We consider the possible relationships among the classes P, NP, and co-NP. First we consider properties of the class of NP-complete problems, as opposed to those which are in NP but not in P. We then consider the possible relationships of the three classes when relativized to an oracle.

1 NP-Complete Problems

There are ostensibly three possible situations involving the class of NP-complete problems. First, we could have P=NP, in which case every problem in P is NP-complete (allowing polynomial-time reductions instead of just log-space ones). Second, we could have P \( \neq \) NP and every problem in NP either in P or NP-complete. Third, we could have intermediate problems, ones which are neither in P nor NP-complete. We show that the second situation can not happen. We begin with a few preliminaries about enumerating complexity classes.

**Definition 1** We say that a class \( C \) of languages is syntactic if there is a recursive enumeration of the languages in \( C \), i.e. a TM \( M \) such that \( M(i) \) outputs the code of a TM \( M_i \) such that:

1. The language \( L_i \) decided by \( M_i \) is in \( C \).

2. For every language \( L \in C \) there is an \( i \) such that \( M_i \) decides \( L \).

We say that a class of functions is recursively enumerable if there is a similar listing of TM’s to compute the functions.

**Lemma 2**

1. P is syntactic, i.e. there is a recursive enumeration of the languages in P.

2. There is a recursive enumeration of the log-space functions.

**Proof:** For the first part, it suffices to enumerate the TM’s with polynomial running time. To do this, we enumerate all TM’s (as we have done many times) as well as all polynomials. For a TM \( M \) and a polynomial \( p(n) \), we construct the polynomial-time machine \( M_p \) by using \( p(n) \) as an “alarm clock”. On input \( x \) we simulate \( M \) for \( p(|x|) \) steps and then halt. If \( M \) originally had running-time bounded by \( p(n) \), then this will behave as \( M \) did; otherwise it will be some other polynomial-time machine. It is straightforward to check that this gives us all and only polynomial-time TM’s.

The second part is similar. We enumerate logarithmic functions instead of polynomials, and use these as “yardsticks” to limit the space used.

Most of the classes we have considered are syntactic. One class for which we do not know this is NP \( \cap \) co-NP; the problem is in determining whether two nondeterministic TM’s decide complementary languages.

**Theorem 3** If P \( \neq \) NP then there is a language \( L \) in NP which is not in P and is not NP-complete.
We now define $f$. Proof: We fix an enumeration $M_i$ of the polynomial-time TM’s (and hence an enumeration $L_i$ of the languages in P) and an enumeration $R_i$ of the log-space functions. We also fix a deterministic TM $S$ which decides the NP-complete language SAT (since we are assuming that $P \neq NP$, $S$ will not have polynomial running-time). We will describe a TM $K$ which decides the language $L$ we are constructing.

We will satisfy two types of requirements. First, to prevent $L$ from being in P, we will prevent $L$ from being equal to $L_i$, the language decided by $M_i$, for each $i$. Second, to prevent $L$ from being NP-complete, we will prevent each $R_i$ from being a reduction from SAT to $L$. Meeting these two requirements will give $L$ the properties we require.

We will define $K$’s behavior on input $x$ as a function $f : \mathbb{N} \rightarrow \mathbb{N} \times \{M, R\}$ by simultaneous recursion on length. The value of $f$ will keep track of which requirement we are trying to meet, either preventing $L$ from being decided by $M_i$ when $f(|x|) = (i, M)$, or preventing $R_i$ from reducing SAT to $L$ when $f(|x|) = (i, R)$. The values of $f$ will change only sporadically, and will always move from $(i, M)$ to $(i, R)$ or from $(i, R)$ to $(i + 1, M)$. We begin by setting $f(0) = (0, M)$. Assuming $f(n)$ has been defined, we define $K(z)$ for all $z$ with $|z| = n$ by

$$K(z) = \begin{cases} S(z) & \text{if } f(|z|) = M \\ NO & \text{if } f(|z|) = R \end{cases}$$

We now define $f(n + 1)$. We have two cases.

