Trabalho 1 para a Disciplina de AA-18.2

Professor: Marcus Poggi
Grupo: Francisco & Lauro

Questão 1.1

É pedido um algoritmo que garante que o número de quedas é da ordem de O(sqrt(n)) e que tem a determinação do valor da altura crítica garantida.

Descrição do Algoritmo:

Step 1 -> Testar alturas múltiplas de sqrt(N) até que o primeiro frasco quebre. Sabemos que essa altura de quebra pode ser escrita da forma X * sqrt(N), onde X >= 1.

Step 2 -> Sabendo que o primeiro frasco quebrou na altura X * sqrt(N) e que ele não quebrou em (X - 1) * sqrt(N), vamos utilizar o segundo frasco disponível para tentar todas as alturas no intervalo semi-aberto ((X - 1) * sqrt(N), X * sqrt(N)], começando a menor altura e aumentando a altura testada de 1 em 1. A altura em que o frasco 2 quebrar é a altura mínima de quebra que estávamos buscando.

Corretude:
Como sabemos que existe uma altura crítica de intervalo [1, N], podemos concluir que existe um primeiro x, tal que x * sqrt(N) >= alturaCrítica. Após encontrar esse primeiro múltiplo, ficamos com um intervalo de tamanho = sqrt(N) para testar de forma exaustiva. Como percorremos esse intervalo de maneira crescente nas alturas, garantimos por construção que a altura na qual o segundo frasco quebra é a alturaCrítica que queremos encontrar.

Análise de Complexidade:
Step 1 -> Claramente O(sqrt(N)), pois só temos sqrt(N) múltiplos de sqrt(N) menores que a altura máxima N.

Step 2 -> O intervalo no qual todas as alturas serão testadas de forma crescente possui tamanho = sqrt(N). Logo a complexidade desse passo é sqrt(N).

Complexidade total: O(sqrt(N)) + O(sqrt(N)) = O(sqrt(N))

Questão 1.2

É pedido para oferecermos um algoritmo que garanta que serão realizadas menos de K * 2^(N / K) quedas, para K = 3, K = 4 e depois generalize.

Descrição do Algoritmo para K = 3

Passo 1 -> Vamos testar as alturas múltiplas de N^(⅔) até que o frasco 1 quebre.
Passo 2 -> Agora temos um espaço de busca de tamanho = N^(⅔).Vamos testar os múltiplos de N^(⅓) que estão dentro do espaço de busca, até que o frasco 2 quebre.
Passo 3 -> Agora temos um espaço de busca de tamanho = N^(⅓), vamos testar todas as alturas nesse intervalo, da menor para a maior. Quando o frasco 3 quebrar, significa que encontramos a altura crítica.

Esse algoritmo basicamente explora a ideia do algoritmo para 2 frascos, mas aproveita o fato de termos mais frascos.

Complexidade:
Passo 1-> O(N^(1/3))
Passo 2 -> O(N^(1/3))
Passo 3 -> O(N^(1/3))
Complexidade total = O(3 * N^(⅓))

Descrição do Algoritmo para K = 4

Passo 1 -> Vamos testar as alturas múltiplas de N^(3^4) até que o frasco 1 quebre.
Passo 2 -> Agora temos um espaço de busca de tamanho = N^(3/4).Vamos testar os múltiplos de N^(2^4) que estão dentro do espaço de busca, até que o frasco 2 quebre.
Passo 3 -> Agora temos um espaço de busca de tamanho = N^(2/4) , vamos testar todas as alturas múltiplas de N^(1/4) no intervalo até que o frasco 3 quebre.
Passo 4 -> Agora temos um espaço de busca de tamanho = N^(1/4), vamos testar todas as alturas de forma crescente. Quando o frasco 4 quebrar, sabemos que encontramos a altura crítica.

Esse algoritmo basicamente explora a ideia do algoritmo para 2 frascos, mas aproveita o fato de termos mais frascos.

Complexidade:
Passo 1 -> O(N^(1/4))
Passo 2 -> O(N^(1/4))
Passo 3 -> O(N^(1/4))
Passo 4 -> O(N^(1/4))
Complexidade total = O(4 * N^(1/4))

Caso para K geral:

Vamos adotar uma estratégia bem similar a dos casos descritos acima.

Passo 1 -> Sabemos que a altura crítica está no intervalo [1, N]. Testar as alturas múltiplas de N^(K - 1 / K) até que o primeiro frasco quebre.

Passo 2 -> Agora temos um espaço de busca de tamanho N^(K - 1/ K). Vamos testar nele, todas as alturas múltiplas de N^(K - 2/ K) dentro desse espaço de busca até que o frasco 2 quebre.

Continuamos os passo dessa forma, até o k-ésimo passo

Passo K -> Temos um espaço de busca de tamanho = N^(1 / K). Vamos testar todas as alturas nesse intervalo, de forma crescente. A altura que o frasco K quebrar é a altura crítica.

Complexidade:
Passo 1 -> O(N^(1/K))
Passo 2 -> O(N^(1/K))
.
.
Passo K -> O(N^(1/K))

Total: K * O(N^(1/K)) = O(K * N^(1/K))

Bônus -> Descrição de um algoritmo (bit-a-bit) para quando K é divisor de B (B = número de bits necessários para representar a maior altura possível). Vou descrever para 3 frascos, mas para qualquer K tal que B % K == 0 a ideia é análoga.

Descrição do Algoritmo para 3 frascos:

Seja B o número de bits máximo necessário para representar a altura máxima de queda. Vamos dividir esse número em 3 grupos com a mesma quantidade de bits.

