ALESSANDRO SOARES DA SILVA

MATRICULA: 20231023705

2° LISTA DE ALGORITMO

1 - Mostre numericamente com sua implementações dos algoritmos de multiplicação de matrizes que o algoritmo de Strassen é mais rápido que o algoritmo convencional.

R = Construção da lógica de Strassen, vamos considerar uma matriz quadrada 2x2.

Calculo convencional:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} x \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

Lógica Strassen: Divisão da matriz em 4 submatrizes e cada submatriz e composta pelos seguintes valores:

$P_1 = A(F-H) = AF – AH$

$P_2 = (A+B)H = AH + BH$

$P_3 = (C+D)E = CE + DE$

$P_4 = D(G-E) = DG – DE$

$P_5 = (A+D)(E+H) = AE + AH + DE + DH$

$P_6 = (B-D)(G+H) = BG + BH – DG - DH$

$P_7 = (A-C)(E+F) = AE + AF – CE - CF$

$$\begin{bmatrix} (P_5+P_4-P_2+P_6) & (P_1+P_2) \\ (P_3+P_4) & (P_1+P_5-P_3-P_7) \end{bmatrix}$$

$P_5 + P_4 - P_2 + P_6 = AE + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} – \cancel{DE} - \cancel{AH} - \cancel{BH} + BG + \cancel{BH} – \cancel{DG} - \cancel{DH} =$ **AE+ BG**

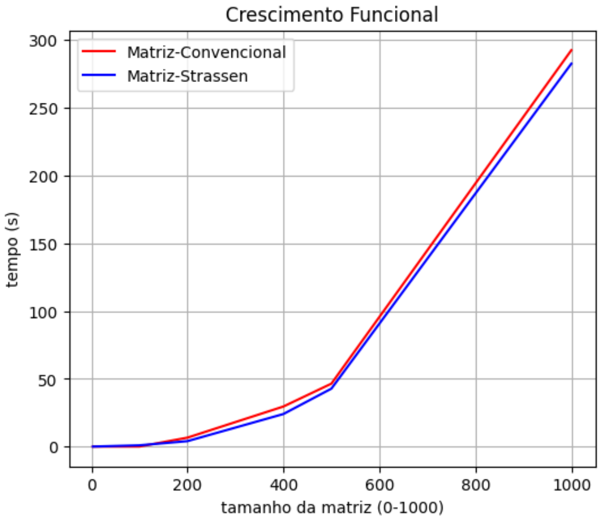
$P_1 + P_2 = AF – \cancel{AH} + \cancel{AH} + BH =$ **AF + BH**

$P_3 + P_4 = CE – \cancel{DE} + DG – \cancel{DE} =$ **CE + DG**

$P_1 + P_5 - P_3 – P_7 = \cancel{AF} – \cancel{AH} + \cancel{AE} + \cancel{AH} + \cancel{DE} + DH - \cancel{CE} – \cancel{DE} - \cancel{AE} - \cancel{AF} + \cancel{CE} + CF =$ **CF + DH**

Pelo teorema mestre temos que: $T(n) = 7T(n/2) + cn^2$:. $T(n) = O \quad n^{[\log_b a]} = n^{[\log_2 7]} = n^{2.81}$

	MATRIZ $\Theta(N^3)$	MATRIZ $\Theta(N^{2.81})$
N = 3X3	0m 0,037 s	0m 0,036 s
N = 100X100	0m 0,693 s	0m 0,966 s
N = 200X200	0m 6,622 s	0m 4,036 s
N = 400X400	0m 29,585 s	0m 24,052 s
N = 500X500	0m 46,452 s	0m 42,769 s
N = 1000X1000	4m 52,415 s	4m 42,494 s



Implementação Multiplicação de Matriz Convencional

```python
from optparse import OptionParser
def read(filename):
    lines = open(filename).read().splitlines()
    A = []
    B = []
    matrix = A
    for line in lines:
        if line != "":
            matrix.append([int(el) for el in line.split("\t")])
        else:
            matrix = B
    return A, B
def printMatrix(matrix):
    for line in matrix:
        print("\t".join(map(str, line)))
def standardMatrixProduct(A, B):
    n = len(A)
    C = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option(
        "-i",
        dest="filename",
        default="1000.in",
        help="input file with two matrices",
        metavar="FILE",
    )
    (options, args) = parser.parse_args()

    A, B = read(options.filename)
    P = standardMatrixProduct(A, B)
    printMatrix(P)
```