1. If $f(n) = (i, M)$ for some $i$, we check for $n$ steps for a $z$ with $|z| \leq n$ such that $M_i(z) \neq K(z)$. Here we count all steps necessary to produce $M_i$ in our enumeration, to successively enumerate the strings of length at most $n$, to compute $M_i(z)$, and to compute $K(z)$ inductively. We stop after $n$ steps even if we are in the middle of a simulation. If at this point we have found a $z$ as required, then we have met the current requirement and we set $f(n + 1) = (i, R)$ and proceed to the next requirement. If not, we continue to try to meet the current requirement and set $f(n + 1) = f(n) = (i, M)$.

2. If $f(n) = (i, R)$ for some $i$, we check for $n$ steps (counting as above) for a $z$ such that $|z| \leq n$, $|R_i(z)| \leq n$, and $S(z) \neq K(R_i(z))$. If found, we set $f(n + 1) = (i + 1, M)$; otherwise we set $f(n + 1) = f(n) = (i, R)$.

This completes the definition of $K$, and hence of $L$. We first check that $L$ is in NP. First we note that $f$ may be computed in linear time, since we are always checking for only $n$ steps. Since SAT is in NP we can thus compute the value $S(z)$ if needed, or simply output NO.

We next claim that $f$ eventually takes on all possible values. If not, then it is eventually constant. Suppose first that we have $f(n) = (i, M)$ for some $i$ and for all $n \geq n_0$. We claim that $K = M_i$ (more precisely, they decide the same language). For if there were a $z$ with $K(z) \neq M_i(z)$, then eventually $n$ would be large enough to find this $z$ in computing $f(n + 1)$, and the value of $f$ would change. So we would then have $L$ in P. We also have that $K(z) = S(z)$ for all $z$ with $|z| \geq n_0$, so that $L$ is equal to SAT for all but finitely many values of $z$. This would imply that SAT was in P as well, contrary to our assumption.
Second, if \( f(n) \) is eventually equal to \((i, R)\), we would have \( S(z) = K(R_i(z)) \) for all \( z \), i.e. SAT would be reducible to \( L \) via \( R_i \). We also would have \( K(z) = \text{NO} \) for all but finitely many \( z \)'s, so that \( L \) would be a finite language and hence in \( P \). Again, SAT would be in \( P \), a contradiction.

We lastly note that this implies that \( L \) is not in \( P \) (since otherwise \( L = L_i \) for some \( i \) and \( f \) would be stuck at \((i, M)\)), and \( L \) is not NP-complete (for otherwise SAT would be reducible to \( L \) via some \( R_i \) and \( f \) would be stuck at \((i, R)\)).

\[ \square \]

## 2 Isomorphism

Any two NP-complete languages are bi-reducible: each can be reduced to the other. We introduce a stronger notion, akin to recursive isomorphism as opposed to many-one reducibility.

**Definition 4** Let \( K, L \subseteq \Sigma^* \) be languages. We say that \( K \) and \( L \) are polynomially isomorphic, \( K \cong_p L \), if there is a bijection \( h : \Sigma^* \rightarrow \Sigma^* \) such that

1. For all \( x, x \in K \) iff \( h(x) \in L \).
2. Both \( h \) and its inverse \( h^{-1} \) are polynomial-time computable.

Note that \( h \) and \( h^{-1} \) are reductions for \( K \) to \( L \) and vice versa, but we are not requiring them to be log-space computable, so they are not literally reductions as we have defined them. Note that we could replace \( \Sigma^* \) by the set of codes for graphs or something similar, and that we could allow \( K \) and \( L \) to have different alphabets as well. Two languages which are polynomially isomorphic are structurally very similar. We are curious when we can find reductions between two NP-complete languages which are actually isomorphisms. We give an easy example:

**Proposition 5** CLIQUE \( \cong_p \) INDEPENDENT SET

**Proof:** As before, we can let \( h(G, K) = h(\bar{G}, k) \), where \( \bar{G} \) is the complementary graph. This is easily seen to be an isomorphism. \[ \square \]

We will first show that for essentially all of the languages of interest, we can find reductions which are injective. We will then prove an analogue of the Shroeder-Bernstein theorem for reductions.

