```
import numpy as np
import time
from random import randint
import matplotlib.pyplot as plt
from math import log2

[3] N = [2 ** k for k in range(1, 10)]

    A = list()
    B = list()
    for n in N:
      a = np.random.randint(1, 1000, size = (n, n))
      b = np.random.randint(1, 1000, size = (n, n))
      A.append(a)
      B.append(b)
```

A, B là hai mảng chứa các ma trận có kích thước n*n, mỗi phần tử có giá trị trong khoảng (1, 1000). Điểm khác so với yêu cầu là N sẽ có giá trị là 2k với k trong khoảng (1, 9) thay vì (10, 32). Lí do là vì thời gian tính toán lớn và máy không đủ memory để lưu ma trận có kích thước lớn (k > 14).

```
def matrix_multiplication(x, y):
  n = len(x)
  result = np.zeros((n, n))

  for i in range(n):
    for j in range(n):
      for k in range(n):
        result[i][j] += x[i][k] * y[k][j]

  return result
```

```
[5] time_1 = list()
    result_1 = list()
    for x, y in zip(A, B):
      start_time = time.time()
      result_1.append(matrix_multiplication(x, y))
      time_1.append(time.time() - start_time)
```

Thuật toán nhân ma trận cơ bản, với tham số đầu vào là hai ma trận vuông kích thước n * n. Ta tiến hành lấy từng hàng của ma trận x, nhân với từng cột của ma trận y, ứng với mỗi hàng cột ta tiến hành lấy các phần tử cùng index nhân với nhau sau đó cộng tất cả lại. Dễ dàng thấy thuật toán này có độ phức tạp O(N3).

Lần lượt áp dụng thuật toán lên các ma trận trong hai mảng A, B. Thời gian thực hiện thuật toán mỗi lần sẽ lưu vào *time_1*.

```
def split(matrix):
  row, col = matrix.shape
  row2, col2 = row//2, col//2
  return matrix[:row2, :col2], matrix[:row2, col2:], matrix[row2:, :col2], matrix[row2:, col2:]
```

Hàm *split()*, dùng để tức ma trận vuông n * n (n chẵn) thành 4 ma trận nhỏ. Hàm này sẽ bổ trợ cho thuật toán Strassen.

```
def strassen(x, y):
    if len(x) == 1:
        return x * y

    a, b, c, d = split(x)
    e, f, g, h = split(y)

    p1 = strassen(a + d, e + h)
    p2 = strassen(c + d, e)
    p3 = strassen(a, f - h)
    p4 = strassen(d, g - e)
    p5 = strassen(a + b, h)
    p6 = strassen(c - a, e + f)
    p7 = strassen(b - d, g + h)

    c11 = p1 + p4 - p5 + p7
    c12 = p3 + p5
    c21 = p2 + p4
    c22 = p1 + p3 - p2 + p6

    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

    return c
```

Thuật toán Strassen được cài đặt giống như file hướng dẫn. Ta sử dùng đệ quy để tính các ma trận P với điều kiện dừng là khi ma trận truyền vào là ma trận vuông kích thước 1.

np.hstack() dùng để ghép ma trận theo chiều ngang. *np.vstack()* dùng để ghép ma trận theo chiều dọc.

Lưu ý là thuật toán Strassen được cài đặt chỉ có thể chạy được với ma trận vuông kích thước 2k. Nếu không, trong quá trình chia ma trận lớn thành bốn ma trận con, sẽ có lúc ma trận có kích thước lẻ, và vì vậy mà không thể chia thành bốn ma trận nhỏ cùng kích thước được.

