Correction.

Please write clearly. I may not give full credit to answers that are hard or impossible to understand, even though they are correct.

As a general principle, you should make sure that the essential argument is presented and stressed in such a way that the reader cannot miss it. On the contrary, if you really insist on having a low grade, don’t hesitate to leave the essential argument implicit, to bury it in the middle of a tedious ten page verification, to use undefined notation, to make unjustified claims (why should you cite a theorem or refer to a previous question when it is clear that the result holds?), or to write in an undecipherable hand writing.

In the solution, I will write the essential argument(s) in bold, and in the example above. You could have done the same (in \LaTeX) or something similar (underlining, for example), right? Most of you didn’t.

Anyway, the point of the “essential argument” point is the same as the “Warning, hot” mention on coffee cups at McDonald’s (averting lawsuits, see https://en.wikipedia.org/wiki/Liebeck_v._McDonald's_Restaurants): if I don’t understand your argument, I can spare some wild guessing as to what you really meant, some head-banging, and consider you didn’t answer the question—even though your argument may be correct.

I am not claiming that what is below is clear either, although I did my best.

1 Sparse languages and circuits

Here and in later sections, and will write each question with the number of lines that my solution takes. I may be more verbose than you, or not. This is just an indication.

We fix an alphabet $\Sigma$ with at least two letters. $\Sigma^n$ is the set of words of length $n$, and $\#E$ denotes the number of elements in the finite set $E$. For simplicity, we shall assume that $\Sigma = \{0, 1\}$, although this is not strictly necessary.

A language $L \subseteq \Sigma^n$ is sparse if and only if $\#(L \cap \Sigma^n)$ is bounded by a polynomial in $n$. Let $\text{SPARSE}$ be the class of all sparse languages.

1. Show that $\text{P/poly}$ is not contained in $\text{SPARSE}$. (1 line)
The language $\Sigma^*$ is not sparse, but can be decided with circuits of constant size.

2. Show that SPARSE contains some undecidable languages. (3 lines)

**Essential argument: as for $\text{P/poly}$.** We have seen that $\text{P/poly}$ contains some undecidable languages, notably the language $L$ of strings $1^n$ such that $n$ is the Gödel encoding of a halting Turing machine. $L$ is sparse because $\#(L \cap \Sigma^n) = 1$.

3. Let $L \in \text{P/poly}$. Show that there is a sparse language $S$ such that $L \in \text{P}^S$. (18 lines)

Let $L$ be decided by a family of polynomial circuits $C_n$, $n \in \mathbb{N}$. Say that $C_n$ is representable on $p(n)$ bits, where $p$ is a polynomial. One possibility is to **use the oracle to reconstruct $C_n$, bit by bit.** This idea leads to a sparse language $S$ that we define below.

Fix a constant $a$ such that $\log p(n) \leq n + a$ for every $n$ (more formally, $p(n) \leq 2^{n+a}$). The point is that all the numbers between 0 and $p(n) - 1$ are representable in binary using $n + a$ bits. Another key point in the argument we propose here is that there is at most one natural number $n$ such that $2n + a + 1$ equals any given natural number. All that will be explained below.

We let $S$ be the language of all those words $1^n0\overline{y}$, where bit $y$ of the representation of $C_n$ is 1. Here $\overline{y}$ is the binary representation of $y$ on exactly $n + a$ bits (not more, not less).

$S$ is a sparse language: given $m \in \mathbb{N}$, the only words in $S$ of length $m$ are those such that $2n + a + 1 = m$ (only one possibility for $n$), the prefix $1^n0$ is then unique, and there are at most $p(n)$ possible values for $y$ such that $1^n0\overline{y} \in S$.

To decide $L$ in $\text{P}^S$, for $y$ from 0 to $p(n) - 1$, we query the oracle on $1^n0\overline{y}$. This gives us bit $y$ of the circuit $C_n$. We need $p(n)$ calls to the oracle for that. Then we evaluate $C_n$ on the input, in polynomial time.

Some of you spent some time converting circuits to special formats, trying to bound the size of a string representing a circuit in terms of its number of gates and wires. Why not, but that is useless: by definition, the size of a circuit is the number of bits needed to describe it (just like any other data structure in complexity theory, lists, graphs, formulae, whatever).

Some of you said the circuit existed, but did not build it explicitly. That is of course required, even if you just give an informal construction as I did above.

