Problem 1: BPP ⊆ PSPACE

Let $L \in \text{BPP}$. Show that $L \in \text{PSPACE}$ by giving an explicit deterministic polynomial-space algorithm that decides $L$.

**Hint:** The construction is slightly easier if you use the definition of BPP that I called the “alternative definition” in the lecture.

**Solution.** The “alternative definition” was the following:

**Definition 1 (BPP)** A language $L$ is in BPP iff there exists a polynomial-time Turing machine and a polynomial $p$ such that for all $x \in \{0,1\}^*$,

- If $x \in L$ then $\Pr[M(\langle x, r \rangle) = 1 : r \leftarrow \{0,1\}^{p(|x|)}] \geq \frac{2}{3}$
- If $x \notin L$ then $\Pr[M(\langle x, r \rangle) = 1 : r \leftarrow \{0,1\}^{p(|x|)}] \leq \frac{1}{3}$

To show that $L \in \text{PSPACE}$, we need to give a polynomial-space algorithm that decides $L$. Let $M$ be the Turing machine from **Definition 1**

<table>
<thead>
<tr>
<th>Input:</th>
<th>$x$ (the $L$-instance to be decided).</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n := 0$</td>
<td></td>
</tr>
<tr>
<td><strong>for every</strong> $r \in {0,1}^{p</td>
<td>x</td>
</tr>
<tr>
<td>Run $x \leftarrow M(\langle x, r \rangle)$</td>
<td></td>
</tr>
<tr>
<td><strong>if</strong> $x = 1$ <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$n := n + 1$</td>
</tr>
<tr>
<td>$p := n \cdot 2^{-p(</td>
<td>x</td>
</tr>
<tr>
<td><strong>if</strong> $p &gt; \frac{1}{2}$ <strong>then</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>return</strong> 1</td>
</tr>
<tr>
<td><strong>else</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>return</strong> 0</td>
</tr>
</tbody>
</table>

In this algorithm, $n$ is the number of randomnesses $r$ for which $M(\langle x, r \rangle) = 1$. Thus $p$ is the probability that $M(\langle x, r \rangle) = 1$ (for $r \leftarrow \{0,1\}^{p(|x|)}$). Hence the algorithm returns 1 iff $\Pr[M(\langle x, r \rangle) = 1 : r \leftarrow \{0,1\}^{p(|x|)}] > \frac{1}{2}$, and by **Definition 1** this implies that the algorithm returns 1 iff $x \in L$. Thus the algorithm decides $L$.

Furthermore, the algorithm runs in polynomial-space: It needs $O(p(|x|))$ bits for maintaining $r, n, p$. And since $M$ is polynomial-time, the space needed for computing $M(\langle x, r \rangle)$ is bounded by $q(|x| + |r|)$ for some polynomial $q$. Since $|r| \leq p(|x|)$, $q(|x| + |r|)$ is polynomially bounded in $|x|$ as well. Thus the algorithm runs in polynomial-space.
Problem 2: Complexity of two-player games

(a) Consider the game Hex on an \(n \times n\) board (see [Wikb] for the rules). Describe an algorithm \(\text{wins}(p, b)\) (in pseudo-code) that returns whether there exists a strategy for player \(p\) (\(p \in \{\text{red}, \text{blue}\}\)) such that \(p\) will win (no matter how the other player plays) when starting from board configuration \(b\).

Estimate the space needed for running your algorithm. (Estimate means that you can ignore constant factors.)

Your algorithm should run in polynomial-space.

Solution. We use the following notation: Let \(\text{fields}\) be the set of the fields of the board (i.e., pairs of numbers \(1, \ldots, n\) since there are \(n \cdot n\) fields where a player can put a stone). Let \(\text{isFree}(b, f) = 1\) if field \(f\) is free on board \(b\). For a board position \(b\) and a field \(f \in \text{fields}\) and a player \(p\), let \(\text{move}(b, p, m)\) be the board after putting a player-\(p\) stone on field \(f\) on board \(b\). Let \(\text{isEnd}(b)\) returns 1 if the board \(b\) is full (no more moves). \(\text{isWin}(b, p)\) returns 1 if the board \(b\) is full and player \(p\) has won. (Note: if \(\text{isEnd}(b) = 1\) and \(\text{isWin}(b, p) = 0\), then player \(p\) has lost.) If \(p\) is a player, then \(\text{other}(p)\) is the other player.

We define the following recursive algorithm \(\text{canWin}(b, p)\) that returns whether \(p\) has a winning strategy when starting with board \(b\) (when \(p\) is the player who moves next).

To estimate the memory needed for this algorithm, we first estimate the size of all variables: Since a board is an \(n \times n\) array of fields, each being either occupied by one of the two players, or empty, we can store a board position in \(O(n^2)\) space. Thus the variable \(b\) needs \(O(n^2)\) space. The variable \(p\) needs \(O(1)\), and the variable \(f\) needs \(O(\log n)\). Thus we need \(O(n^2)\) space on the stack for each recursion level.

Since in each recursion level, there is one more stone on the board, and the board can hold at most \(n^2\) stones, there will be at most \(n^2\) recursion levels. Thus the total amount of memory is \(n^2 \cdot O(n^2) = O(n^4)\).
(b) Chess (in a generalized form where the board has size \( n \times n \) and the number of pieces is accordingly increases, except for the number of kings) is shown to be \( \text{EXP}-\text{hard} \) [FL81]. In particular, assuming \( \text{PSPACE} \neq \text{EXP} \), this means that there is no polynomial-space algorithm that decides whether a given chess position has a winning strategy for a given player.

