

3.2 $SPACE(n) \neq NP$

Suppose $SPACE(n) = NP$. By the Space Hierarchy Theorem, we know that there is a language $L \in SPACE(n^2) - SPACE(n)$; let M be a TM recognizing L in quadratic space. Define language $L' = \{(x, 1^{|x|^2}) \mid x \in L\}$. Notice that $L' \in SPACE(n)$ since we can construct linear-space TM M' which recognizes L' by taking input $(x, 1^{|x|^2})$ and simulating M on x in $|x|^2$ space (which is linear w.r.t. input $(x, 1^{|x|^2})$). Since $SPACE(n) = NP$, $L' \in NP$. Next, notice that $L \leq_p L'$ trivially since we can simply change an instance x of L to an instance $(x, 1^{|x|^2})$ of L' . Since $L' \in NP$ and NP is closed under poly-time reductions, $L \in NP$. But this means $L \in SPACE(n)$, which contradicts our original definition of L .

3.3 $\exists B \in EXP$ such that $P^B \neq NP^B$

We use essentially the same definition of B as in the textbook's proof of Baker-Gill-Solovay. The only difference is that we need to remove some ambiguities/non-determinism from the definition, or else we could end up with an undecidable B . Here is the book's description of stage i construction; I added disambiguations in italics (I also changed n to m to avoid confusion later):

Stage i : So far, we have declared whether or not a finite number of strings are in B . Choose m to be the smallest integer that exceeds the length of any such string, choose all undetermined strings of length less than m to not be in B , and run M_i on input 1^m for $2^m/10$ steps. Whenever M_i queries the oracle about strings whose status is undetermined, we declare that the string is not in B . After letting M_i finish computing on 1^m , we now wish to ensure that the answer of M_i on 1^m (whatever it was) is incorrect. So if M_i accepts 1^m , we declare that all remaining strings of length m are also not in B , thus ensuring $1^m \notin U_B$. Conversely, if M_i rejects 1^m , we pick the lexicographically smallest string x of length m that M_i has not queried (such a string exists because M_i made at most $2^m/10$ queries) and declare that x is in B , thus ensuring $1^m \in U_B$. In either case, the answer of M_i is incorrect, so we have shown U_B is not in P^B .

Now we just need to show that this B can indeed be recognized in exponential time. Pick a string x of length n . We have to be a little careful because technically this construction of B allows m to be as large as 2^i , so running for $2^m/10 = 2^{2^i}/10$ is too many steps if we run our simulation for n stages. However, in the case that $m = 2^i$, i ends up being on the order of $\log n$, so we actually only need to run the simulation for $\log n$ stages. The

way to see this formally is that we only need to simulate the construction until we reach a stage where $m > n$. Once this happens, we know we already picked a determination for all strings of length less than m , which includes x . Thus, at each stage i , m is at most n , so we simulate at most $2^n/10$ steps of M_i .

Now we need to work out how long it can take for our simulation to compute the answer to an oracle query. This is actually quite tricky. If we take the naive approach and say that we simply record whether each string is in B or not as we come across them, we run into a problem; on the final stage, if $m = n$, then M_i can ask for queries of length up to $\Theta(2^n)$. This means we need to keep track of all strings up to length $\Theta(2^n)$ - but that's on the order of 2^{2^n} strings, which is far too many to store in singly exponential time! Thus we need to be a little more clever and notice that each stage of our simulation places at most one string in B . Since there can be at most n stages, we see that there can be at most n strings in B . So we can keep track of only those strings that are in B . Of course, we still need to be able to distinguish between strings not in B and strings that have yet to be determined, but this is possible due to the fact that at stage i , all strings of length less than m have been determined to be either in B or not, and M_i can make at most 2^m queries on strings of length at most 2^m , which can be stored in time $O(2^{2^m}) = O(2^{2^n})$ (we only need to store these queries until the end of this stage, then we can forget about them).