Some of you build a machine instead of a circuit. This is impossible, since SPARSE contains languages that are not accepted by any machine—there are uncountably many sparse languages, and only countably many machines.

4. Show that SPARSE $\subseteq \text{P/poly}$. (7 lines)
Let $S$ be a sparse language, say $\#(S \cap \Sigma^n) = p(n)$. We define circuit $C_n$ as follows. For each $w \in S \cap \Sigma^n$, we build a circuit $C_w$ that recognizes just $w$. This is just a big, $n$-ary and gate. Its input number $i$ is negated if bit $i$ of $w$ is 0, otherwise it is not. $C_w$ contains at most $n+1$ gates, and is therefore of size at most $O(n \log n)$.

Then we build a big, $p(n)$-ary or gate with inputs the outputs of all the circuits $C_w$. This contains at most $p(n)(n+1) + 1$ gates, and has size at most $O(np(n) \log n)$.

5. Show that $\mathsf{P}^{\mathsf{poly}} \subseteq \mathsf{P}/\mathsf{poly}$. By this, we mean that for every language $S$ in $\mathsf{P}/\mathsf{poly}$ (say, decided by a family of polynomial circuits $C_m$, $m \in \mathbb{N}$, say of size $q(m)$), every language $L \in \mathsf{P}^S$ is in $\mathsf{P}/\mathsf{poly}$. (9 lines, but I’ve really only said the essential points in my solution)

This was meant to be a difficult question. My initial idea, and what most of your did, was to imitate the proof of $\mathsf{P} \subseteq \mathsf{P}/\mathsf{poly}$.

The only new ingredient is the call to oracles. In that case, we plug circuit $C_m$ to the bits of the query tape to the oracle, and we obtain the answer (yes/no) as output of $C_m$.

The number $m$ may vary from call to call, but will always be at most $p(n)$, where the machine deciding $L$ runs in polynomial time $p(n)$. And there will be only $p(n)$ many calls to the oracle. So the final size of a circuit deciding inputs of size $n$ of $L$ is $O(p(n)^2(p(n) + q(p(n))) + p(n)^2)$ (counting the $O(p(n)^2)$ of the rest of the computation).

There is a difficulty with whether to call an oracle, and which $C_m$. A solution is to use multiplexer gates. Nobody (including me) described how that should be done precisely, but nobody lost any points for that.

A simpler solution is as follows. This was a solution proposed by one of you. Let $M$ be a $p(n)$-time machine with an $S$ oracle. Build a new machine $M’$ that takes tuples $(x, u_0, u_1, \ldots, u_{p(n)})$ as input, and works as $M$ on input $x$, except that whenever $M$ queries the oracle (on some query of size $m \leq p(n)$), instead $M’$ decodes $u_m$ as a circuit and evaluates that circuit on the query. $M’$ works in polynomial time, hence defines a language $L’$ with polynomial circuits $C’_n$ (to make things simpler, we take $C’_n$ the circuit that simulates $M’$ on inputs $(x, u_0, u_1, \ldots, u_{p(n)})$ where $x$ is of size $n$; $n$ is not the size of the input of $M’$). For inputs $x$ of size $n$, plug the binary encodings of $C_m$ for each $u_m$ into the adequate inputs of $C’_n$, and voilà: a polynomial-sized circuit for all inputs $x$ of size $n$.

6. Conclude that $\mathsf{P}/\mathsf{poly} = \mathsf{P}^\text{SPARSE}$. (4 lines)

Easy consequence of the previous questions. The inclusion $\mathsf{P}/\mathsf{poly} \subseteq \mathsf{P}^\text{SPARSE}$ is Question 3. For the converse, any language in $\mathsf{P}^\text{SPARSE}$ is in $\mathsf{P}^S$ for some sparse language $S$, hence in $\mathsf{P}^{\mathsf{P}/\mathsf{poly}}$ by Question 4, hence in $\mathsf{P}/\mathsf{poly}$ by Question 5.
7. Assume $\Pi_2^p \neq \Sigma_2^p$. Show that no $\textbf{NP}$-complete problem is polynomial time Turing reducible to any sparse language. (5 lines)

If $L$ was such an $\textbf{NP}$-complete problem, then $L$ would be in $\textbf{P}^{\textbf{SPARSE}}$. Since the latter is trivially closed under polynomial time reductions, $\textbf{NP}$ would be included in $\textbf{P}^{\textbf{SPARSE}}$, hence in $\textbf{P}/\text{poly}$ by Question 6. The Karp-Lipton theorem (Proposition 1.22 in the lecture notes) then implies $\Pi_2^p = \Sigma_2^p$, contradicting our assumption.

