equations over $\mathbb{Z}_2$:

$$P_i(x_1, x_2, \ldots, x_n) = 0 \quad i = 1, \ldots, m.$$  

Note that one can check in polynomial time if a there exists a solution to a single polynomial equation over $\mathbb{Z}_2$. But if we look for a solution for the whole system of equations the problem is not known to be in $P$.

On the other hand, the problem is trivially in $NP$: guess the solution and evaluate all the polynomials to verify correctness.

Exercise
Find a polynomial time test for a single polynomial equation over $\mathbb{Z}_2$.  

Suppose we have a procedure cycle($x$) that, given any vertex $x$, returns some arbitrary cycle anchored at vertex $x$. This is easy to do in time linear in the size of the cycle provided the graph is given in some reasonable representation such as adjacency lists.

Using cycle($x$) as a subroutine we can construct an Eulerian cycle as follows.

- Pick an arbitrary vertex $x$ and use cycle($x$) to obtain a cycle $C$ anchored at $x$.
- Remove all the edges on $C$ from $G$.
- Traverse $C$ until a vertex $y$ is found that still has edges.
- Splice the output of cycle($y$) into $C$.
- Repeat till all edges are used up.

Note that finding a Hamiltonian cycle seems to be a little easier than tackling TSP.

For suppose $G = (V, E)$ is an undirected graph. Think of the vertices as cities, and define the following distances:

$$d(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Then $G$ has a Hamiltonian cycle iff there is a TSP tour of cost $n = |V|$.

Suppose we have a system of polynomial equations over $\mathbb{Z}_2$:  

$$P_i(x_1, x_2, \ldots, x_n) = 0 \quad i = 1, \ldots, m.$$  

Note that one can check in polynomial time if a there exists a solution to a single polynomial equation over $\mathbb{Z}_2$. But if we look for a solution for the whole system of equations the problem is not known to be in $P$.

On the other hand, the problem is trivially in $NP$: guess the solution and evaluate all the polynomials to verify correctness.

Exercise
Find a polynomial time test for a single polynomial equation over $\mathbb{Z}_2$.  

Suppose we have a procedure cycle($x$) that, given any vertex $x$, returns some arbitrary cycle anchored at vertex $x$. This is easy to do in time linear in the size of the cycle provided the graph is given in some reasonable representation such as adjacency lists.

Using cycle($x$) as a subroutine we can construct an Eulerian cycle as follows.

- Pick an arbitrary vertex $x$ and use cycle($x$) to obtain a cycle $C$ anchored at $x$.
- Remove all the edges on $C$ from $G$.
- Traverse $C$ until a vertex $y$ is found that still has edges.
- Splice the output of cycle($y$) into $C$.
- Repeat till all edges are used up.

Note that finding a Hamiltonian cycle seems to be a little easier than tackling TSP.

For suppose $G = (V, E)$ is an undirected graph. Think of the vertices as cities, and define the following distances:

$$d(u, v) = \begin{cases} 1 & \text{if } \{u, v\} \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Then $G$ has a Hamiltonian cycle iff there is a TSP tour of cost $n = |V|$.
The real goal is to minimize temporary storage when executing a straight-line program. For example, here is a circuit corresponding to a SLP for complex multiplication (given two pairs of real numbers):

\[
\begin{align*}
&\times \quad + \\
&\ 	imes \quad + \\
&\times \\
\end{align*}
\]

For simple arithmetic operations, the pebbles correspond to registers, so this is really about register allocation.

Recall that there is a simple linear time algorithm to “sequentialize” the tasks using just one processor: topological sorting.

But this does not answer our problem: it just says “execute the tasks in such and such order” and ignores resource issues entirely.

A minor variant of the problem allows one to shift one pebble from \( y \) to \( x \) given an edge \( (y, x) \in E \).

Makes essentially no difference (just one pebble).

A much different version of the Pebbling game allows one to re-pebble a vertex, arbitrarily often.

This corresponds to recomputing a value, rather than storing it in a register.

Needless to say, there is a trade-off: recomputation takes longer but may well require fewer pebbles than our version.

This is actually very important for simulations translating time into space, but we won’t go there.

Given a permutation \( \pi = x_1, \ldots, x_n \) of the vertices which describes a certain pebbling order, the corresponding minimal pebble sets are determined as follows:

\[
\begin{align*}
P_0 &= \emptyset \\
P_t &= P_{t-1} + x_t \quad \text{all useless pebbles in } P_{t-1}
\end{align*}
\]

A pebble at \( y \) is useless if all vertices \( x \) such that \( (y, x) \in E \) are already taken care of.

The size of \( P_t \) is considered to be the number of pebbles before the removal of the useless ones (that is, \( |P_{t-1}| + 1 \)).

Exercise

Verify that given the permutation \( \pi \) we can compute in polynomial time the optimal pebbling strategy that pebbles the vertices in order given by \( \pi \).
Bad Solution, Contd.

The pebble sets for the bad example. The top row indicates the stages of the construction.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The bottom row indicates the number of pebbles used at this stage, before removal of unneeded pebbles.

Pratt’s Theorem

Theorem

PRIME is in NP.