**Definition 6** Let \( L \subseteq \Sigma^* \). We say that a function \( \text{pad} : (\Sigma^*)^2 \rightarrow \Sigma^* \) is a padding function for \( L \) if

1. \( \text{pad} \) is computable in log-space.
2. For all \( x \) and \( y \), \( \text{pad}(x, y) \in L \) iff \( x \in L \).
3. For all \( x \) and \( y \), \( |\text{pad}(x, y)| > |x| + |y| \).
4. There is a log-space algorithm to determine whether a given \( z \) is in the range of \( \text{pad} \), and to recover \( y \) from \( z = \text{pad}(x, y) \) if it is.

We say that \( L \) has a padding function if such a function exists for \( L \).
The idea is that a padding function is a length-increasing reduction from $L$ to itself which can also encode a string $y$. Note that pad need not literally be injective, but it is injective with respect to $y$.

**Lemma 7** SAT has a padding function.

**Proof:** Let a formula $\varphi$ and a string $y$ be given. For simplicity, we assume that $\Sigma = \{0, 1\}$. Suppose $\varphi$ has $m$ clauses and $n$ variables $x_1, \ldots, x_n$. We will pad $\varphi$ with $m + |y|$ additional clauses with $|y| + 1$ new variables. We let

$$\text{pad}(\varphi, y) = \varphi \land (x_{n+1} \land \cdots \land x_{n+1}) \land \left( \begin{array}{l} x_{n+1+i} \\ \neg x_{n+1+i} \end{array} \right) \land \cdots \land \left( \begin{array}{l} x_{n+1+|y|} \\ \neg x_{n+1+|y|} \end{array} \right)$$

where we use $x_{n+1+i}$ if $y(i-1) = 1$ and $\neg x_{n+1+i}$ if $y(i-1) = 0$. It is straightforward to check that pad is log-space computable, does not affect satisfiability, and is length-increasing. It is also easy to check whether a formula is in the range by looking for the long middle group of $x_{n+1}$’s, and then to decode $y$ using the last group of literals in case it is. □

**Lemma 8** CLIQUE has a padding function (for $K > 2$).

**Proof:** Let a graph $G$ and a goal $K > 2$ be given as well as a string $y \in \{0, 1\}^*$. We produce a new graph $G'$ and leave $K$ unchanged. Suppose $G$ has $n$ vertices, and the first is $v_1$. We first connect a chain of $n$ new vertices to $v_1$. We then add an additional tail of $|y|$ vertices to that, where the $i$’th new vertex has degree 3 or 4 depending on whether $y(i-1)$ is 0 or 1. See Figure 1. It is clear that this works, since we have left a copy of $G$ intact and have not added a clique of size more than 2, and $y$ can be recovered by scanning the tail vertices. □

Note that our padding depends on the presentation of $G$: We can have two isomorphic graphs $G_1$ and $G_2$ whose padded graphs (even with the same $y$) are not isomorphic, if the vertices are given in different orders.

Almost any natural language (including all of the ones we have seen so far) has a padding function. For instance, this is true of NODE COVER, HAMILTONIAN, TRIPARTITE MATCHING, and KNAPSACK.
Proposition 9 Suppose $R : K \leq L$ is a reduction and $L$ has a padding function. Then the function $R'(x) = \text{pad}(R(x), x)$ is a length-increasing, one-to-one reduction from $K$ to $L$. Moreover, there is a log-space computable partial inverse $R^{-1}$ which recovers $x$ from $R'(x)$.

Proof: All of these properties are immediate from the properties of a padding function. ∎

Theorem 10 Suppose $K, L \subseteq \Sigma^*$ are languages for which there exist one-to-one, length-increasing, log-space (partially) invertible reductions $R$ from $K$ to $L$ and $S$ from $L$ to $K$. Then $K \cong_p L$. In particular, this holds if $K$ and $L$ are bi-reducible and both have padding functions.

Proof: Let $R^{-1}$ and $S^{-1}$ be the partial inverses. For a string $x$ we define the $S$-chain of $x$ to be the sequence
\[
\langle x, S^{-1}(x), R^{-1}(S^{-1}(x)), S^{-1}(R^{-1}(S^{-1}(x))), \ldots \rangle
\]
where the sequence terminates when some term is undefined. Since $R$ and $S$ are length-increasing, their inverses must be length-decreasing. Hence, the strings in the $S$-chain have decreasing length, so the chain is finite for each $x$ (with length at most $|x| + 1$). We can define the $R$-chain of $x$ similarly.