Some of you made the assumption that $L$ is Karp-Cook polynomial time reducible to a sparse language. Some of you concluded that $\textbf{NP} \subseteq \textbf{SPARSE}$, but $\textbf{SPARSE}$ is not closed under polynomial time reductions.

2 Mahaney’s theorem

The purpose of this section is to examine in which cases SAT is polynomial time Cook-Karp reducible to a sparse language (to be precise, a non-empty one). The Cook-Karp reductions are those we are accustomed to work with, and which we simply call (polynomial time) reductions in the lectures. I will again just say “polynomial time reducible” in the sequel.

The difference with Turing reducibility is best explained as follows: in a Cook-Karp reduction, the oracle is called just once, and its answer is our final answer (we cannot even take the negation of the oracle’s answer).

8. Show that, if $\textbf{P} = \textbf{NP}$, then SAT is polynomial time reducible to some non-empty sparse language. (This is a very stupid reduction.) (3 lines)

In that case, SAT is in $\textbf{P}$. As reduction, we just solve the SAT instance $x$ in polynomial time. If $x$ is satisfiable, then we return, say, 1, otherwise 0. This is a polynomial time reduction to the sparse language $\{1\}$...

Some of you showed the more general fact that, if $\textbf{P} = \textbf{NP}$, then SAT is polynomial time reducible to every non-empty sparse language. That is true, too, but not what I asked from you. In principle, I should only give credit if you give me the explicit description of a sparse language to which SAT reduces. That was not hard to do.

Given a finite list of propositional variables $A_1, \ldots, A_m$, we define a binary tree $T_m$ that organizes the various truth-value assignments $\rho$ on $A_1, \ldots, A_m$. Instead of defining it formally, let us give an example. Here is $T_3$: 

4
The leaves of $T_m$ are in bijection with the (complete branches) from the root, and define truth-value assignments. For example, leaf 11 defines the environment $A_1$ false, $A_2$ and $A_3$ true.

The nodes $T_m$ (not just the leaves) are, accordingly, called partial assignments $\sigma$. For example, node 5 maps $A_1$ to false, $A_2$ to true, and does not map $A_3$ to anything.

We say that the leaf $\rho$ is to the left of the node $\sigma$ if and only if $\rho$ is a descendant of node $\sigma$, or graphically to the left of $\sigma$. For example, the leaves that are to the left of node 5 are 8, 9, 10, 11. The descendants of 5 are 10 and 11. (Formally, if we encode assignments and partial assignments as words over $\{0, 1\}$, then “to the left of” is a variant of the lexicographic ordering.)

Let LEFT-SAT be the following language.

INPUT: a pair of a set $E$ of propositional clauses, a partial assignment $\sigma$.

QUESTION: is there an assignment $\rho$ to the left of $\sigma$ that satisfies $E$?

9. Show that LEFT-SAT is NP-complete. (6 lines)

It is in NP: guess $\rho$, then check that it satisfies $E$, and that it is to the left of (lexicographically smaller than) $\sigma$. Concretely, we can also: fix $\rho(A_1), \ldots, \rho(A_i)$ to false where $A_1, \ldots, A_i$ are the longest prefix of variables set to false by $\sigma$; then we guess $\rho(A_{i+1}), \ldots, \rho(A_m)$. That concrete solution avoids the need to explain why “being to the left of” is polynomial-time decidable.

It is NP-hard. For that, we build a reduction from SAT to LEFT-SAT: on input $E$ (to SAT), we produce $(E, \epsilon)$ where $\epsilon$ is the empty partial assignment (the root of the tree). The essential idea is: there is a satisfying assignment at all if and only there is one to the left of $\epsilon$.

Some of you gave a wrong formal definition of what it meant to be “to the left of”, or failed to verify that condition in checking that LEFT-SAT is in NP. Some even failed to check that altogether.

For the NP-hardness part, you have to produce a reduction. Some of you directly described a machine that solved the problem. That was not what was required.