Vamos separar a altura em 3 partes de tamanho = (B / 3).
Decomponha a altura crítica da seguinte forma.

parte | 1 | 2 | 3 |
alturaCrítica = 01010 | 10010 | 11110

A ideia do algoritmo se resume a usar cada um dos frascos disponíveis para descobrir uma das 3 partes da resposta.

Vamos começar encontrando os bits mais significativos da altura crítica. Inicialmente, vamos considerar todos os grupos de bits mais à direita como 1’s.

alturaDeTeste 00000 | 11111 | 11111

Vamos testar todas as possíveis alturas da parte mais à esquerda, até que o frasco 1 quebre.

Parte 1:

Alturas testadas -> 00000 | 11111 | 11111
00001 | 11111 | 11111
00010 | 11111 | 11111
.
.
.
01010 | 11111 | 11111 <- Altura de quebra do Frasco 1

Agora sabemos a resposta para os primeiros B / 3 bits. Vamos manter essa resposta fixada na primeira parte e testar a segunda parte. Vamos sempre manter os bits mais à esquerda do grupo testado como 1’s.

Parte 2:

Alturas testadas -> 01010 | 00000 | 11111
01010 | 00001 | 11111
.
.
.
01010 | 10010 | 11111 -> Altura de quebra do Frasco 2

Temos agora a resposta para a segunda parte. Falta agora encontrar quais são os B / 3 bits menos significativos da altura crítica.

Parte 3:

Alturas testadas -> 01010 | 10010 | 00000
01010 | 10010 | 00001
.
.
.
01010 | 10010 | 11110 -> Altura de quebra do Frasco 3 = Altura crítica!

O número total de quedas provocadas pelo algoritmo é no máximo = 3 * 2^(B/3).
Para cada um dos 3 frascos, nós iremos testar todos os valores de [0, 2^(N/3) - 1] e isso nos leva ao total de quedas mencionado acima acima.

Para K, com B % K == 0, vale que a complexidade é = K * 2^(B / K)

Detalhe de implementação:

A versão implementada para 1, 2, 4, 8 e 16 frascos goza do fato que B % K == 0. Foi feita uma pequena modificação na forma de comparar alturas para diminuir constantes.

Pela forma qual a qual o algoritmo é implementado, quando vamos testar um dos conjuntos de bits descritos acima. Sabemos que todos os grupos de bits a esquerda do grupo atual possuem a valoração igual a da altura crítica. Sabemos também que os bits dos grupos a direita do grupo que está sendo testado são todos iguais a 1.
Foi utilizada uma forma de comparação especial de um determinado número com a altura crítica.

Digamos que estamos utilizando 3 frascos, e estamos querendo encontrar a valoração dos bits da parte 2.

```
Parte                1   |   2   |   3
altura crítica   = 00110 | 10010 | 01001
altura testada = 00110 | 00001 | 11111
```

Como sabemos que os bits da parte 1 já estão corretos, não nos interessa olhar para eles. Como sabemos que os bits da terceira parte da altura de teste estão todos inicializados para 1, pelo algoritmo que propus, também não nos interessa olhar para eles, pois sabemos que que altura testada será necessariamente >= que a altura crítica na terceira parte.

As observações feitas acima nos inspiraram a olhar exclusivamente para os bits da parte 2 da altura testada e da altura crítica. Surge então a função que compara apenas os bits da parte selecionada da altura testada com os bits dessa parte da altura crítica. Ela é apenas uma particularização da comparação habitual.

```
// Essa funcao retorna -1 se altura1 < altura2, 0 se altura1 = altura2 e 1 se altura1 < altura2
ComparacaoHabitual(altura1, altura2):
        Para cada Bit em (0, B):
                Se altura1[Bit] != altura2[Bit]:
                        retorna altura1[Bit] - altura2[Bit]
        retorna 0

// A funcao que usei na implementacao, se importa com apenas o bits da parte interessada
// Ela retorna -1 se alturaTestada < alturaCritica e 1 caso contrário.
ComparacaoEspecial(alturaTestada, alturaCritica, inicio, fim):
        Para cada Bit em (inicio, fim):
                Se altura1[Bit] < altura2[Bit]:
```

```
                retorna -1
            Se altura1[Bit] > altura2[Bit]:
                retorna 1
        retorna 1
```

A função acima não altera a complexidade assintótica da implementação. Foi utilizada apenas para diminuir a constante e para poder representar cada grupo como um unsigned int. Não usamos nenhuma informação sobre a resposta que não deveríamos ter acesso no decorrer do algoritmo, apenas criamos uma função de comparação especial.
Se for estritamente necessário que só seja possível comparar todos os bits de uma vez de duas alturas, a implementação pode ser adaptada para usar a ComparacaoHabitual com pouco esforço.

Questão 1.3

A menor complexidade assintótica possível é O(log(N)). Vamos apresentar o algoritmo que atinge essa complexidade.

Algoritmo Log(n)

A ideia é basicamente explorar o espaço de busca, realizando uma busca binária para encontrar a menor altura de quebra (altura crítica). Essa estratégia só é possível, pois a função quebra descrita a seguir é monotônica.