Implementação Multiplicação de Matriz Strassen

```python
def read(filename):
    lines = open(filename).read().splitlines()
    A = []
    B = []
    matrix = A
    for line in lines:
        if line != "":
            matrix.append([int(el) for el in line.split("\t")])
        else:
            matrix = B
    return A, B

def print_matrix(matrix):
    for line in matrix:
        print("\t".join(map(str, line)))

def ikj_matrix_product(A, B):
    n = len(A)
    C = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for k in range(n):
            for j in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

```python
def add(A, B):
    n = len(A)
    C = [[0 for j in range(0, n)] for i in range(0, n)]
    for i in range(0, n):
        for j in range(0, n):
            C[i][j] = A[i][j] + B[i][j]
    return C

def subtract(A, B):
    n = len(A)
    C = [[0 for j in range(0, n)] for i in range(0, n)]
    for i in range(0, n):
        for j in range(0, n):
            C[i][j] = A[i][j] - B[i][j]
    return C

def strassenR(A, B):
    n = len(A)
    if n <= LEAF_SIZE:
        return ikj_matrix_product(A, B)
    else:
        # initializing the new sub-matrices
        new_size = n // 2
        a11 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        a12 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        a21 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        a22 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
```

```python
        b11 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        b12 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        b21 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        b22 = [[0 for j in range(0, new_size)] for i in range(0, new_size)]

        aResult = [[0 for j in range(0, new_size)] for i in range(0, new_size)]
        bResult = [[0 for j in range(0, new_size)] for i in range(0, new_size)]

        # dividing the matrices in 4 sub-matrices:
        for i in range(0, new_size):
            for j in range(0, new_size):
                a11[i][j] = A[i][j]  # top left
                a12[i][j] = A[i][j + new_size]  # top right
                a21[i][j] = A[i + new_size][j]  # bottom left
                a22[i][j] = A[i + new_size][j + new_size]  # bottom right

                b11[i][j] = B[i][j]  # top left
                b12[i][j] = B[i][j + new_size]  # top right
                b21[i][j] = B[i + new_size][j]  # bottom left
                b22[i][j] = B[i + new_size][j + new_size]  # bottom right

        # Calculating p1 to p7:
        aResult = add(a11, a22)
        bResult = add(b11, b22)
        p1 = strassenR(aResult, bResult)  # p1 = (a11+a22) * (b11+b22)
        aResult = add(a21, a22)  # a21 + a22
        p2 = strassenR(aResult, b11)  # p2 = (a21+a22) * (b11)

        bResult = subtract(b12, b22)  # b12 - b22
        p3 = strassenR(a11, bResult)  # p3 = (a11) * (b12 - b22)

        bResult = subtract(b21, b11)  # b21 - b11
        p4 = strassenR(a22, bResult)  # p4 = (a22) * (b21 - b11)

        aResult = add(a11, a12)  # a11 + a12
        p5 = strassenR(aResult, b22)  # p5 = (a11+a12) * (b22)

        aResult = subtract(a21, a11)  # a21 - a11
        bResult = add(b11, b12)  # b11 + b12
        p6 = strassenR(aResult, bResult)  # p6 = (a21-a11) * (b11+b12)

        aResult = subtract(a12, a22)  # a12 - a22
        bResult = add(b21, b22)  # b21 + b22
        p7 = strassenR(aResult, bResult)  # p7 = (a12-a22) * (b21+b22)

        # calculating c21, c21, c11 e c22:
        c12 = add(p3, p5)  # c12 = p3 + p5
        c21 = add(p2, p4)  # c21 = p2 + p4

        aResult = add(p1, p4)  # p1 + p4
        bResult = add(aResult, p7)  # p1 + p4 + p7
        c11 = subtract(bResult, p5)  # c11 = p1 + p4 - p5 + p7
```