We now define $h$. Given $x$, there are two cases:

1. If the last term of the $S$-chain of $x$ is of the form $S^{-1}(\cdots)$ (so that $R^{-1}$ of this is undefined) we let $h(x) = S^{-1}(x)$. This is well-defined, since $x$ must have been in the range of $S$.

2. If the last term of the $S$-chain is of the form $x$ or $R^{-1}(\cdots)$ (so that $S^{-1}$ of this is undefined) we let $h(x) = R(x)$.

We first check that $h$ is one-to-one. Suppose we had $x \neq y$ with $h(x) = h(y)$. We first note that $x$ and $y$ must have been defined by different cases, since if $h(x) = S^{-1}(x)$ and $h(y) = S^{-1}(y)$ then $S^{-1}$ would not have been one-to-one (impossible for an inverse), and if $h(x) = R(x)$ and $h(y) = R(y)$ then $R$ would not have been one-to-one (contrary to our assumption). So suppose $h(x) = S^{-1}(x)$ and $h(y) = R(y)$. Then $y = R^{-1}(S^{-1}(x))$, so that the $S$-chain of $y$ is a tail of the $S$-chain of $x$. Hence they end with the same string, and would have fallen into the same case, a contradiction.

We next check that $h$ is onto. Let $y$ be given, and consider the $R$-chain of $y$:
\[
\langle y, R^{-1}(y), S^{-1}(R^{-1}(y)), \ldots \rangle
\]
Suppose first that the last term is of the form $y$ or $S^{-1}(\cdots)$ with $R^{-1}$ of this undefined. Then the $S$-chain of $S(y)$ will contain the $R$-chain of $y$, and hence also end at the same point. Thus $h(S(y))$ is defined by case 1, so that $h(S(y)) = S^{-1}(S(y)) = y$, so $y$ is in the range. If, on the other hand, the $R$-chain of $y$ ends with a term of the form $R^{-1}(\cdots)$ with $S^{-1}$ of this undefined, then in particular $R^{-1}(y)$ is defined, and its $S$-chain is contained in the $R$-chain of $y$ and so ends at the same point. We then have $h(R^{-1}(y))$ defined by case 2, so $h(R^{-1}(y)) = R(R^{-1}(y)) = y$, and again $y$ is in the range.

Checking that $h$ is an isomorphism is immediate, since $R$ and $S^{-1}$ both are (partial) reductions from $K$ to $L$. Finally, we can compute $h$ in log-space by first computing the $S$-chain of $x$ (using...
the same sort of trick for composing log-space functions we have used before) to find where it ends and hence which case we are in; we then compute either $S^{-1}$ or $R$, which are both log-space. 

Corollary 11 The following are all polynomially isomorphic: SAT, CLIQUE, NODE COVER, HAMILTONIAN, TRIPARTITE MATCHING, KNAPSACK.

3 Density

We study a structural property of languages which is preserved under polynomial isomorphism, and which holds of all NP-complete languages if $P \neq NP$.

Definition 12 For a language $L \subseteq \Sigma^*$, we define its density function as

$$dens_L(n) = |\{x \in L : |x| \leq n\}|$$

Notice that if $|\Sigma| = k$, then $0 \leq dens_L(n) \leq k^n$ for all $n$. Recall the facts stated in an earlier homework about the zero-one law for the relative density of Horn-definable properties of graphs.

Proposition 13 If $K \cong_p L$, then $dens_K$ and $dens_L$ are polynomially related.

Proof: Let $f$ be an isomorphism from $K$ to $L$, and let $p(n)$ bound the length of $f$. Then

$$dens_K(n) = |\{x \in K : |x| \leq n\}| \leq |\{f(x) : x \in K \wedge |x| \leq n\}| \leq |\{y \in L : |y| \leq p(n)\}| = dens_L(p(n))$$

Similarly, with $q$ bounding the length of $f^{-1}$, we have

$$dens_L(n) \leq dens_K(q(n))$$

Thus the two densities are polynomially related. 

Definition 14 We say that a language $L$ is sparse if its density function is bounded by a polynomial. We say that $L$ is dense if its density function is super-polynomial.