Quebra(x) = { 0 se x < altura crítica, 1 caso contrário }

Descrição do algoritmo:

Low = 1, High = N

```
Enquanto Low < High:
    Mid = (Low + High) / 2
    Se Quebra(Mid) == 1:
        High = Mid
    Senão:
        Low = Mid + 1
```

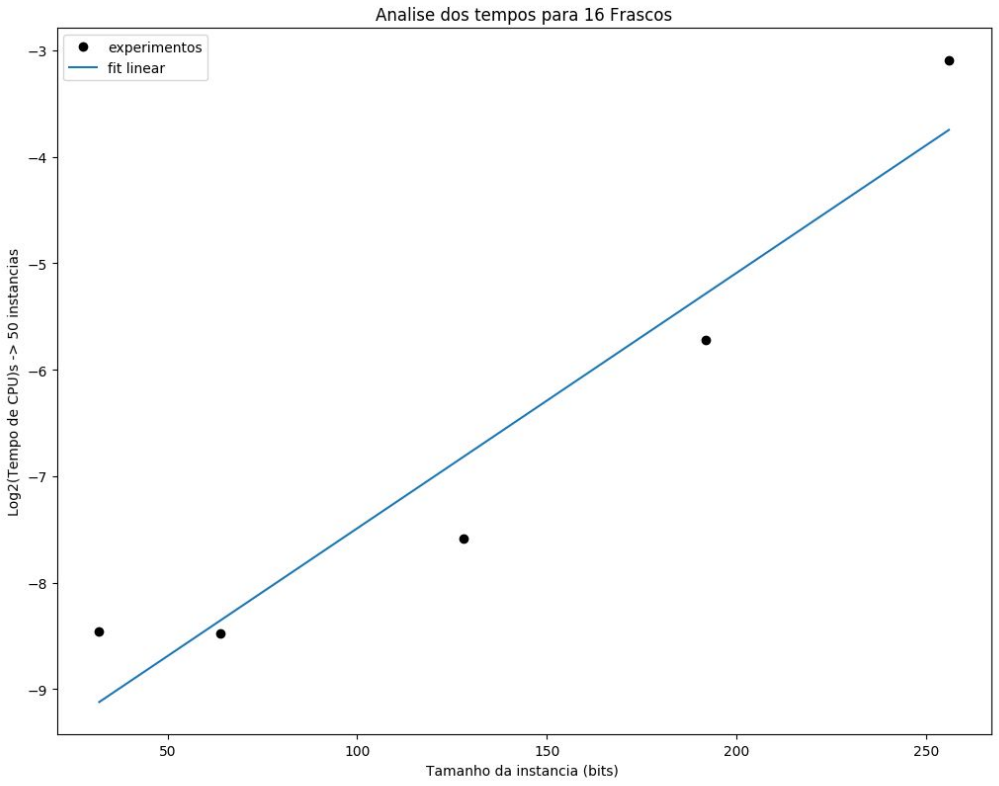
Como na busca binária, a cada passo o espaço de busca se reduz pela metade (arredondado para cima se o espaço for ímpar no pior caso).

O algoritmo descrito acima vai gerar no máximo ceil(log(N)) quedas.

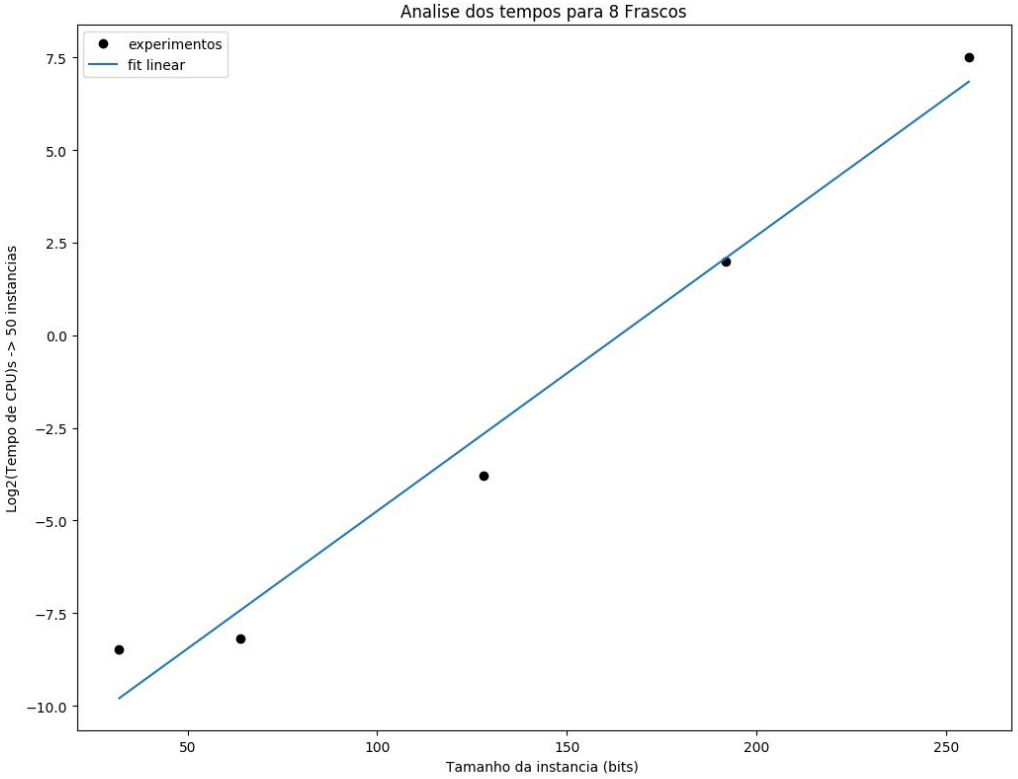
Discussão dos resultados e da implementação