```python
        aResult = add(p1, p3)  # p1 + p3
        bResult = add(aResult, p6)  # p1 + p3 + p6
        c22 = subtract(bResult, p2)  # c22 = p1 + p3 - p2 + p6

        # Grouping the results obtained in a single matrix:
        C = [[0 for j in range(0, n)] for i in range(0, n)]
        for i in range(0, new_size):
            for j in range(0, new_size):
                C[i][j] = c11[i][j]
                C[i][j + new_size] = c12[i][j]
                C[i + new_size][j] = c21[i][j]
                C[i + new_size][j + new_size] = c22[i][j]
        return C

def strassen(A, B):
    assert type(A) == list and type(B) == list
    assert len(A) == len(A[0]) == len(B) == len(B[0])

    # Make the matrices bigger so that you can apply the strassen
    # algorithm recursively without having to deal with odd
    # matrix sizes
    nextPowerOfTwo = lambda n: 2 ** int(ceil(log(n, 2)))
    n = len(A)
    m = nextPowerOfTwo(n)
    APrep = [[0 for i in range(m)] for j in range(m)]
    BPrep = [[0 for i in range(m)] for j in range(m)]
    for i in range(n):
        for j in range(n):
            APrep[i][j] = A[i][j]
            BPrep[i][j] = B[i][j]
    CPrep = strassenR(APrep, BPrep)
    C = [[0 for i in range(n)] for j in range(n)]
    for i in range(n):
        for j in range(n):
            C[i][j] = CPrep[i][j]
    return C

if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option(
        "-i",
        dest="filename",
        default="400.in",
        help="input file with two matrices",
        metavar="FILE",
    )
    parser.add_option(
        "-l",
        dest="LEAF_SIZE",
        default="8",
        help="when do you start using ikj",
        metavar="LEAF_SIZE",
    )
    (options, args) = parser.parse_args()

    LEAF_SIZE = int(options.LEAF_SIZE)
    A, B = read(options.filename)