The properties of being sparse or dense are preserved under polynomial isomorphism. Notice that it is possible for both a language and its complement to be dense. All the examples of NP-complete languages we have seen turn out to be dense (see the homework). There is a simple class of language which are all sparse, namely the unary languages, i.e. those whose alphabet $\Sigma = \{0\}$ contains only one symbol. Any unary language $U$ is sparse, since $dens_U(n) \leq n$. Recall here our comments about the subtleties of defining the length of an input in unary notation as opposed to $n$-ary notation for $n \geq 2$. Unary languages turn out to be (presumably) simpler than other languages, as shown by the following theorem.

Theorem 15 Suppose that some unary language $U \subseteq \{0\}^*$ is NP-complete. Then $P=NP$. 

6
Proof: Let $U$ be an NP-complete unary language (so in particular $U$ is in NP). Let $R$ be a log-space reduction from SAT to $U$ (the proof works as well if we only assume $R$ is polynomial-time). We may assume that $\emptyset$ is not in the range of $R$. We will describe a polynomial-time algorithm to decide SAT, which would imply that P=NP.

Recall how we used a putative polynomial-time algorithm for SAT to compute FSAT by (essentially) forming one branch in a tree of partial truth assignments. Let $\varphi$ be a formula with variables $x_1, \ldots, x_n$, and let $t \in \{0, 1\}^j$ define a partial truth assignment, i.e. we set $x_i$ true iff $t(i-1) = 1$ for $1 \leq i \leq j$. We let $\varphi[t]$ be the formula obtained by setting these variables to the values given by $t$, removing any false literals, and removing any true clauses (those with at least one true literal). Then $\varphi[t]$ is satisfiable iff there is a truth assignment extending $t$ which satisfies $\varphi$.

We give a first approximation to our algorithm. We will consider the problem SAT($\varphi, t$) of determining whether $\varphi[t]$ is satisfiable; in the end we evaluate SAT = SAT($\varphi, \emptyset$). Given $\varphi$ and $t$, we proceed as follows. If $|t| = n$ then we return YES if $\varphi[t]$ has no clauses (i.e. all clauses are true under $t$ and were removed); otherwise we return NO. If $|t| < n$, then we recursively evaluate SAT($\varphi, t \triangleright 0$) and SAT($\varphi, t \triangleright 1$) and return YES if one of these returned YES and return NO otherwise.

This algorithm is correct, but its worst-case running-time is exponential. We will speed up its running with a versatile trick, that of a hash table. As we proceed with the algorithm, we will keep track of formulas whose satisfiability we have already determined, in order to avoid repeating work. The hash table will only retain part of this information in order to be efficient, but will retain enough to speed up the algorithm.

We will define a function $H(t)$ and store values $(H(t), v)$ where $v$ is either YES or NO for $t$'s we have already evaluated. Our modified algorithm for SAT($\varphi, t$) will then be as follows. If $|t| = n$ then return YES if $\varphi[t]$ has no clauses; otherwise return NO. If $|t| < n$ we first check our table for a pair $(H(t), v)$; if one is found we return $v$. If no such pair is found, we recursively evaluate SAT($\varphi, t \triangleright 0$) and SAT($\varphi, t \triangleright 1$) and return YES if one of these returned YES and return NO otherwise. At the end, we add $(H(t), v)$ to our table, where $v$ is the value returned.

We then start with an empty table, and evaluate SAT($\varphi, \emptyset$). We require two properties of our function $H$:

1. If $H(t) = H(t')$ for two truth assignments $t$ and $t'$, then we need $\varphi[t]$ to be satisfiable iff $\varphi[t']$ is satisfiable. This is necessary for correctness of our algorithm.

2. The range of $H$ should be small, i.e. it should have size polynomial in $n$. This is necessary in order to search the table efficiently in order to speed up our initial algorithm.

We actually have such a function at hand, namely the reduction $R$ from SAT to $U$. Let

$$H(t) = R(\varphi[t])$$

This immediately satisfies property (1) since $R$ is a reduction, and it satisfies property (2) since for a given $\varphi$ the range satisfies

$$|\{H(t) : t \in \{0, 1\}^\leq n\}| \leq p(n)$$

7
where $p(n)$ is a polynomial bound on $R$. Thus, with this choice of $H$ our algorithm is correct; we now show that it has polynomial running time.