Finally, we compute the overall runtime of our simulation. There are at most n stages. Each stage simulates at most $2^n/10$ steps of M_i and then we may need to pick a string to put in B , which requires looking at the list of strings that M_i queried, i.e. $O(2^{2^n})$ time. Furthermore, each step of M_i could be an oracle query of a string of length up to 2^n , so that could require looking at our list of strings queried this stage and our list of all strings in B so far, which is $O(2^{2^n} + n2^n)$. Thus the total running time is $O((2^n/10)(2^{2^n} + n2^n) + 2^{2^n}) = O(2^{3n})$, which is *EXP* time.

3.6 Ladner's Theorem

a.) To compute $H(n)$, we simply iterate from $i = 1$ to $\log \log n$ and check that $M_i(x) = SAT_H(x)$ for all x with $|x| \leq \log n$ when we run M_i for $i|x|^i$ steps. If $A(n)$ is the time to compute $M_i(x)$ and $B(n)$ is the time to compute $SAT_H(x)$, then the overall running time of this algorithm is $T(n) = O(n(\log \log n)(A(n) + B(n)))$. Now $A(n) = O((\log \log n)(\log n)^{\log \log n})$ and $B(n) = (\log n)T(\log n)$ since it involves computing $H(\log |x|)$ and brute-forcing *SAT* formulas of size $O(\log n)$. Hence we have $T(n) = O(n(\log \log n)((\log \log n)(\log n)^{\log \log n} + (\log n)T(\log n)))$. Now no-

tice that $f(n)^2 = o(2^{f(n)}) \Rightarrow (\log \log n)^2 = o(\log n) \Rightarrow (\log n)^{\log \log n} = 2^{(\log \log n)^2} = o(n)$. Thus $T(n) = O(n^2 \log n + n(\log \log n)(\log n)T(\log n))$. We easily verify that $T(n) = O(n^3)$: $O(n^2 \log n + n(\log \log n)(\log n)T(\log n)) = O(n^2 \log n + n(\log \log n)(\log n)^4) = O(n^2 \log n + n(\log n)^5) = O(n^3)$.

b.) Let f be a polynomial time reduction from SAT to SAT_H . Since f is poly-time, it can only use poly space, so pick c such that $|f(x)| = |x|^c$. Now since $\lim_{n \rightarrow \infty} H(n) = \infty$, pick N such that for all $n \geq N$, $H(n) > 2c$. Consider the following algorithm for solving SAT on input ϕ :

1. If $|\phi| \leq 2N^2$, then brute force all possible assignments to check for a satisfying one, and accept or reject appropriately
2. Otherwise, compute $f(\phi)$ – if it's not of the form $\psi 01^{|\psi|^{H(|\psi|)}}$, reject
3. Recurse on ψ

Notice that each recursive call of this algorithm runs in polynomial time since we can compute H in polynomial time. We will now show that at any step of the recursion, $|\psi| \leq \sqrt{|\phi|}$. Let $n = |\phi|$ and $m = |\psi|$. Then $n^c = |\phi|^c = |f(\phi)| = |\psi 01^{|\psi|^{H(|\psi|)}}| = m + 1 + m^{H(m)} \leq 2m^{H(m)}$. Thus we have $m^{H(m)} \leq n^c \leq 2m^{H(m)}$. Now, note that since we're doing the recursive call for this step, we didn't do the base case, so $|\phi| > 2N^2$, i.e. $N < \sqrt{n/2}$. Since H is monotone, this implies that $H(\sqrt{n/2}) > H(N)$. But this means that $H(m) \leq 2c$ is impossible, since that would give us $n^c \leq 2m^{H(m)} \Rightarrow n^c \leq 2m^{2c} \Rightarrow m \geq \sqrt{n/2} \Rightarrow H(m) \geq H(\sqrt{n/2}) > H(N) > 2c \geq H(m)$. Hence $H(m) > 2c$, and so we have $n^c \geq m^{H(m)} \geq m^{2c} \Rightarrow m \leq \sqrt{n} \Rightarrow |\psi| \leq \sqrt{|\phi|}$. Now we see that the algorithm above reduces its problem size by a square root with each recursive call, so there can be at most $\log n$ recursive calls, each of which executes in polynomial time, and so we have a polynomial time algorithm for SAT .

