Задача: *Быстрое перемножение матриц*

Для систем автоматической проверки

Краткое описание

Необходимо реализовать умножение двух квадратных целочисленных матриц. Наивная реализация с тройным циклом имеет асимптотику $O(n^3)$ и слишком медленна для больших размерностей n на Python.

Ограничения в этой задаче подобраны так, что простая реализация $O(n^3)$ на Python не успеет пройти все тесты. Для успешного решения задачи участнику предстоит реализовать один из быстрых алгоритмов:

- алгоритм Штрассена;
- или его модификацию Штрассена–Винограда.

Использование любых сторонних библиотек (в том числе `numpy`, `scipy` и т. п.) запрещено. Разрешена только стандартная библиотека Python и собственная реализация алгоритма умножения.

1 Условие задачи (для участника)

Даны две квадратные матрицы A и B размера $n \times n$, с целыми элементами. Требуется вычислить их произведение

$$C = A \cdot B$$

и вывести матрицу C.

Формат входных данных

Первая строка содержит одно целое число n — размерность матриц ($1 \le n \le N_{\max}$).

Далее следуют n строк, в каждой из которых записано n целых чисел a_{ij} — элементы матрицы A.

После этого следуют ещё n строк, в каждой из которых записано n целых чисел b_{ij} — элементы матрицы B.

Все элементы матриц удовлетворяют условию

$$-9 \le a_{ij}, b_{ij} \le 9.$$

Формат выходных данных

Выведите n строк по n целых чисел — элементы матрицы

$$C = A \cdot B.$$

Числа в строках можно разделять пробелами или переводами строк. Дополнительные пробелы в конце строк допускаются.

Ограничения и требования к реализации

- Язык решения: **Python 3**.
- Запрещено использование любых внешних библиотек и модулей: нельзя делать `import numpy`, `import scipy`, `import numba`, `import sympy` и т. п. Решения с такими импортами будут отклоняться.
- Запрещено использовать любые готовые функции/методы для перемножения матриц из внешних библиотек. Допускается только собственная реализация алгоритма.
- Разрешено использовать:
  - стандартный ввод/вывод;
  - базовые типы данных Python (списки, кортежи и т. п.);
  - простые функции стандартной библиотеки, не связанные с линейной алгеброй (например, `sys.stdin`, `random` при генерации локальных тестов и т. п.).
- Ограничения по времени и памяти перечислены в системе проверки; они подобраны так, что наивный алгоритм $O(n^3)$ на Python не проходит самые тяжёлые тесты.

Для прохождения всех тестов рекомендуется реализовать алгоритм Штрассена или его модификацию Штрассена–Винограда с рекурсивным делением матриц на блоки и переходом на обычный $O(n^3)$ алгоритм при малых размерностях (порог можно выбирать самостоятельно).