10. Let $E, \sigma \mapsto f(E, \sigma)$ be some polynomial time reduction from LEFT-SAT to some sparse language $S$. Let $E$ be a set of propositional clauses on variables $A_1, \ldots, A_m$. We define the level of a partial assignment as its distance from the root of $T_m$. So node 5 in our example is at level 2, and node 11 is at level 3.
Let \( \sigma, \sigma' \) be two nodes of \( T_m \) at the same level \( \ell \). Assume \( \sigma \) is strictly to the left of \( \sigma' \) (graphically), and that \( f(E, \sigma) = f(E, \sigma') \). Assume also there is a satisfying assignment \( \tau \) to \( E \), and take one that is leftmost. Can \( \tau \) be a descendant of \( \sigma' \)? (4 lines)

Let \( w = f(E, \sigma) = f(E, \sigma') \). If \( w \) is in \( S \), then there is a satisfying assignment to the left of both \( \sigma \) and \( \sigma' \), and the leftmost one is therefore to the left of \( \sigma \). Otherwise, \( w \) is not in \( S \), then \( \tau \) is not to the left of \( \sigma' \). In any case, \( \tau \) is not a descendant of \( \sigma' \).

Some of you used what is, in fact, a more direct argument: assume the leftmost satisfying assignment \( \tau \) is a descendant of \( \sigma' \); in particular, \( f(E, \sigma) \) is in \( S \); since \( f(E, \sigma) = f(E, \sigma') \), \( f(E, \sigma') \) is also in \( S \), so there is a satisfying assignement to the left of \( \sigma' \), contradicting the fact that \( \tau \) is leftmost.

11. Let \( E, \sigma \mapsto f(E, \sigma) \) be some polynomial time reduction from LEFT-SAT to some sparse language \( S \), say \( S \cap \Sigma^n \) is of cardinality \( p(n) \), where \( n \) is the size of \( E \), and \( p \) is a polynomial. Say that \( f \) works in time \( q \) of its input size, for some polynomial \( q \). How many distinct elements of \( S \) at most can be obtained as \( f(E, \sigma) \) when \( \sigma \) ranges over all the partial assignments to \( E \)? I am expecting a big \( O \), not an exact formula. (7 lines)

Since \( \sigma \) is coded on at most \( n \) bits, the size of the pair \( E, \sigma \) is at most \( 2n + 1 \). In time \( q(2n + 1) \), \( f(E, \sigma) \) can only produce a word of length at most \( q(2n + 1) \). The number of elements of \( S \) of at most that length is the sum \( p(0) + p(1) + \cdots + p(q(2n + 1)) \).

We now use the fact that \( 0^k + 1^k + \cdots + j^k \leq \frac{1}{k+1}(j+1)^{k+1} \) (upper bound the sum by an integral, say), so that the required number is \( O(q(2n + 1)p(q(2n + 1))) \).

We shall later use the fact that this is polynomial in \( n \).

Some of you found slightly different formula. \( O(q(n)p(q(n))) \) and slight variants were also correct, but \( O(p(q(n))) \), \( O(np(q(n))) \) and \( O(mp(q(n))) \) were not.

12. Let \( E, \sigma \mapsto f(E, \sigma) \) be some polynomial time reduction from LEFT-SAT to some sparse language \( S \). For a given \( E \) of size \( n \), let us call \( a(n) \) the polynomial found in Question 11. Let \( A_1, \ldots, A_m \) be the propositional variables in \( E \).

Imagine that we have a list \( L_\ell \) of partial assignments \( \sigma \) at level \( \ell < m \), ordered from left to right, with the guarantee that: (a) the words \( f(E, \sigma), \sigma \in L_\ell \), are all distinct, (b) \( L_\ell \) contains at most \( a(n) \) elements, and (c) if \( E_\ell \) has a satisfying assignment, then the leftmost one is a descendant of some \( \sigma \in L_\ell \).

Show that you can build, in polynomial time, a new list \( L_{\ell+1} \) satisfying (a), (b), and (c) but at level \( \ell + 1 \). Condition (b) is the hardest to satisfy. Question 10 should lead you to remove certain duplicates, i.e., certain nodes \( \sigma' \) at level \( \ell + 1 \) that have the same \( f(E, \sigma') \) value as some others. If this is not enough, you should remove enough nodes from one side of the list you have obtained. (19 lines)
We first form the list \( L' \) of partial assignments \( \sigma_0 \) and \( \sigma_1 \), for \( \sigma \in L \), organized from left to right.