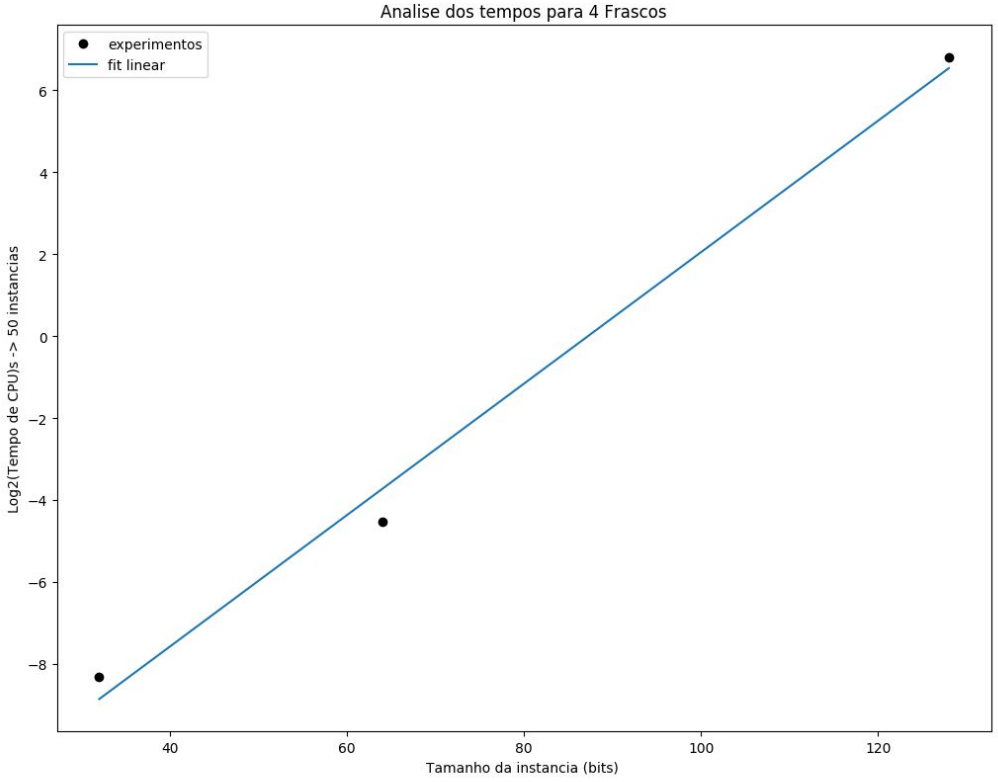
Para K = 1, 2, 4, 8, 16 o código elaborado realiza as contas bit-a-bit. Utilizamos scipy + numpy + matplotlib para plotar gráficos e analisar os tempos de execução vs tamanho da instâncias.

Pudemos observar uma correlação linear de: Log do tempo de execução X Tamanho da Instância em bits. O que nos diz, em outras palavras, que existe uma correlação exponencial do tempo de execução com o tamanho das instâncias em bits.

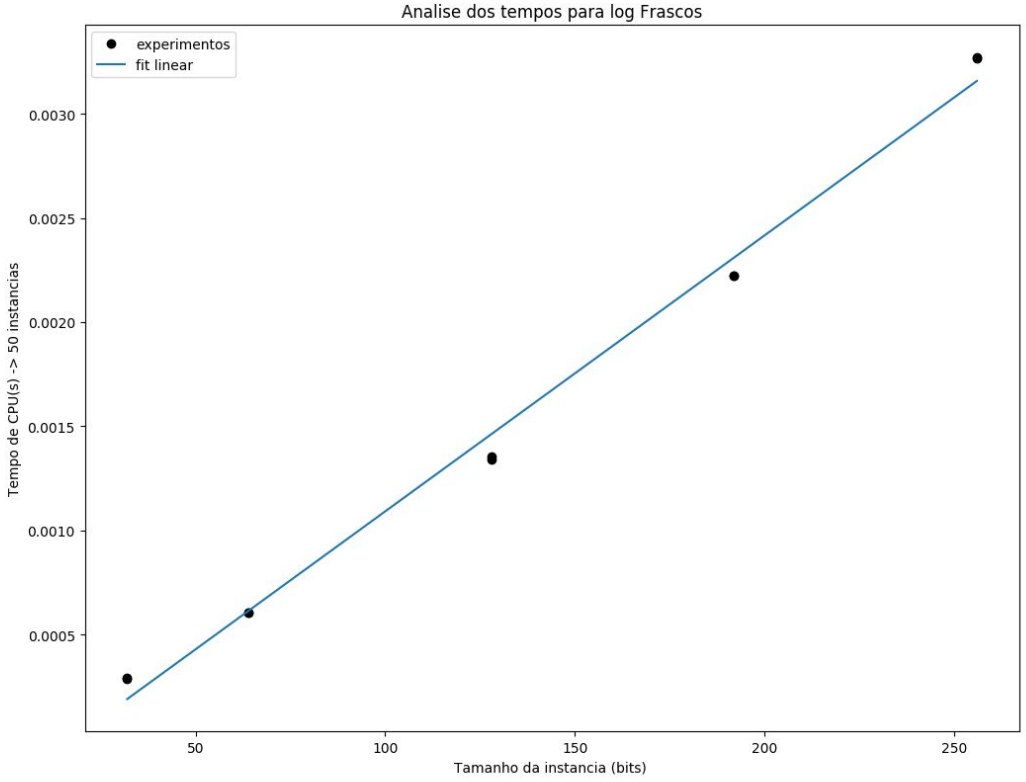


Utilizamos o log dos tempos para analisar esse caso, pois o fit com função logarítmica não ficou muito bom de ver no gráfico. Essa tendência se repete para todas as quantidades de frascos do grupo(1, 2, 4, 8, 16)...


Analise dos tempos para 8 Frascos
experimentos
fit linear
Log2(Tempo de CPU)s -> 50 instancias
Tamanho da instancia (bits)
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
50
100
150
200
250


Analise dos tempos para 4 Frascos
experimentos
fit linear
Log2(Tempo de CPU)s -> 50 instancias
Tamanho da instancia (bits)
6
4
2
0
−2
−4
−6
−8
40
60
80
100
120

Já para o caso com quantidade irrestrita de frascos, no qual adotamos a estratégia da busca binária, vemos uma correlação linear entre o tempo de execução das 50 instâncias X tamanho da instância em bits. Não foi necessário aplicar o log do tempo de execução antes.