Пример:

```
Вход:
2
1 2
3 4
5 6
7 8

Выход:
19 22
43 50
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}.$$

2 Комментарии к реализации (для участника)

- Для корректности алгоритм должен работать при любых $1 \le n \le N_{\max}$, в том числе если n не является степенью двойки. Теоретически простой приём — дополнить обе матрицы нулями до ближайшей сверху степени двойки $m = 2^k \ge n$, выполнить алгоритм для размера m, а затем обрезать результат обратно до $n \times n$. Однако при n, существенно меньших m, такая стратегия может приводить к заметному **замедлению**. В рамках данной задачи более практичным является гибридный подход: рекурсивно применять Штрассена только тогда, когда текущий размер блока чётный и достаточно большой, а для нечётных (и малых) размеров переходить к обычному тройному циклу $O(n^3)$ без дополнения нулями.

- При реализации алгоритма Штрассена/Штрассена–Винограда обычно вводят порог n_0, при котором рекурсия прекращается, а для оставшегося блока используется обычный тройной цикл $O(n^3)$. Оптимальное значение n_0 зависит от среды исполнения и может подбираться экспериментально (например, в районе $32 \dots 128$).

- Внимательно следите за индексацией и разбиением матриц на блоки: все рекурсивные вызовы должны получать квадратные матрицы одинакового размера.

Приложение: Постановка задачи

Этот раздел содержит ориентиры по выбору размерностей, тестов и лимитов задачи.

1. Цель

Сделать так, чтобы:

- корректная реализация алгоритма Штрассена или Штрассена–Винограда на Python 3 проходила все тесты;
- наивный алгоритм $O(n^3)$ в чистом Python не укладывался в лимит времени на больших тестах;
- реализации, которые *наивно дополняют* матрицы до ближайшей степени двойки при "неудобных" размерах n, демонстрировали худшее поведение и отсеивались по времени.

2. Эмпирические замеры (локальные тесты)

Ниже приведены ориентировочные результаты локальных замеров времени (один и тот же компьютер, CPython 3.x, целые элементы в диапазоне $[-9, 9]$, порог отключения $n_0 = 64$, однократный прогон; значения округлены до сотых):

n	Classic $O(n^3)$	Strassen	Strassen–Winograd
256	≈ 1.11 s	≈ 0.93 s	≈ 0.92 s
512	≈ 9.67 s	≈ 6.83 s	≈ 6.61 s

Из этих замеров видно, что:

- при $n = 256$ рекурсивный алгоритм даёт выигрыш по времени порядка 15–20% по сравнению с чистым $O(n^3)$;
- при $n = 512$ выигрыш уже порядка 30%;
- различие между классическим Штрассеном и вариантом Штрассена–Винограда в интерпретируемом Python невелико (преимущество последнего не всегда проявляется из-за накладных расходов интерпретатора).

3. Рекомендуемая структура тестов

Имеет смысл разбить тесты на группы (subtasks) примерно так, чтобы отразить три аспекта задачи:

- **Группа A (корректность, малые n)**: $1 \le n \le 64$ (например, отдельный тест с $n = 64$). На этих тестах проходят любые корректные реализации, в том числе наивный тройной цикл.
- **Группа B («неудобные» и средние n)**: сюда можно включить значения среднего размера и специально подобранные "плохие" размеры, например

  $$n \in \{128, 192, 256, 257, 300, 384\}.$$

  Особенно интересны случаи $n = 257, 300, 384$: на них наивное дополнение до ближайшей степени двойки (например, до 512) приводит к перерасходу операций и может быть заметно медленнее обычного $O(n^3)$. При правильно реализованном гибридном алгоритме (рекурсия по Штрассену только для чётных и больших размеров, без грубого дополнения) решение не должно сильно проигрывать классике даже на таких "неудобных" размерностях.
- **Группа C (основная нагрузка, большие n)**: один или несколько тестов с "сильными" размерностями, например

  $$n = 512$$

  (при желании можно добавить ещё один-два похожих теста). Эти тесты должны давать основную часть баллов и быть подобраны так, чтобы наивный $O(n^3)$ на Python не укладывался в лимит времени, а реализация Штрассена/Штрассена–Винограда с хорошим порогом и аккуратным обращением с нечётными размерностями — проходила.

Важно! Тесты с n, который **близок** к числу, являющемуся степенью двойки, но **чуть больше или меньше** него (например, $n = 257$ или $n = 384$), чувствительны к стратегии: если матрицы наивно дополнять нулями до ближайшей степени двойки, реализация Штрассена может оказаться *медленнее* чистого $O(n^3)$. Включение таких тестов в отдельную группу B делает задачу интереснее и стимулирует участников реализовать гибридный алгоритм, который использует Штрассена только там, где это действительно выгодно.

4. Рекомендации по лимитам времени и памяти

Точные значения зависят от производительности сервера, версии Python и так далее. Общая стратегия:

Время:

- оценить время работы наивного $O(n^3)$ и Страссена для $n = 384$ и $n = 512$ напрямую на сервере;
- выбрать лимит времени T для группы C так, чтобы:
  - реализация Страссена/Штрассена–Винограда укладывалась примерно в $(0.6 \dots 0.8)\,T$;
  - наивный тройной цикл превышал T примерно в 1.2–1.4 раза.
- для группы B можно поставить более мягкие лимиты, чтобы и классика, и гибридные реализации успевали, но при этом различие во времени было заметно (для анализа).

Память:

- Рекурсивные реализации Страссена заметно расходуют память под временные матрицы. Для Python имеет смысл ставить лимит повыше (не менее 512MB или 1GB).

5. Запрет библиотек

Для гарантированного запрета сторонних библиотек можно проверять исходный код на наличие запрещённых паттернов (например, `import numpy`, `import scipy`, `import numba` и т. п.) и отклонять такие решения на этапе компиляции. Также можно просто запретить любой из импортов, так как для решения задачи они не обязательны, гибридный пример решения задачи будет ниже.

Пример решения — *Алгоритм Штрассена*

Ниже приведён пример решения на Python 3, реализующий *гибридный* алгоритм Штрассена для умножения квадратных целочисленных матриц. При размерности блока $n \leq n_0$ используется обычный тройной цикл $O(n^3)$, где $n_0 = 64$. При чётном и достаточно большом n матрица рекурсивно делится на четыре блока и обрабатывается по формулам Штрассена. Если на каком-либо уровне размер становится нечётным, алгоритм переключается на обычное умножение *без* дополнения до степени двойки — это позволяет избегать резкого замедления на "неудобных" размерностях n.