Về độ phức tạp thuật toán, ta thấy mỗi lần chạy thuật toán Strassen, ta sẽ gọi đệ quy 7 lần. Sau đó, cộng ma trận để thành ma trận kết quả bằng kích thước ma trận đầu vào sẽ có độ phức tạp O(N2). Vậy ta có:

$$T(N) = 7T(N/2) + O(N2)$$

Theo Master's Theorem (Định lí Thợ) thì:

$$T(N) = O(N\log2(7))$$

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

```
T(n) = aT(n/b) + f(n) where a >= 1 and b > 1
```

There are following three cases:

1. If $f(n) = O(n^c)$ where $c < \text{Log}_b a$ then $T(n) = \Theta(n^{\text{Log}_b a})$

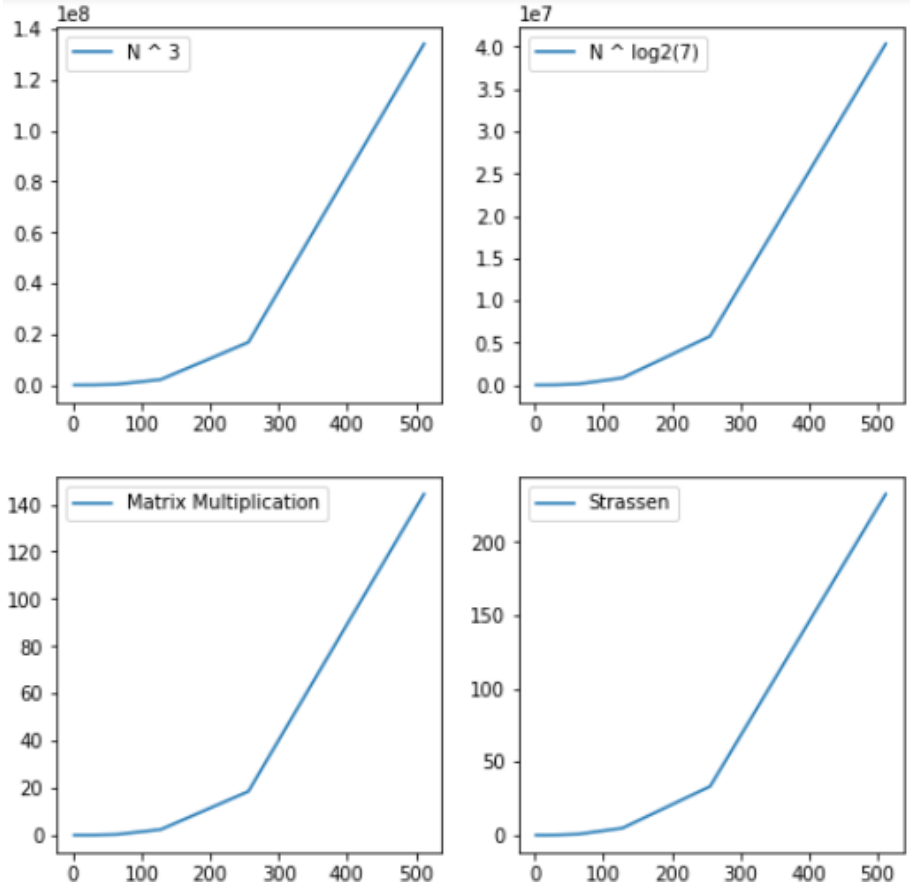
2. If $f(n) = \Theta(n^c)$ where $c = \text{Log}_b a$ then $T(n) = \Theta(n^c \text{Log } n)$

3. If $f(n) = \Omega(n^c)$ where $c > \text{Log}_b a$ then $T(n) = \Theta(f(n))$

```
[7]  time_2 = list()
     result_2 = list()
     for x, y in zip(A, B):
       start_time = time.time()
       result_2.append(strassen(x, y))
       time_2.append(time.time() - start_time)
```