By Question 10, for every pair of partial assignments \( \sigma' \) and \( \sigma'' \) in \( L' \), with \( \sigma' \) strictly to the left of \( \sigma'' \), if \( f(E, \sigma') = f(E, \sigma'') \), then we can safely remove \( \sigma'' \) from the list. Removing these duplicates can be done in polynomial time since \( L' \) contains at most \( 2a(n) \) elements, by (b).

After this duplication removal process, we obtain a new list \( L'' \), which satisfies (a) and (c). If (b) is satisfied as well, then we have found the desired \( L_{\ell+1} \).

Otherwise, not all words \( f(E, \sigma') \), \( \sigma' \in L'' \) can be in \( S \), by Question 11. Since the list \( L'' \) is ordered from left to right, all the nodes of \( L'' \) whose \( f \) value is not in \( S \) will be to the left of all those whose \( f \) value is in \( S \). This is because of the semantics of LEFT-SAT: any node \( \sigma'' \) strictly to the right of a node \( \sigma' \) at the same level with an \( f \) value in \( S \) must also have an \( f \) value in \( S \).

Therefore it is enough to let \( L_{\ell+1} \) be the list of the \( a(n) \) rightmost elements of \( L'' \). \( L_{\ell+1} \) will then satisfy (b), and clearly (a). It will also satisfy (c), because the nodes \( \sigma \) such that the leftmost satisfying assignment is a descendant of \( \sigma \) are amongst the last \( a(n) \) distinct nodes of \( L'' \).

Note that the latter requires one to count up to \( a(n) \) in polynomial time. But polynomials are proper, hence computable in polynomial time.

Some of you wrongly claimed that (a) implies (b). This would be the case if we knew that \( f(E, \sigma) \) was always in \( S \). But remember: that \( f \) is a reduction from LEFT-SAT to \( S \) does not mean it is a function that takes its values in \( S \); it just means that \( f(x) \in S \) iff \( x \) is in LEFT-SAT (and we call it on instances \( x \) that may be outside the language LEFT-SAT).

13. Using the previous questions, show that if LEFT-SAT reduces in polynomial time to some sparse language \( S \), then \( P = NP \). (8 lines)

Under the given assumption, we solve SAT in polynomial time as follows. On input \( E \), with propositional variables \( A_1, \ldots, A_m \), we compute in variable \( L \) the successive lists \( L_0, L_1, \ldots, L_m \) given in Question 12, level by level. Initially, \( L \) is \( L_0 \), which is just the list containing the root node. For \( \ell \) ranging from 0 to \( m-1 \), knowing that \( L \) contains \( L_\ell \), we compute \( L_{\ell+1} \) as in Question 12, and store it in \( L \). At the end of the loop, \( L \) contains \( L_m \), and we accept if and only if some \( \tau \in L \) satisfies \( E \). This is correct because of property (c), and the fact that the descendants of nodes at level \( m \) are the nodes themselves.

Some of you did not make the algorithm explicit. You may think the algorithm is obvious, considering the previous questions, but stating the algorithm is (at least) half of the question. Some of you confused SAT and LEFT-SAT and, for example, reduced SAT (which is fine) but said they would reduce LEFT-SAT (that was another fine possibility). Some attempted to reduce LEFT-SAT, but did not say from which level they started, and what they did with the first few levels.
14. Conclude Mahaney’s Theorem: SAT reduces in polynomial time to some sparse language if and only if $P = NP$. (4 lines)

The if direction is Question 8. Conversely, if SAT reduces in polynomial time to some sparse language $S$, then **LEFT-SAT reduces** to SAT (because LEFT-SAT is in $NP$; we do not need the fact that it is $NP$-complete), hence to $S$. By Question 13, $P = NP$.

Some of you built an explicit reduction from LEFT-SAT to SAT. That is overkill, and maximizes your chances of having it wrong.