```
def read_matrix(n):
    # Читаем квадратную матрицу n×n из стандартного ввода
    return [list(map(int, input().split())) for _ in range(n)]


def matmul_classic(A, B):
    # Классическое умножение матриц O(n^3) с тройным циклом
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        rowA = A[i]
        rowC = C[i]
        for k in range(n):
            aik = rowA[k]
            rowB = B[k]
            for j in range(n):
                rowC[j] += aik * rowB[j]
    return C


def add_matrix(A, B):
    # Поэлементная сумма двух квадратных матриц одного размера
    n = len(A)
    return [[A[i][j] + B[i][j] for j in range(n)] for i in range(n)]


def sub_matrix(A, B):
    # Поэлементная разность двух квадратных матриц одного размера
    n = len(A)
    return [[A[i][j] - B[i][j] for j in range(n)] for i in range(n)]


def strassen_core(A, B, cutoff):
    """
    Рекурсивное ядро алгоритма Штрассена.

    Идея:
      - если размер блока мал (n <= cutoff), используем классический
        тройной цикл;
```

```
  - если размер блока нечётный, тоже возвращаемся к классике,
    чтобы не дополнять до степени двойки;
  - иначе (n чётное и достаточно большое) делим обе матрицы
    на четыре блока n/2 × n/2 и применяем формулы Штрассена.
"""
n = len(A)
if n <= cutoff:
    return matmul_classic(A, B)

if n % 2 == 1:
    # Нечётный размер: используем классический алгоритм без дополнения
    return matmul_classic(A, B)

m = n // 2

# Разбиение A на блоки
a11 = [row[:m] for row in A[:m]]
a12 = [row[m:] for row in A[:m]]
a21 = [row[:m] for row in A[m:]]
a22 = [row[m:] for row in A[m:]]

# Разбиение B на блоки
b11 = [row[:m] for row in B[:m]]
b12 = [row[m:] for row in B[:m]]
b21 = [row[:m] for row in B[m:]]
b22 = [row[m:] for row in B[m:]]

# 7 рекурсивных умножений (формулы Штрассена)
m1 = strassen_core(add_matrix(a11, a22), add_matrix(b11, b22), cutoff)
m2 = strassen_core(add_matrix(a21, a22), b11, cutoff)
m3 = strassen_core(a11, sub_matrix(b12, b22), cutoff)
m4 = strassen_core(a22, sub_matrix(b21, b11), cutoff)
m5 = strassen_core(add_matrix(a11, a12), b22, cutoff)
m6 = strassen_core(sub_matrix(a21, a11), add_matrix(b11, b12), cutoff)
m7 = strassen_core(sub_matrix(a12, a22), add_matrix(b21, b22), cutoff)

# Комбинация промежуточных результатов в блоки C11..C22
c11 = sub_matrix(add_matrix(add_matrix(m1, m4), m7), m5)
c12 = add_matrix(m3, m5)
c21 = add_matrix(m2, m4)
c22 = add_matrix(sub_matrix(add_matrix(m1, m3), m2), m6)

# Склейка блоков в одну матрицу C размера n×n
C = [c11[i] + c12[i] for i in range(m)] + \
    [c21[i] + c22[i] for i in range(m)]
return C
```

```
def matmul_strassen(A, B, cutoff=64):
    """
    Обёртка над strassen_core.

    Проверяет корректность размеров и запускает рекурсивный алгоритм.
    Здесь нет дополнения матриц до степени двойки: при "неудобных"
    размерах (например, n нечётное) алгоритм автоматически переключается
    на классическое умножение.
    """
    n = len(A)
    if n == 0:
        return []

    if n != len(B) or n != len(A[0]) or n != len(B[0]):
        raise ValueError("Матрицы должны быть квадратными и одинакового размера")

    return strassen_core(A, B, cutoff)


def main():
    # Чтение входных данных
    n = int(input())
    A = read_matrix(n)
    B = read_matrix(n)

    # Умножение матриц гибридным алгоритмом Штрассена
    C = matmul_strassen(A, B, cutoff=64)

    # Вывод результата
    for row in C:
        print(*row)


if __name__ == "__main__":
    main()
```