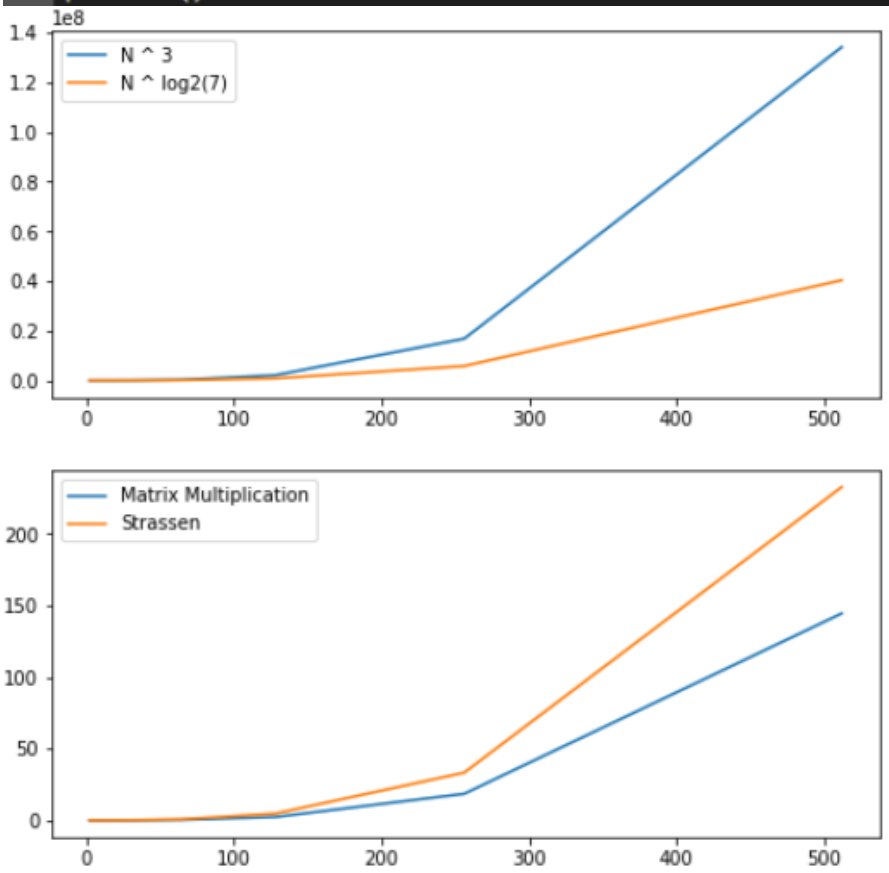
Chạy và lưu thời gian thực thi vào *time_2*.

```
11] plt.figure(figsize=(8, 8))

    plt.subplot(221)
    plt.plot(N, [n ** 3 for n in N], label = "N ^ 3")
    plt.legend()

    plt.subplot(222)
    plt.plot(N, [n ** log2(7) for n in N], label = "N ^ log2(7)")
    plt.legend()

    plt.subplot(223)
    plt.plot(N, time_1, label = "Matrix Multiplication")
    plt.legend()

    plt.subplot(224)
    plt.plot(N, time_2, label = "Strassen")
    plt.legend()

    plt.show()
```



Do chênh lệch giữa $\log_2 7$ và 3 khá nhỏ nên ta thấy biểu đồ khá giống nhau.

```
plt.figure(figsize=(8, 8))

plt.subplot(211)
plt.plot(N, [n ** 3 for n in N], label = "N ^ 3")
plt.legend()
plt.plot(N, [n ** log2(7) for n in N], label = "N ^ log2(7)")
plt.legend()

plt.subplot(212)
plt.plot(N, time_1, label = "Matrix Multiplication")
plt.legend()
plt.plot(N, time_2, label = "Strassen")
plt.legend()

plt.show()
```



Ở đây, ta thấy thuật toán Strassen khi code sẽ chạy chậm hơn thuật toán nhân ma trận cơ bản. Đây là do thuật toán Strassen phải lưu trữ rất nhiều ma trận trong đệ quy, điều này đồng nghĩa với việc hệ thống phải thực hiện cấp vùng nhớ để lưu trữ liên tục.

Về việc nhân ma trận với độ phức tạp $O(N^2)$, điều này là không thể, vì chỉ việc chọn dòng và cột thôi đã có phức tạp đó rồi. Theo Wikipedia, thuật toán nhân ma trận tốt nhất ở thời điểm hiện tại có độ phức tạp $O(N^{2.373})$.