Proof.

procedure prime(n)
if n ≤ 10
then use brute force to test primality
else
guess a generator g of \( \mathbb{Z}_n^* \)

Guess the prime factorization of \( n − 1 \), say \( n − 1 = \prod p_i^{e_i} \).

verify that \( g^{n−1} \equiv 1 \text{ (mod } n) \)

verify that for all \( q = \frac{n−1}{p_i} \), \( g^q \not\equiv 1 \text{ (mod } n) \)

verify that all \( p_i \)'s are prime

if everything checks
then return Yes
else return No

Primality

It is clear that the problem COMPOSITE is in NP.

But how about PRIME? It is not clear that guessing helps in any way.

Some preliminaries: the multiplicative subgroup of \( \mathbb{Z}_n^* \):

\[ \mathbb{Z}_n^* = \{ x \mid 1 ≤ x < n, \gcd(x, n) = 1 \} \]

Euler's totient function \( \phi(n) = |\mathbb{Z}_n^*| \), order of \( x \in \mathbb{Z}_n^* \),

\[ \text{ord}(x) = \min(k ≥ 1 \mid x^k \equiv 1 \text{ (mod } n)) \]

\( g \) is a generator of \( \mathbb{Z}_n^* \) iff \( \text{ord}(g) = \phi(n) \).

E.g., \( n = 7 \) has \( g = 3 \), \( n = 13 \) has \( g = 2 \).

Proposition

\[ \phi(n) = \prod p_i^{e_i − 1} (p_i − 1) \]

where \( n = \prod p_i^{e_i} \).

Hence \( n \) is prime iff \( \phi(n) = n − 1 \) iff \( \mathbb{Z}_n^* \) has a generator of order \( n − 1 \).

Optimal Solution

Requires only 5 pebbles.

Example

Verification of primality of the \( p_i \)'s is done recursively by making calls prime\( (p_i) \). The arithmetic for each call to prime is clearly polynomial in \( \Theta(\log n) \). The total number of recursive calls is \( O(\log n) \) as all \( p_i \leq n/2 \).

Example

The following tables show that 733 is prime.

<table>
<thead>
<tr>
<th>( n = 733 )</th>
<th>( n = 61 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n − 1 = 2^2 \cdot 3 \cdot 61, g = 6 )</td>
<td>( n − 1 = 2^2 \cdot 3 \cdot 5, g = 2 )</td>
</tr>
<tr>
<td>( 6^{732} \equiv 1 \text{ (mod 733)} )</td>
<td>( 2^{60} \equiv 1 \text{ (mod 61)} )</td>
</tr>
<tr>
<td>( 6^{366} \equiv −1 \text{ (mod 733)} )</td>
<td>( 2^{30} \equiv −1 \text{ (mod 61)} )</td>
</tr>
<tr>
<td>( 6^{244} \equiv 425 \text{ (mod 733)} )</td>
<td>( 6^{180} \equiv 47 \text{ (mod 61)} )</td>
</tr>
<tr>
<td>( 6^{12} \equiv 299 \text{ (mod 733)} )</td>
<td>( 6^{12} \equiv 9 \text{ (mod 61)} )</td>
</tr>
</tbody>
</table>
Recall our Loop programming language from a while ago. Let’s say that two loop programs $P_1$ and $P_2$ are inequivalent if there is some input $x$ such that $P_1(x) \neq P_2(x)$.

In general, Inequivalence is undecidable for loop programs: Diophantine equations strike again. This requires nested loops to handle multiplication.

Define Loop$_1$ to be the collection of all loop programs where loops are not nested. The Diophantine argument no longer applies in this case.

So how difficult is Inequivalence for Loop$_1$ programs?

We could guess a witness $x$, and then check that $P_1(x) \neq P_2(x)$.

More precisely, it is not hard to see that given $x$ and a Loop$_1$ program $P$, we can evaluate $P(x)$ in time polynomial in the size of $P$ and $x$.

But how large is the shortest witness $x$?

There is a messy theorem that shows that a witness is indeed polynomial in the size of the programs, establishing membership in NP.

Membership in NP is not a lower bound, every polynomial time problem is trivially in NP.

But the ones we have just seen seem to be outside of $P$: a great many people have tried very hard for at least three decades to find fast algorithms, and they have all failed.

While separating $P$ from $NP$ is currently still an open problem (and arguably the most important problem in theoretical computer science, some would even give it very high rank in mathematics in general), there is quite a bit of evidence that suggests these problem are indeed not tractable.

We need a bit more machinery to compare the difficulty of problems to explain this evidence, though.

Because of the asymmetry in the definition of NP, it is unclear what happens under complementation:

$$co-NP = \{ \Sigma^* - L \mid L \in NP \}$$

Obviously, if $P = NP$, then $NP = co-NP$. It seems more likely, though, that $P \neq NP \neq co-NP$.

Another open problem.

Slightly dated, but still the best source for NP-completeness:

M. R. Garey and D. S. Johnson

Computers and Intractability

Freeman, 1979

Contains hundreds of examples of problems in NP (and worse). Many more are known today, but these are the most important ones.