For a given $\varphi$, we can form a tree $T$ recording the recursive computation of $\text{SAT}(\varphi, \emptyset)$. Each node will correspond to a $t$ for which we check $\text{SAT}(\varphi, t)$ during the computation. We have $\emptyset$ at the top, and if $t \sim 0$ and $t \sim 1$ are checked, they occur immediately below $t$ in $T$. We say that one node $t_i$ is a prefix of a second node $t_j$ if $t_i$ occurs above $t_j$ in $T$, i.e. $t_j$ is a truth assignment extending $t_i$. The tree has depth at most $n$. Each invocation of $\text{SAT}(\varphi, t)$ requires (in addition to any recursion) order $p(n)$ steps, due to checking the hash table for $H(t)$. Thus, the running time of the whole algorithm will be $O(M \cdot p(n))$, where $M$ is the number of nodes in $T$. We will see that $M$ is small (polynomial in $n$).

Call a node $t$ in $T$ terminal if it has no nodes below it. Terminal nodes correspond to those $t$’s for which we do not have to perform recursion, i.e. those for which $|t| = n$ or $H(t)$ has previously been added to the hash table. Other nodes we call recursion nodes. We claim that we can find a set $N = \{t_1, t_2, \ldots\}$ of nodes in $T$ with the following properties:

1. $|N| \geq \frac{M-1}{2n}$
2. Each $t_i \in N$ is a recursion node.
3. No element of $N$ is a prefix of another element of $N$.

To see this, we build $N$ inductively. We start by removing all terminal nodes from $T$. Note that there are at most $\frac{M+1}{2}$ terminal nodes in $T$, since we can pair off each terminal node with a recursive node, leaving at most one terminal node left over. Next, we pick a node at the bottom of the tree and add it to $N$. We remove it and any prefixes of it from $T$. This removes at most $n$ nodes from $T$. We now repeat this process. We add at least $\frac{M-1}{2n}$ nodes, since we remove at most $n$ at each step and there were at least $\frac{M-1}{2}$ left after removing terminal nodes. No node in $N$ can be a prefix of another, since we removed all prefixes of a node when we added it to $N$, and all nodes in $N$ are recursion nodes.

We next observe that any two nodes $t_i$ and $t_j$ in $N$ are assigned different values by $H$. Suppose $t_i$ is computed first. Then, when we compute $t_j$, if it has the same $H$ value as $t_i$ we would find this in the hash table and stop, so that $t_j$ would be a terminal node, which it is not. Hence, our hash table must contain at least $\frac{M-1}{2n}$ values, but we know the size of the hash table is at most $p(n)$. Thus $\frac{M-1}{2n} \leq p(n)$, so $M \leq 2np(n) + 1$, and the running time of our algorithm is $O(2np(n)^2)$. □

This theorem can be extended to the case of any sparse language: If some sparse language is NP-complete, then P=NP.

4 Oracles

We now consider possible relationships among the classes P, NP, and co-NP in the presence of an oracle. We are here concerned with the ability of an oracle to speed computation, as opposed to allowing us to compute new (non-recursive) functions. We recall the definition of an oracle machine, with emphasis on running-time considerations.
**Definition 16** An oracle machine $M^{-}$ is like a regular TM, with the following changes. We have an oracle $A \subseteq \Sigma^{*}$ (which may be viewed as stored on a tape, but this is inessential), a query tape which can hold a string $\tau \in \Sigma^{*}$, and three new states: $q_{?}$, $q_{YES}$, and $q_{NO}$. When the machine is run with oracle $A$, we refer to it as $M^{A}$. The running is the same as an ordinary TM, except that when the machine enters state $q_{?}$ it consults the oracle to determine whether $\tau \in A$, where $\tau$ is the string on the query tape. If $\tau \in A$ the machine enters state $q_{YES}$; otherwise it enters state $q_{NO}$. Computation then proceeds.

Note that we can store other information (such as the state before we make a query) on a work tape. Each query of the oracle is counted as one computation step, although more steps may be necessary in order to write the string $\tau$ on the query tape.

**Definition 17** For a time-bounded complexity class $C$ and an oracle $A$, the class $C^{A}$ are those functions computed by machines in $C$ with oracle $A$. We refer to this as the class $C$ relativized to oracle $A$.