    C = strassen(A, B)
    print_matrix(C)
```

2 - Dê limites assintóticos superiores e inferiores para T(n) em cada uma das recorrências a seguir. Considerando que T(n) é contante para $n \leqslant 2$. Torne seus limites tão restritos quanto possível e justifique suas respostas.

Teoria: O teorema mestre resolve recorrências do tipo: $T(n)=aT(\frac{n}{b})+\Theta(n^k)$ com $a \geq 1, b > 1 \, e \, k \geq 0$ constantes.

n^k	Nível	Tamanho	#nós	Tempo por nó
/ \	0	n	1	n^k
(n/b)^k (n/b)^k	1	n/b	a	$(n/b)^k$
/ \ \	2	n/b^2	a^2	$(n/b^2)^k$
O(n) O(n) O(n)	i	n/b^i	a^i	$(n/b^i)^k$
	$\log_b n$	1	$n^{\log_b a}$	1^k

$n/b^x = 1 \Rightarrow b^x = n$

$(1)\, se\, a > b^k, então\, T(n) = \Theta(n(\log_b a))$
$(2)\, se\, a = b^k, então\, T(n) = \Theta(n^k) logn$
$(3)\, se\, a < b^k, então\, T(n) = \Theta(n^k)$

$a.T(n)=2T(n/2)=n^4$
R = T(n) ϵ $\Theta(n^4)$:. a = 2, b = 2, f(n) = n^4, $n^{\log_2 2}$ = n, Então $n^3 = \Omega(n^{\log_2 2+2})$, $a/b^k = 2/2^3 = 1/4 < 1$, caso 3 do teorema mestre.

$b.T(n)=T(7n/10)=n$
R = T(n) ϵ $\Theta(n)$:. a = 1, b = 10/7, f(n) = n, $n^{\log_{10/9} 1}$ = n, Então n = $\Omega(n^{\log_{10/7} 1+1})$, $a/b^k = 1/(10/7) = 7/10 < 1$, caso 3 do teorema mestre.

$c.T(n)=16T(n/4)=n^2$
R = T(n) ϵ $\Theta(n^2 \log n)$:. a = 16, b = 4, f(n) = n^2, $n^{\log_4 16} = n^2$, Então $n^2 = \Theta(n^{\log_4 16})$, caso 2 do teorema mestre.

$d.T(n)=7T(n/3)=n^2$
R = Caso 3 do teorema mestre, T(n) ϵ $\Theta(n^2)$

$e.T(n)=7T(n/2)=n^2$
R = T(n) ϵ $\Theta(n^{\log 7})$):. a = 7, b = 2, f(n) = n^2, $n^{\log_2 7}$, 2 < log7 < 3, Nós temos que $n^2 = O(n^{\log_2 7-\varepsilon})$ para um constante ϵ > 0. Temos então o caso 1 do teorema mestre.

$f.T(n)=2T(n/4)=\sqrt{n}$
R = Caso 2 do teorema mestre, T(n) ϵ $\Theta(n^{1/2} \log n)$

$g.T(n)=T(n-2)=n^2$
R = Vamos provar pro indução que $T(n) \leq cn^3$
base: n=1. Sabemos que T(1) = 1 e $c * 1^3 = c$
Então vale se $c \geq 1$
Queremos mostrar que $T(n) \leq cn^3$ se n>1
Hipótese: $T(k) \leq ck^3$ para todo $1 \leq k \leq n$

Sabemos que $T(n) = T(n-2) + n^2$

Como $n-2 < n$ (pois $n>1$, vale por hipótese que $T(n-2) \leq c\,(n-2)^3 = n^3-6n^2+12n-8$

$T(n) = T(n-2) + n^2 \leq n^3-6n^2+12n-8+n^2$

$n^3-5n^2+12n-8 \leq cn^3$

$n^3 \leq cn^3 \Rightarrow 1 \leq c$

3 – Este problema examina três algotimos para procurar um valor x em um arranjo não ordenado A que consiste em n elementos.

Considere a seguinte estratégia aleatória: escolha um índice aleatório i em A. Se A[i] = x, então terminamos; caso contrário, continuamos a busca escolhendo um novo índice aleatório em A. Continuamos a escolher índices aleatórios em A até encontrarmos um índice j tal que A[j] = x ou até verificarmos todos os elementos de A. Observe que, toda vez escolhemos um índice no conjunto inteiro de índices, é possível que examinemos um dado elemento mais de uma vez.

a . Escreva pseudocódigo para um procedimento *Random-Search* pra implementar a estratégia citada. Certifique-se de que o seu algoritmo termina quando todos os índices em A já tiverem sido escolhidos.

R = Suponha que A tenha N elementos. Nossos algoritmos irá rastrear os elementos que foram vistos e serão adicionandos a um contador C, cada vez que um novo elemento for verificado. Assim que este contador atingir N, saberemos que todos os elementos foram verificados.

Algoritmo – Random-Search

inicializa um vetor **p** de tamanho **n** contendo todos os elementos
Inicializa um inteiro **c** e **i** em 0

