Generator zur Erzeugung kartisischer Produkte.

```
def generate_product_lex(maxima):
    """Generator, der einen Iterator für das Kartesische Produkt
    in _lexikographischer_ Ordnung liefert: Das Argument dieser Funktion
    ist eine Liste von natürlichen Zahlen m_1, m_2, ... m_n.
    Wir erzeugen also alle Elemente des Produkts [m_1]x[m_2]x...x[m_n],
    aber nicht "alle auf einmal, in einer langen Liste", sondern "on demand",
    also immer nur ein einziges Tupel, wenn es angefordert wird.

    Dabei achten wir auch auf zwei "Grenzfälle"."""
    # Grenzfall 1:
    # Wenn das kartesische Produkt _leer_ ist, also keinen einzigen Faktor
    # (anders gesagt: "0 Faktoren") enthält, dann ist das (einzige!) Ergebnis
    # das leere Tupel (mit 0 Komponenten)
    if not maxima:
        # Mathematisch: Leeres Tupel, hier als Python-Datentyp list.
        return []

    # Implicit else: Wenn wir nicht bereits aus der Funktion mit
    # "return" aus der vorigen if-Abfrage "herausgesprungen" sind.

    # Grenzfall 2:
    # Wenn ein Faktor des Produkts die leere Menge ist,
    # ist auch das ganze kartesische Produkt leer!!!
    if 0 in maxima:
        # Dieser "normale" return-Befehl bedeutet: Der vom "Generator"
        # erzeugte "Iterator" endet hier - er liefert also kein einziges
        # Tupel.
        return

    # Implicit else: Wenn wir nicht bereits aus der Funktion mit
    # "return" aus den vorigen if-Abfragen "herausgesprungen" sind.

    # Normalfall:
    dim = len(maxima)
    # (1,1,...1)  ist das erste Tupel (in Python: Datentyp Liste,
    # denn ein "tuple" in Python ist "immutable", d.h., wir können seine
    # Komponenten später nicht mehr verändern) in der lexikographischen
    # Ordnung:
    current_tuple = [1]*dim

    # Mit dem Walross-Operator wird die Sache einfach:
    while (last_pos := find_last_pos(current_tuple, maxima)) > -1:
        # Der Befehl yield macht aus der Funktion einen "Generator".
        # Man muß sich das so vorstellen: Im "Iterator", der von dem
        # "Generator" erzeugt wird, funktioniert "yield" wie "return";
        # nur wird nach dem "yield" die Funktion nicht beendet, sondern
        # der "Iterator" setzt bei der nächsten "Aufforderung, einen
        # Wert zu liefern", unmittelbar nach diesem "yield" wieder fort.
        # (Die "Aufforderung, einen Wert zu liefern" erfolgt in der
        # typischen Anwendung einer for-Schleife _unsichtbar_ für die
        # Programmiererin.)
        yield current_tuple

        current_tuple[last_pos] += 1
        current_tuple[last_pos+1:] = [1]*(dim-last_pos-1)

        last_pos = find_last_pos(current_tuple, maxima)

# Abschließend nochmals "yield", um auch die letzte Liste auszugeben:
yield current_tuple
# Hier endet der vom "Generator" erzeugte "Iterator". (Den "return"
# Befehl könnten wir auch weglassen, weil er "implizit" am Ende der
# Funktionsdefinition (englisch: function body) ergänzt wird.)
# In der typischen Anwendung für eine for-Schleife wird die Schleife
# nun beendet.
return
```