Os tempos de execução, os gráficos, os códigos e os executáveis serão disponibilizados em um zip.

As medições de tempo foram feitas utilizando o high_resolution_clock de C++ e o módulo timeit de Python.

Questão 2

LETRA A – Método Pedestre

1) O algoritmo realiza a multiplicação de matrizes. Para isso:

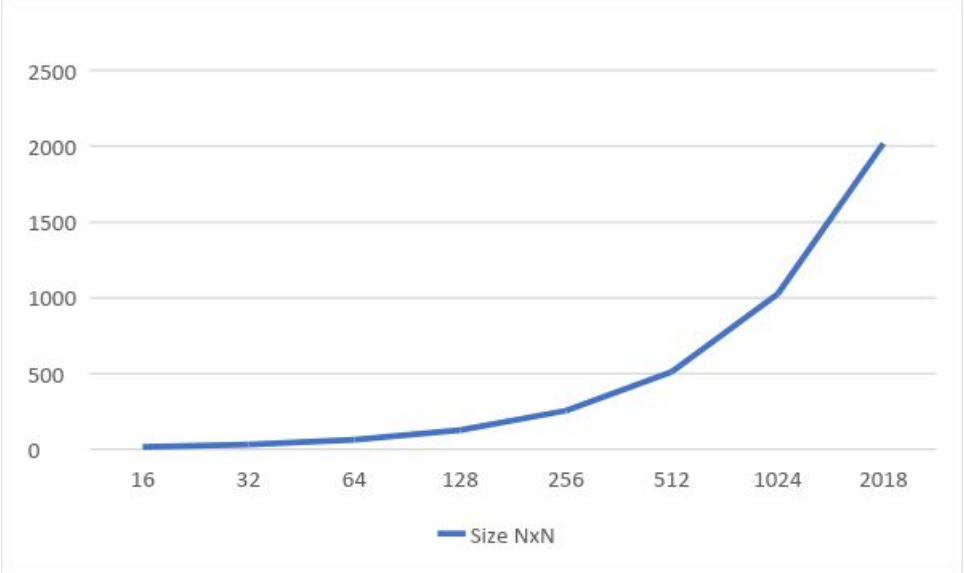
Percorre cada linha e da M1, onde M1 é a 1a matriz e multiplica cada vetor-linha da M1 por cada vetor coluna da M2. O resultado da multiplicação da linha i de M1 por coluna j de M2 é inserida na posição i,j da matriz resultante M3.

2) Para isso temos que fazer 3 For's aninhados para selecionar cada linha da M1, cada coluna da M2 e o terceiro vem de fato para percorrer e multiplicar.

3) O algoritmo vem direto do método de multiplicação de matrizes. Já que as matrizes são quadradas, cada For, vai iterar N vezes e como são 3 Fors aninhados temos O(n^3).

Segue abaixo os resultados:

Size NxN	Tempo(s)
16	0,0001
32	0,0002
64	0,0015
128	0,0107
256	0,0860
512	0,9050
1024	12,5116
2018	138,9660



Execução:

Para executar a multiplicação de matrizes com o método pedestre, é preciso ir na função **main** do arquivo source.c e descomentar da linha 310 a linha 327 e comentar o código da linha 333 a 348.

O código que está nessas linhas irão rodar 10x para cada tamanho de matriz 2,32,64,128,256,512,1024 e 2048, criando aleatoriamente os seus valores e irá imprimir os tempos.

LETRA B - Método de Strassen

1) O algoritmo realiza a multiplicação de matrizes. Para isso:

Utiliza uma abordagem de divisão em conquista com um truque matemático observado por Strassen. A divisão e conquista por si só, particiona a matriz em 4 pedaços iguais de tamanho SIZE /2 e soma recursivamente os valores de ae + bg, af + bh, ce + dg and cf + dh conforme pode ser visto na imagem abaixo:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Dessa forma, nós fazemos 8 multiplicações recursiva e 4 adições. Adição de matrizes tem complexidade O(N^2), então a complexidade desse algoritmo pode ser escrita como:

$T(N) = 8T(N/2) + O(N^2)$

Utilizando o Teorema Mestre, temos: a=8,b=2,k=2 e portanto 8 > 2^2 e assim O(n^log(8,base=2)) = O(n^3).

E assim não há melhora em relação ao método pedestre. Mas a observação de Strassen era de que era possível chegar no produto de duas matrizes através das seguintes fórmulas:

P1= A * (F – H)
P2= H * (A + B)
P3= E * (C + D)
P4= D

P5= (A + D) * (E + H)
P6= (B – D) * (G + H)
P7= (A – C) * (E + F)

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

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

E com isso, diminuir o número de recursões para **7.** Com isso, temos: T(N) = 7T(N/2) + $O(N^2)$, e portanto aplicando o teorema mestre temos a=7,b=2e k=2, gerando o resultado final de O(n^log(7,base=2)) = O(n^2.8074).

Etapas do algoritmo:

O caso base foi definido como 64x64 e não uma matriz de 1x1. O motivo dessa escolha foi porque quando iniciamos o algoritmo com o caso base de uma matriz 1x1, vimos computador dar crash algumas vezes. Pesquisando online foi observado que isso é decorrente de um overhead por chamar a função recursiva até 1 elemento.

De acordo com esse estudo:

http://ce.u-sys.org/Veranstaltungen/SiWiR1%20(Ru%CC%88de)/Uebungen%20WS0910/Julian2_Steffi_Balthi/ex01-presentation.pdf

O breakeven para o algoritmo de Strassen foi uma matriz de 64x64 que usamos como nossa referência.