4.4 Strongly Connected Digraph

Let $SC = \{G | G \text{ is a strongly connected digraph}\}$. Clearly, $SC \in NL$ since $PATH \in NL$ and to determine membership in SC we need to check that there is a path from each vertex to every other vertex in G . This only requires logspace as test for membership in $PATH$ only requires logspace and we can store the pair of vertices (represented as integers $\leq n$) for which we are currently trying to find a path in logspace.

Now we show that SC is NL -hard by giving a logspace reduction from $PATH$ to SC . Given an input to the path problem, (G, s, t) , construct a

graph G' by duplicating G and adding a directed edge from every vertex in $V - \{s, t\}$ to s and an edge from t to every vertex in $V - \{s, t\}$. Also add an edge from t to s . If the input to $PATH$ contains n nodes then the input to SC contains n nodes and can be represented as an n by n adjacency matrix.

This is a logspace reduction since we can represent the graphs in question as adjacency matrices. Then $A_{G'}(i, j) = 1 \iff A_G(i, j) = 1 \vee i = t \vee j = s$. All of these can be verified in logspace by scanning the input to the $PATH$ problem and keeping track of the row and column integers. Thus we can compute each bit of $A_{G'}$ in logspace.

$(G, s, t) \in PATH \iff G' \in SC$

(\Rightarrow) Suppose $(G, s, t) \in PATH$. Pick vertices $u, v \notin \{s, t\}, u \neq v$ of G' . Then we have a path from u to v by taking the edge from u to s , the path from s to t that was in G , and finally the edge from t to v . Note that a piece of this same path will still work if $u \in \{s, t\}$ or $v \in \{s, t\}$, except for when $u = t$ and $v = s$, in which case we take the edge from t to s that we added.

(\Leftarrow) We prove the contrapositive; suppose $(G, s, t) \notin PATH$, and suppose by way of contradiction that $G' \in SC$. Then there's a path from s to t in G' that doesn't exist in G - i.e. if we label the shortest path from s to t in G' as $(s = v_0, v_1, v_2, \dots, v_k = t)$, then there's an edge (v_i, v_{i+1}) ($0 \leq i \leq k - 1$) that's in G' but not in G . This means it was an added edge, so there are only two cases:

- Case 1: $v_i = t$ - Our path looks like $(s = v_0, \dots, v_i = t, v_{i+1}, \dots, v_k = t)$. But this is impossible since (v_0, \dots, v_i) is a shorter path from s to t .
- Case 2: $v_{i+1} = s$ - Similarly, we have that (v_{i+1}, \dots, v_k) is a shorter path from s to t , which is a contradiction.

Since we get a contradiction in every possible situation, we see that $G' \notin SC$.

4.6 Logspace Reduction to SAT

First, we will verify that the ϕ_x produced in the book by the Cook-Levin reduction can be implicitly computed in logspace. By Problem 4.8, we can do this by giving a logspace algorithm to output ϕ_x in a "write-once" fashion, which is perhaps a little easier to think about. Consider the four parts of ϕ_x that the Cook-Levin reduction produces. For the first part, we simply write the string equality formula directly from left to right. We also maintain a counter (requiring $\log n$ space) to keep track of where we are. The second and fourth parts are both easy since they're constant formulas requiring no

extra space. The third part is written from left to right as a conjunction of $T(n)$ formulas; this requires a counter of size $\log(T(n))$, which is $O(\log n)$ since $T(n)$ is polynomial in n . For each of the $T(n)$ formulas, we need to compute $\text{prev}(i)$ and $\text{inputpos}(i)$ and then use the results to write a boolean function formula as in Claim 2.13. The latter part is easy since the function formula only requires counters of constant size (the constant being related to the size of a snapshot). For the computation of $\text{prev}(i)$ and $\text{inputpos}(i)$, we simulate M on input $(x, 0^{p(|x|)})$. Notice, however, that for the normal Cook-Levin Theorem, we precompute $\text{prev}(i)$ and $\text{inputpos}(i)$ for all i in one simulation, and then look up values as needed. For our logspace version, we don't have the space to do this, so we instead must run a new simulation every time we come to $\text{prev}(i)$ or $\text{inputpos}(i)$ in our formula.