Space bounds and other complexity measures may also be defined relative to an oracle, but the definition is somewhat more subtle as we must determine how to measure the space used in queries. We shall not consider this here. We make several comments:

1. For any class $C$ and any oracle $A$, we have $C \subseteq C^{A}$, since we can simulate a normal TM by an oracle machine which never consults the oracle.

2. It is not true that whenever $C_{1} \subseteq C_{2}$, then $C_{1}^{A} \subseteq C_{2}^{A}$ as well. We will give an example below.

3. It is true that $P^{A} \subseteq NP^{A}$ for any $A$; this is immediate from the definitions of these classes.

4. We can also define compositions of oracles. We will only note the following: Assuming $C$ has reasonable closure properties, if $A \in C$ then $C^{A} = C$.

We will now show that the question of whether $P = NP$ can be decided in either way in the presence of oracles.

**Theorem 18** There is an oracle $A$ such that $P^{A} = NP^{A}$.

**Proof:** We need to make $P$ stronger, without making $NP$ even stronger. The idea is to try to eliminate any advantage of nondeterminism. One situation where we have seen that nondeterminism does not increase strength is in space bounds. Recall Savitch’s Theorem, that $PSPACE = NPSPACE$. We note the following:

**Fact:** There is a $PSPACE$-complete language. To see this, let

$$L = \{(M, x) : M \text{ accepts } x \text{ in space } |x|\}$$

Using compression, we can reduce any $PSPACE$ language to $L$. We note one more natural $PSPACE$-complete language, $QSAT$. This is the set of true propositional formulas of the form

$$\exists x_{1} \forall x_{2} \exists x_{3} \cdots Q_{n} x_{n} \varphi$$
where \( \varphi \) is a propositional formula with variables \( x_1, \ldots, x_n \). In particular, we can easily see that SAT is reducible to this by letting \( \varphi \) have only odd-indexed variables; similarly VALIDITY is reducible by letting \( \varphi \) have only even-indexed variables.

Let \( A \) be any PSPACE-complete language. Then we claim

\[
PSPACE \subseteq P^A \subseteq NP^A \subseteq \text{NPSPACE} \subseteq \text{PSPACE}
\]

Hence all of these classes coincide, so that \( P^A = \text{NP}^A \).

The first inclusion holds since \( A \) is PSPACE-complete; given a language in PSPACE we can reduce it to \( A \) and then query the oracle. The second inclusion is immediate, as noted earlier. The third inclusion holds since we can compute \( A \) in PSPACE whenever we need to answer a query.

The fourth inclusion is Savitch’s Theorem.

**Theorem 19** There is an oracle \( B \) such that \( P^B \neq \text{NP}^B \).

**Proof:** We want \( B \) to enhance the power of nondeterminism, and \( L \) to exploit this somehow. We will first define \( L \) in terms of \( B \):

\[
L = L_B = \{0^n : \text{there is an } x \in B \text{ with } |x| = n\}
\]

Note that \( L \) is a unary language. We see that, regardless of \( B \), \( L_B \in \text{NP}^B \), since we can guess an \( x \) with \( |x| = n \) and use the oracle to verify that \( x \in B \). We must define \( B \) so that \( L_B \notin P^B \).

We will use diagonalization. Let \( \langle M_0^-, M_1^-, \ldots \rangle \) be an enumeration of all polynomial-time deterministic oracle machines such that every machine appears infinitely often in the list. Here we mean that there are equivalent machines which behave identically to each other regardless of the oracle given. Also, we want the polynomial time-bound to be independent of the oracle used; we can use pairs \((M, p)\) of machines and polynomial alarm clocks as done earlier to ensure this. We will define \( B \subseteq \Sigma^* \) in stages, where \( \Sigma = \{0, 1\} \). At each stage \( i \) we will build the set \( B_i \) of strings in \( B \) of length at most \( i \). We will also keep track of a set \( X \subseteq \Sigma^* \) of exceptions, strings which we will require to not end up in \( B \). We start with \( B_0 = \emptyset \) and \( X = \emptyset \) initially.