Apesar do Strassen ter uma menor complexidade tem um Big-O constant bem maior. É como comparar n^3 com n^2.8 + 100n^2 e portanto não vale a pena descer até 1 elemento.

Ao atingir 64, estamos fazendo o Fall Back para o algorítimo pedestre que performa melhor em instância menores.

O segundo passo do algoritmo segue a seguir:

Particiona a matriz em sub matrizes n/2.

Começa a calcular os Ps a partir de:

Faz a submatriz para o espaço específico aonde a soma ou subtração de Ps ocorrerão.

Executa a soma ou subtração, por exemplo: a+b ou f-h

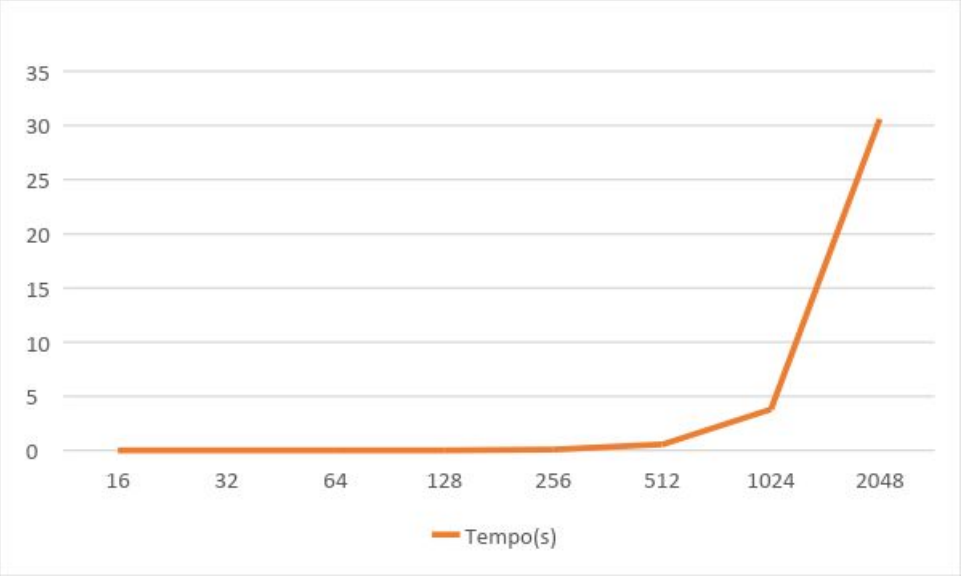
Chama a Multiplicação recursivamente pela partição conforme a formula de Strassen, por exemplo a*(f-h).

Aloca a soma o “p” dentro do seu quadrante específico.

Executa os passos acima de P1 a P7.

Seguem os resultados abaixo:

NxN	Tempo(s)
16	0
32	0,0002
64	0,0026
128	0,0128
256	0,0938
512	0,5547
1024	3,807
2048	30,6102



Comparativo:

Size NxN	Pedestre	Strassen
16	0,0001	0
32	0,0002	0,0002
64	0,0015	0,0026
128	0,0107	0,0128
256	0,0860	0,0938
512	0,9050	0,5547
1024	12,5116	3,807
2018	138,9660	30,6102

Para o cálculo do tempo de execução foram rodadas 10 vezes para o mesmo tamanho da amostra e calculado uma média no final. Para o cálculo do tempo foi usado a biblioteca time.h.

Execução:

Para executar a multiplicação de matrizes com o método Strassen, é preciso ir na função **main** do arquivo source.c e comentar da linha 310 a linha 327 e descomentar o código da linha 333 a 348.

O código que está nessas linhas irão rodar 10x para cada tamanho de matriz 2,32,64,128,256,512,1024 e 2048, criando aleatoriamente os seus valores e irá imprimir os tempos.

Letra C - Método O(n^2) descrito no paper disponibilizado no enunciado do trabalho

Descrição do Algoritmo: Seguimos a risca o que foi proposto pelo paper.