```
import numpy as np
import random
A = [1,3,5,15,20,30,41,9,10,23,13,15,80,16,17]
c = 0
i = 0
while  c != (len(A)):
    i = random.randint(0, len(A)-1)
    if A[i] == 10:
        print (i)
        c = 15
    else:
        print("Tentativa {}".format(c+1))
        c = c+1
```

b. Suponha que exista exatamente um índice i tal que A[i] = x. Qual é o número esperado de índices em A que devemos escolher antes de encontrarmos x e *Random-Search terminar?*

*R = Seja **n** a variável aleatória para número de pesquisa necessárias, então*

$$E(X) = E\left[\sum_{i=1}^{n} n_i\right]$$

$$E(X) = \sum_{i=1}^{n} \frac{n}{n-(i+1)}$$

$$E(X) = n\sum_{i=1}^{n} \frac{1}{i}$$

$$E(X)=\frac{1}{n}$$

c. Generalizando sua solução para a parte (b), suponha que existam $k \geq 1$ índices i tais que A[i] = x. Qual é o número esperado de índices em A que devemos escolher antes de encontramos x e *Random-Search* terminar? Sua resposta deve ser uma função de n e k.

*R = Seja **n** a variável aleatória para número de pesquisa necessárias, então*

$$E(X)=E\left[\sum_{i=1}^{n} n_i\right]$$
$$E(X)=\sum_{i=1}^{n} \frac{n}{n-(i+1)}$$
$$E(X)=n\sum_{i=1}^{n} \frac{1}{i} \quad \therefore\ n=k$$

$$E(X)=\frac{k}{n}$$

d. Suponha que não exista nenhum índice i tal que A[i] = x. Qual é o número esperado de índices em A que devemos escolher antes de verificarmos todos os elementos de A e *Random-Search* terminar?

R = *Seja **n** a variável aleatória para número de pesquisa necessárias, então*

$$E(X)=E\left[\sum_{i=1}^{n} n_i\right]$$
$$E(X)=\sum_{i=1}^{n} \frac{n}{n-(i+1)}$$
$$E(X)=n\sum_{i=1}^{n} \frac{1}{i} \quad = b(\ln b + O(1))$$

Agora, considere um algoritmo de busca linear determinística, que denominamos *Deterministic-Search*, Especificamente, o algoritmo procura A[i] = x em ordem, considerando A[1], A[2], A[3], …, A[n] até encontrar A[i] = x ou chegar ao fim do arranjo. Considere todas as permutações possíveis do arranjo de entrada igualmente prováveis.

```
import numpy as np
import random
c=0
i=0
A = [1,3,5,15,20,30,41,9,10,23,13,15,80,16,17]
n = len(A)
while  i != n:
    if A[i] == 10:
        print ("X encontrado na posição {}".format(i+1))
        i = 15
    else:
        print("Tentativa {}".format(i+1))
        i = i+1
```

e. Suponha que exista examente um índice i tal que A[i] = x. Qual é o tempo de execução do caso médio de *Deterministic-Search*? Qual é o tempo de execução do pior caso de *Deterministic-Search?*

R = O tempo médio de execução do caso médio é ***(n+1)/2*** *e o tempo de execução do pior caso é* ***n****.*

f. Generalizando sua solução para parte (e), suponha que existam $k \geq 1$ índices o tais que A[i] = x. Qual é o tempo de execução do caso médio de *Deterministic-Search?* Qual é o tempo de execução do pior caso de *Deterministic-Search*? Sua resposta dece ser uma função de n e k.

R = NÃO CONSEGUI FAZER

g. Suponha que não exista nenhum índice i tal que A[i] = x. Qual é o tempo médio de execução do caso médio de *Deterministic-Search*? Qual é o tempo de execução do pior caso de *Deterministic-Search*?

R = O tempo médio e o pior caso de execução são ambos **n**.

Finalmente, considere um algoritmo aleatorizado *Scamble-Search* que funciona primeiro permutando aleatoriamente o arranjo de entrada e depois executando a busca linear determinística dada anteriormente para o arranjo permutado resultante.

```
import numpy as np
import random
c=0
i=0
n = 15
A = []

while len(A) < n:
    numero = random.randint(1, 15)
    if numero not in A:
        A.append(numero)
print (A)

n = len(A)
while  i != n:
    if A[i] == 10:
        print ("X encontrado na posição {}".format(i+1))
        i = 15
    else:
        print("Tentativa {}".format(i+1))
        i = i+1
```

h. Sendo k o número de índices i tais que A[i] = x, dê os tempos de execução esperado e do pior caso de *Scamble-Search* para os casos nos quais k=0 e k=1. Generalize sua solução para tratar o caso no qual $k \geq 1$.

R = Scamble-Search funciona de forma idêntica ao determinisc-search, exceto que adicionamos ao tempo de execução o tempo necessário para randomizar a matriz de entrada.

j. Qual dos três algoritmos de busca você usaria? Explique sua resposta.

R = Eu usaria a pesquisa determinística, uma vez que tem o melhor tempo de execução esperado e é garantido que terminará após **n** etapas, ao contário da pesquisa aleatória. Além disso, no tempo que leva para permutar aleatoriamente a matriz de entrada , poderíamos ter realizado uma pesquisa linear de qualquer maneira.