Now, given \( B_{i-1} \) we define \( B_i \) to try to prevent \( M_i^- \) from deciding \( L_B \). We begin by trying to simulate \( M_i^-(0^i) \) for \( i^{\log i} \) steps. The significance of \( i^{\log i} \) is that it grows faster than any polynomial, but still slower than exponentially. The problem with the simulation is that \( M_i^- \) may query the oracle \( B \). Suppose that we are required to answer the question of whether \( x \in B \) for some \( x \). If \( |x| < i \) this may be determined from \( B_{i-1} \), and we return YES iff \( x \in B_{i-1} \). Otherwise, if \( |x| \geq i \) we answer NO and add \( x \) to the exceptional set \( X \). If we meet our commitment of keeping elements of \( X \) out of \( B \), our simulation will turn out to be correct.

We have three possibilities for the result of this simulation.

1. If the machine halts after at most \( i^{\log i} \) steps and rejects \( x \), then we want to make \( 0^i \in L_B \) to make \( L_B \neq L(M_i^B) \). We let

\[
B_i = B_{i-1} \cup \{x \in \{0, 1\}^i : x \notin X\}
\]
We need to check that we have added a string of length \(i\); if so \(0^i\) will be in \(L_B\) as desired. We only add elements to \(X\) when there is a query to \(B\); hence the size of \(X\) is bounded by the total lengths of previous simulations, i.e.

\[
|X| \leq \sum_{j=0}^{i-1} j^{\log j} < 2^i
\]

Hence there is a string of length \(i\) not in \(X\), so we are fine.

2. If the machine halts after at most \(i^{\log i}\) steps and accepts \(x\), then we want \(0^i \notin L_B\) to make \(L_B \neq L(M_i^B)\). This is achieved by setting \(B_i = B_{i-1}\).

3. If \(M_i^B(0^i)\) has not halted after \(i^{\log i}\) steps, then the polynomial bound \(p\) on \(M_i\) may be large enough that \(i^{\log i} < p(i)\). In this case we set \(B_i = B_{i-1}\). We have not achieved our goal for \(M_i^B\); however, since our enumeration of machines repeats them infinitely often (with the same polynomial bound) there will eventually be a \(j\) such that \(M_i^B = M_j^B\) and \(j^{\log j} \geq p(j)\) (since \(i^{\log i}\) grows faster than any polynomial). We will then ensure that \(L_B \neq L(M_j^B)\), and hence \(L_B \neq L(M_i^B)\).

We proceed in this fashion, and at the end set \(B = \bigcup_i B_i\). This completes the construction, and it is clear that we have prevented \(L_B\) from being in \(P_B^i\).

We note (without proof) the following:

**Theorem 20** There are oracles \(C\), \(D\), and \(E\) such that

1. \(P_C \neq NP_C\) and \(NP_C = \text{co-NP}_C\)
2. \(NP_D \neq \text{co-NP}_D\) and \(NP_D \cap \text{co-NP}_D = P_D\)
3. \(NP_E \neq \text{co-NP}_E\) and \(NP_E \cap \text{co-NP}_E \neq P_E\)

Hence, all of the possible relationships among these classes can be achieved with some oracle.

We make a few final comments about relativization. The above theorems do not settle the question of whether \(P = \text{NP}\) or not, but they do tell us something about the techniques necessary to settle this question. Informally, we say that a theorem relativizes if the theorem remains true when all appropriate notions are relativized to an arbitrary oracle. An example of this is the theorem that \(P \subseteq \text{NP}\), where this remains true for any other oracle.

A second example of this is the theorem that, assuming \(P \neq \text{NP}\), there is a language in \(\text{NP}\) which is not in \(P\) but not \(\text{NP}\)-complete. Here the relativization is more subtle. That proof used the \(\text{NP}\)-completeness of the language SAT. This language does not readily relativize; however, we can replace it by the following, which is always \(\text{NP}^A\)-complete:

\[
\{(M^A, x) : \text{the nondeterministic TM \(M\) accepts } x \text{ with oracle } A \text{ in time } |x|\}
\]

With this modification, the proof relativizes as well.
In fact, most results in complexity theory can be relativized, including all of the ones we have seen so far. There are some that do not, though. One example is the result that PSPACE = IP (Interactive Protocol). This proof does not relativize: There is an oracle under which the relativized classes are not equal. The proof of this result heavily relies on specific complete languages, and it is believed that this is the best approach to settling questions which are independent under oracles. For instance, SAT is a natural language to consider when trying to determine whether P = NP.