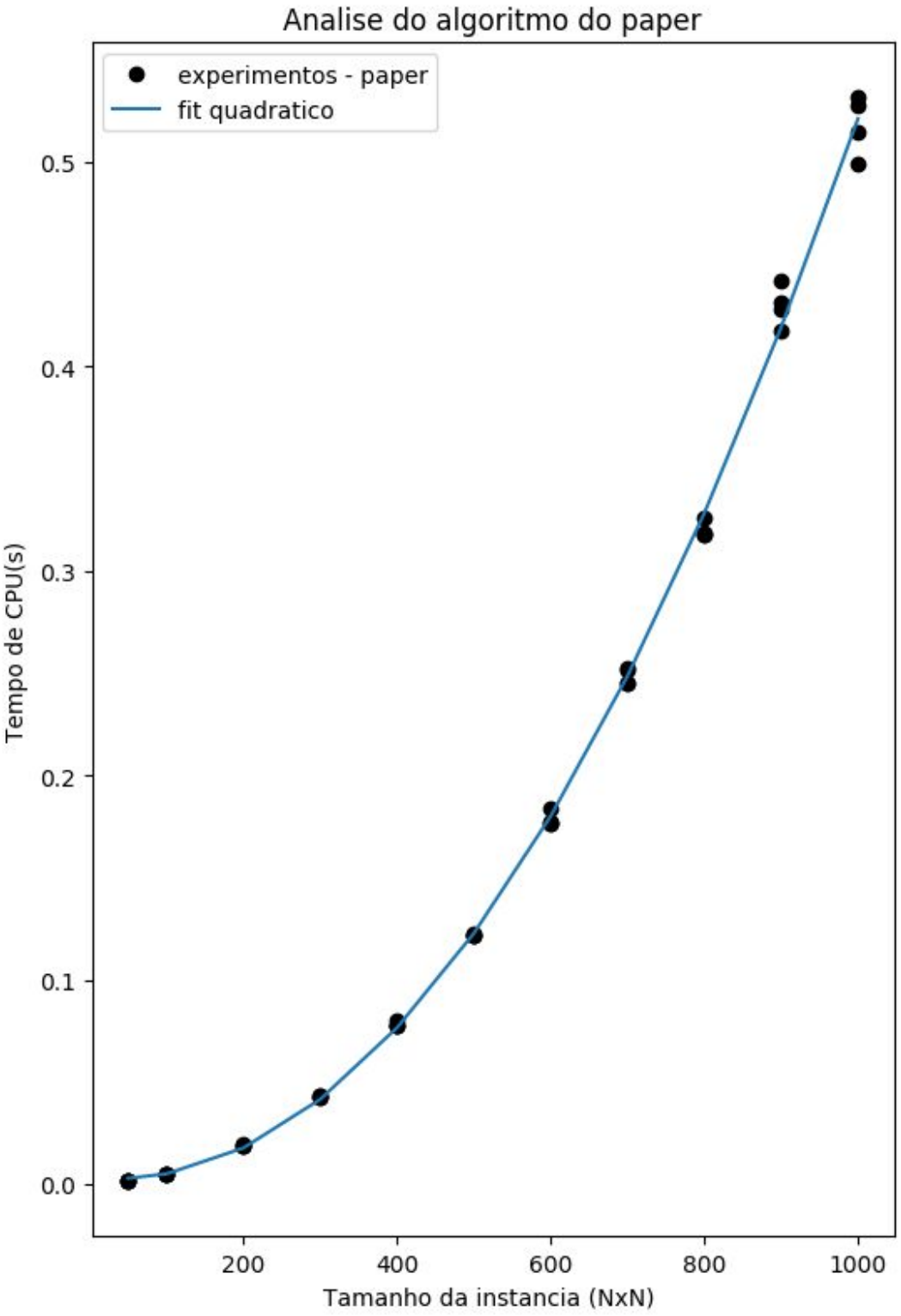
Complexidade: O algoritmo é de fato O(n^2) se estivermos lidando matrizes N x N com N pequeno e se os valores das matrizes forem limitados por uma constante baixa. Ao implementar o Algoritmo e ler o paper com calma, fica evidente que não é realizado mais do que 2 nested-loops de trabalho em nenhum momento. A grande questão do paper é que a forma de calcular os valores que ele propõe, envolve fazer cálculos com números monstruosos e sabemos que multiplicação e soma de números de 1000 dígitos não pode ser feito em tempo constante.

Tabela de tempos:

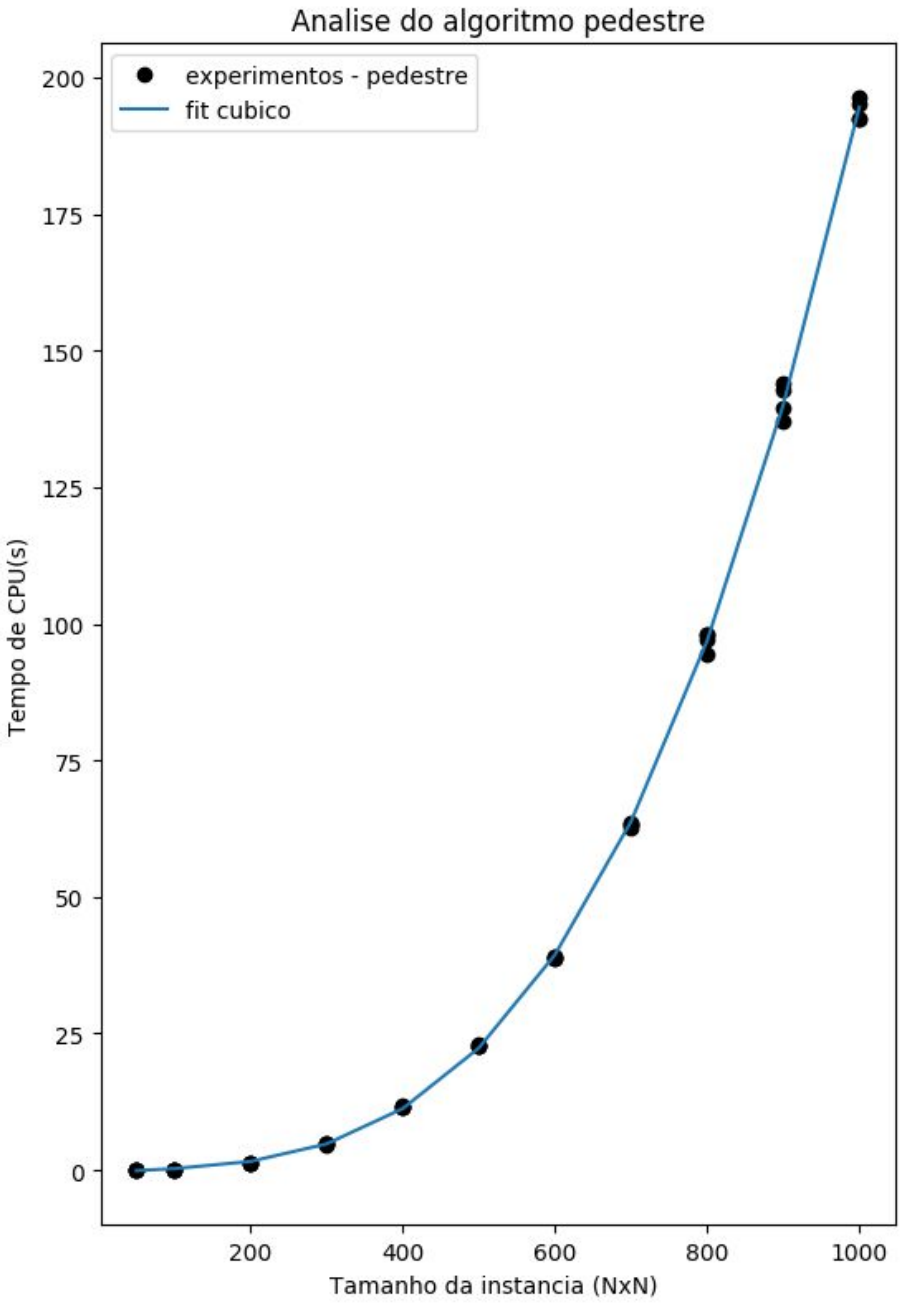
Dimensão da Matriz	Valor máximo Permitido	Tempo Pedestre (s)	Tempo Artigo (s)
100x100	1	0.1750	0.0049
200x200	1	1.3817	0.0187
300x300	1	4.6520	0.0424
400x400	1	11.5849	0.0775
500x500	1	22.9454	0.1220
600x600	1	39.1444	0.1760