Next we need to show that $SAT \leq_l 3SAT$. We use the same reduction as normal, being careful about space usage. We keep track of a global variable i which is a counter telling us the last-used index for a new "z" variable. We initialize i to 0 and do the following:

on input clauses $C_1 \wedge \dots \wedge C_m$, with $C_1 = u_1 \vee \dots \vee u_n$:

1. if $m = 0$, halt
2. if $n \leq 3$:
3. output " $C_1 \wedge$ "
4. recurse on $C_2 \wedge \dots \wedge C_m$
5. else:
6. increment i
7. output " $(u_1 \vee u_2 \vee z_i) \wedge$ "
8. recurse on $(\bar{z}_i \vee u_3 \vee \dots \vee u_n) \wedge C_2 \wedge \dots \wedge C_m$

Note that we technically can't do the recursion at step 8 because $(\bar{z}_i \vee u_3 \vee \dots \vee u_n) \wedge C_2 \wedge \dots \wedge C_m$ isn't part of the input, and making a copy of the input takes more than log space. This is easy to get around, however, by keeping a variable z to which we store \bar{z}_i every time we recurse at step 8. Then steps 3 and 7 output this z as necessary. z gets overwritten at each recursive call, so this only uses log space.

Now we have $SAT \in L \Rightarrow$ everything in NP is logspace reducible to something in $L \Rightarrow$ everything in NP is in $L \Rightarrow NP = L$. Also, $NP = L \Rightarrow SAT \in L$ since $SAT \in NP$. Hence $SAT \in L \iff NP = L$.

4.8 Write-Once TMs

(\Rightarrow) Suppose f is write-once logspace computable. Let M be a write-once TM computing f in logspace. For any input (x, i) , we can compute bit i of $f(x)$ in logspace by simply simulating M without its output tape and keeping a count of how many times M has written to its write-once output tape. After M writes for the i^{th} time, we output the bit it just wrote. We can also check whether $i \leq |f(x)|$ by simply checking whether M halts before our counter reaches i . Hence f is implicitly logspace computable.

(\Leftarrow) Suppose f is implicitly logspace computable. Let M be a logspace TM recognizing $\{(x, i) \mid f(x)_i = 1\}$ and M' be a logspace TM recognizing $\{(x, i) \mid i \leq |f(x)|\}$. Given an input x , we can output $f(x)$ in a write-once fashion by simply running M on $(x, 1)$ and outputting the result, and then running M on $(x, 2)$ and outputting the result, etc. We continue doing this until we reach (x, i) , where $i \not\leq |f(x)|$ (which is checked each time using M'), at which point we halt with precisely $f(x)$ on the output tape. Note that this requires a counter for the current bit – this counter is guaranteed to take only log space since $|f(x)|$ is polynomial in $|x|$.

4.10 Game Theory

Represent the game as an instance of QBF with n variables. As described in Example 4.15 in the textbook, player 1 has a winning strategy iff $Q = \exists x_1 \forall x_2 \exists x_3 \dots (\exists/\forall) x_n \phi(x_1, \dots, x_n)$ is true, for some ϕ which describes what it means for player 1 to win. We will show that if player 1 doesn't have a winning strategy, then player 2 must have one. Suppose player 1 doesn't have a winning strategy. Then Q is false, and so $\neg Q$ is true. $\neg Q = \forall x_1 \exists x_2 \dots (\forall/\exists) x_n \neg \phi(x_1, \dots, x_n)$. Since the existential and universal quantifiers have swapped positions, this formula corresponds to a strategy for player 2. Also, notice that $\phi(x_1, \dots, x_n)$ in Q encoded whether player 1 wins; thus $\neg \phi(x_1, \dots, x_n)$ encodes player 1 not winning, which is the same as player 2 winning since there are no ties and the game is finite. Thus $\neg Q$ precisely encodes whether player 2 has a winning strategy, and since $\neg Q$ is true, player 2 does indeed have one.