Why is it not possible to use the algorithm from (a) to decide whether a chess position has a winning strategy for a given player? (With the algorithm suitably modified to use chess rules instead of Hex rules, of course.) More precisely, why is the algorithm not polynomial-space in that case?

Chess is often played with the “fifty-move rule” [Wika]. This rule says that, if for 50 moves in a row, no piece has been captured, and no pawn has been moved, the game is a draw. Show that with this rule, chess is in \( \text{PSPACE} \). (More specifically, show that the algorithm from (a), modified to use chess rules, is terminating and polynomial-space.)

Solution. When estimating the size used by the algorithm from (a), we used the fact that at most \( n^2 \) moves can be performed during one game of Hex. In chess, there is no such upper bound, a chess game can, in principle, be arbitrary long. (Game positions might repeat. Even if we disallow repeating game positions, there can be exponentially many different positions in a single game.) Thus the algorithm will run into infinite recursion (or, if we check for repeated positions, at least need an exponentially large stack).

With the fifty-move rule, we have that every fiftyth move will either move a pawn, or capture a piece. Since on an \( n \times n \) board, there can be at most \( n^2 \) pawns, and each pawn can move at most \( n \) steps, and at most \( n^2 \) pieces can be captured, we have that the total length of the game can be at most \( 50n^2 \cdot n + 50n^2 = O(n^3) \).

Thus, if we use the algorithm from (a) (suitably modified for chess), we have a recursion depth of \( O(n^3) \) (in particular, the algorithm terminates). In each recursion level we need to store a chess board which needs \( O(n^2) \), thus the total space-complexity of the algorithm is \( O(n^5) \).

(c) (Bonus problem) In the game Nim, we have a number of heaps of objects on the table. Players move alternatingly. When it is a player’s turn, he can remove an arbitrary (nonzero) number of objects from a single heap. Who removes the last object wins.

Show that Nim is in \( \text{P} \).

Hint: If \( x_1, \ldots, x_n \) are the sizes of the heaps, show the following: If \( x_1 \oplus \cdots \oplus x_n = 0 \) before a move, necessarily \( x_1 \oplus \cdots \oplus x_n = 0 \) after the move. And if \( x_1 \oplus \cdots \oplus x_n \neq 0 \), then there is a move that leads to \( x_1 \oplus \cdots \oplus x_n = 0 \). And note that the empty board (which is what you need to achieve to win) has \( x_1 \oplus \cdots \oplus x_n = 0 \). Use these facts to design a simple winning strategy.
Solution. If $x_1, \ldots, x_n$ are the sizes of the heaps before a move, let $x'_1, \ldots, x'_n$ denote the sizes after the move.

If $x_1 \oplus \cdots \oplus x_n = 0$, and the current player removes something from heap $i$, then $x_i \neq x'_i$ and $x_j = x'_j$ for $j \neq i$. Thus

$$(x'_1 \oplus \cdots \oplus x'_n) = (x_1 \oplus \cdots \oplus x_n) \oplus (x'_1 \oplus \cdots \oplus x'_n) = x_i \oplus x'_i \neq 0.$$  

Consider the case that $p := x_1 \oplus \cdots \oplus x_n \neq 0$. Let $k$ be the position of the most significant non-zero bit of $p$ (i.e., $p \in [2^k, \ldots, 2^{k+1} - 1]$). Pick $i$ such that the $k$-th bit of $x_i$ is 1. (Such $i$ must exist, since otherwise the $k$-th bit of all $x_j$ would be zero, and thus the $k$-th bit of $p$ would be zero.) We have that $x_i \oplus p < x_i$. (Since changing $x_i$ to $x_i \oplus p$ will set the $k$-th bit from 1 to 0, and change only less significant bits from 0 to 1.) Thus $d \geq 1$ and $d \leq x_i$ for $d := x_i - (x_i \oplus p) > 0$. We let the current player remove $d$ objects from heap $i$. Then $x'_i = x_i - d = x_i \oplus p$, and $x'_j = x_j$ for $j \neq i$. Thus

$$x'_1 \oplus \cdots \oplus x'_n = (x_1 \oplus \cdots \oplus x_n) \oplus x_i \oplus x'_i = p \oplus x_i \oplus (x_i \oplus p) = 0.$$  

Thus there is a move so that $x'_1 \oplus \cdots \oplus x'_n$ is zero afterwards.

This gives the following optimal strategy:

- If $x_1 \oplus \cdots \oplus x_n = 0$, make an arbitrary move.
- If $x_1 \oplus \cdots \oplus x_n \neq 0$, make a move such that $x'_1 \oplus \cdots \oplus x'_n = 0$.

If the starting position for player 1 is $x_1 \oplus \cdots \oplus x_n \neq 0$, this strategy will ensure that player 2 always has a board with $x_1 \oplus \cdots \oplus x_n = 0$, and player 1 always a board with $x_1 \oplus \cdots \oplus x_n \neq 0$. This implies that the player who will end up with the empty board is player 2. Thus player 2 losses. (The same holds with players 1 and 2 exchanged.)

If the starting position for player 1 has $x_1 \oplus \cdots \oplus x_n = 0$, then player 1 cannot win (unless player 2 makes a mistake) since player 1 will necessarily move to a board position where $x_1 \oplus \cdots \oplus x_n \neq 0$ for player 2 (who then can win using the above strategy). Thus it does not matter what the strategy does in this case. (The same holds with players 1 and 2 exchanged.)

This shows that the above is an optimal strategy.

To decide whether the current player has a winning strategy, one needs to check whether $x_1 \oplus \cdots \oplus x_n \neq 0$. This is in P.

References