15. Here is an alternative argument showing that if SAT is reducible to a sparse language, then $P = NP$. Let $g$ be a polynomial time reduction from SAT to $S \in SPARSE$. Given a set $E$ of propositional clauses on variables $A_1, \ldots, A_m$, we call $g$ on $E[A_1 := 1]$ (meaning $E$ where $A_1$ was replaced by true, and simplifying); if the result is a word in $S$, which we can do since $S$ only contains polynomially many words to check for equality, then we set $A_1$ to 1 and replace $E$ by $E[A_1 := 1]$, otherwise we set $A_1$ to 0 and replace $E$ with $E[A_1 := 0]$; we proceed in the same way with $A_2, \ldots, A_m$, until we reach a set of clauses with no variable. There are only two such sets, the empty set or the set containing just the empty clause. In the first case, we declare $E$ unsatisfiable. In the second case, we declare $E$ satisfiable. What is the problem with this argument? (3 lines)

Although $S$ “only contains polynomially many words to check for equality”, $S$ may fail to be a decidable language. For example, the set of unary encodings of halting Turing machines is a sparse, undecidable language.

3 PSPACE and sparse languages

Recall that a QBF formula is of the form $F = \exists \vec{A}_1. \forall \vec{A}_2. \exists \vec{A}_3. \cdots Q_m \vec{A}_m. G$, where $\vec{A}_1, \vec{A}_2, \vec{A}_3, \ldots, \vec{A}_m$ are finite disjoint lists of propositional variables, $G$ is a finite set of clauses over the propositional variables in the set $\bigcup_{i=1}^m \vec{A}_i$, and all the quantifiers (in particular $Q_m$) are chosen among $\{\forall, \exists\}$. We see $F$ as the specification of a game between $\exists$ and $\forall$. The $\exists$ player wins iff $F$ is true.

A (partial) play is just an assignment of truth values to the variables in $\vec{A}_1 \cup \vec{A}_2 \cup \cdots \vec{A}_i$, for some $i$, $0 \leq i \leq m$. It is an $\exists$-play when $i$ is even (it is the existential player’s turn to play), and a $\forall$-play otherwise.

A strategy $\sigma$ for the $\exists$ player (given $F$ as above) is a function that maps $\exists$-plays (with $i$ even, $i < m$) to a table of truth values for the variables in $\vec{A}_{i+1}$. We play the QBF game, following $\sigma$, as follows: $\exists$ computes the values to $\vec{A}_1$ by calling $\sigma$ on the empty $\exists$-play; then $\forall$ proposes values for $\vec{A}_2$; $\exists$ computes values to $\vec{A}_3$ by calling $\sigma$ on the length 2 $\exists$-play thus obtained; and so on. Finally, at round $m+1$, $\exists$ wins if and only if $G$ is true under the obtained truth value assignment.
Note that the given QBF formula $F$ is true if and only if the $\exists$ player has a winning strategy $\sigma$ in this game, namely: when $\exists$ plays following $\sigma$, (s)he will win whatever the values proposed by $\forall$ along the game.

16. Assume that QBF is Turing reducible in polynomial time $q(n)$ to some sparse set $S$ (with $p(n)$ elements of size $n$). Show that QBF is then in $\Sigma^p_2$.

Let me just give you a hint to get you started. We would like to guess a strategy $\sigma$ existentially, and play the QBF games following $\sigma$, but any reasonable description of $\sigma$ will take exponential space. Can we use the sparsity of $S$ to solve that issue? (37 lines)

The polynomial time reduction to $S$ will make a polynomial number of queries to the oracle, and since it works in polynomial time (say, $q(n)$), the size of these queries is bounded by $q(n)$. As in Question 11, this implies that the number of query strings that will happen to fall inside $S$ is bounded by a polynomial in $n$, of the order of $q(n)p(q(n))$.

They are also of polynomial size. Define the list $L_n$ of those query strings of size at most $q(n)$ that are in $S$. We have just seen this is a polynomial size string.

We cannot compute it but we can guess it.

Here is a first idea. It won’t work, but it’s a good start, in the sense that $L_n$ should somehow be our short encoding of a winning strategy. On input $F$ of size $n$, we guess a list $L$ of the same size as $L_n$, then we run the polynomial time Turing reduction, replacing the calls to the oracle by a procedure that looks whether the query string is in the list $L$. This is an NP procedure, which is even better than $\Sigma^p_2$, but it does not solve our problem, because there might be other lists $L$ than $L_n$ that will make this procedure accept.

Write $\text{Check}(F, L)$ for that procedure (after guessing $L$). Instead, we use that to define a winning strategy as follows. We reuse the notations $F, G, \overline{A}_i, m, \ldots$. At round $i$ ($i$ even, $i < m$), let $F_i$ be the formula that remains to be evaluated: $\exists \overline{A}_i \forall \overline{A}_{i+1} \cdots Q_m \overline{A}_m G$, with $\overline{A}_1, \overline{A}_2, \ldots, \overline{A}_{i-1}$ replaced by their values as given in the current $\exists$-play. Let $\overline{A}_i$ consist, say, of variable $A_{i1}, \ldots, A_{ip}$. We use a local form of self-reducibility to obtain values for those variables, using $\text{Check}(\_, L)$: if $\text{Check}(F_i[A_{i1} := \text{true}], L)$ accepts then set $A_{i1}$ to true else set it to false; let $F_{i1}$ be the formula $F_i$ with $A_{i1}$ replaced by the picked value; if $\text{Check}(F_{i1}[A_{i2} := \text{true}], L)$ accepts then set $A_{i2}$ to true else set it to false; etc. Note that, given $L$, this strategy is computable in polynomial time, call it $\sigma_L$.

We now play the game, given $F$, following $\sigma_L$. More precisely: given $F$, we guess $L$, then play the game following $\sigma_L$, simulating the $\forall$-player’s moves by guessing them universally. If that actually decides whether $F$ is true, then this will be a $\Sigma^p_2$ algorithm to solve QBF.

If $F$ is true, then we can guess $L$ as $L_n$. In that case, $F = F_0$ is true, $\sigma_L$ will give us values for $\overline{A}_1$ that make the rest of the formula $\forall \overline{A}_2 \exists \overline{A}_3 \cdots Q_m \overline{A}_m G$ with $\overline{A}_1$
replaced by the given values) true. We guess \( \vec{A}_2 \) universally, so the remainder of the formula \( F_2 \) is true again. Then \( \sigma_L \) will give us values for \( \vec{A}_3 \), and we continue until we reach \( i = m \): evaluating \( G \) (with all variables replaced by the values found) will necessarily yield true.

If \( F \) is false, then whichever \( L \) we guess yields a strategy that has no chance of winning, simply because the \( \exists \) player has no winning strategy in this case.

That question was harder than most others, and it was more open in that I gave you more freedom in defining the tools and notions you needed. As a result, I've got many answers that were far from being clear. That came into consideration in the grades.

One funny mistake I've seen consisted in guessing the language \( S \), and checking it was the right one. This is wrong in several respects. First, how do you guess a language, do you guess a Gödel encoding from it, for example? Instead, what you try to guess is the subset of \( S \) that was queried by the given machine on a given input. This is very different (and is the list \( L \) mentioned above). Second, how do you check that \( L \) is the right list? Well, you cannot, and you don't care. What you do care about is whether the list \( L \) you guessed can be used to form an assignment that can be checked later on (self-reducibility). Any such \( L \) fits.

I've also seen complicated arguments such as: take the negation of the input formula, build new machines that should reject when the given machine accepts and conversely, with some extra checks involving \( L \), . . . that double-negation encoding is not needed, and is in direct contradiction to the clarity requirement.

17. Conclude that, if \( \text{PSPACE} \subseteq \text{P/poly} \), then the polynomial hierarchy collapses at level 2 and even more: \( \text{PSPACE} = \Sigma^p_2 = \Pi^p_2 \). (6 lines)

Under the given assumption, \( \text{QBF} \) is in \( \text{P/poly} \), hence in \( \text{PSPACE} \) by Question 6, hence in \( \Sigma^p_2 \) by Question 16. Since \( \Sigma^p_2 \) is **closed under polynomial time reductions** and \( \text{QBF} \) is \( \text{PSPACE}-complete \), \( \text{PSPACE} \subseteq \Sigma^p_2 \). This implies \( \Pi^p_2 \subseteq \text{PSPACE} \subseteq \Sigma^p_2 \), hence \( \Pi^p_2 = \Sigma^p_2 \) (\( \Sigma^p_2 \subseteq \Pi^p_2 \) is a classical argument: any language \( L \) in \( \Sigma^p_2 \) is the complement of a language in \( \Pi^p_2 \), hence of a language in \( \Sigma^p_2 \) since \( \Pi^p_2 \subseteq \Sigma^p_2 \); so \( L \) is in \( \Pi^p_2 \)).

Some of you confused (Karp-Lipton) reducibility with Turing polynomial time reducibility here.