700x700	1	63.6889	0.2519
800x800	1	94.2639	0.3186
900x900	1	139.5154	0.4309
1000x1000	1	196.3926	0.5273

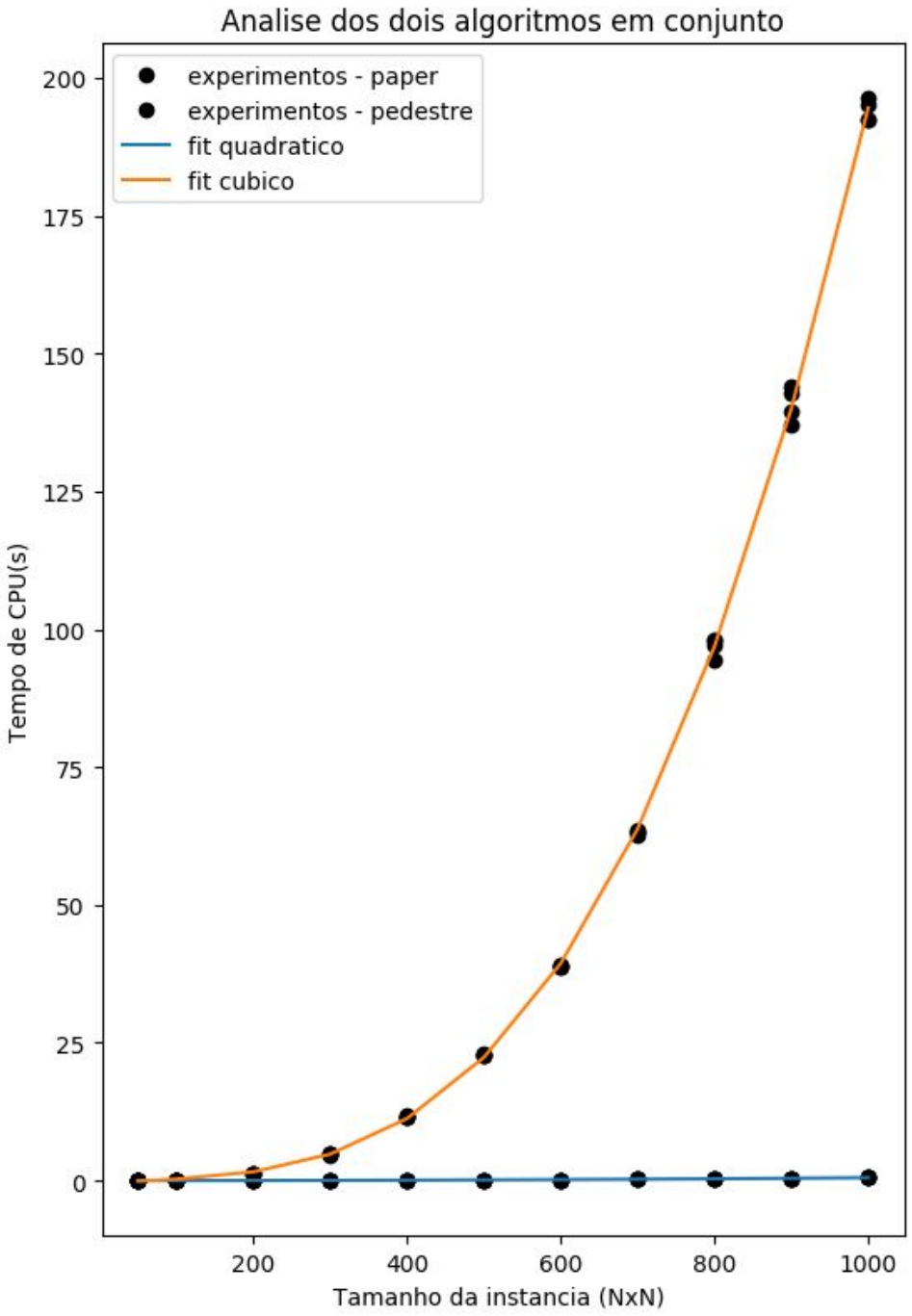
Para matrizes 0-1, pudemos observar que o Algoritmo proposto no paper se comporta como O(n^2). Provavelmente porque os números não ficaram grandes o suficiente para sentirmos o impacto real do custo das operações de multiplicação e adição para inteiros realmente grandes.


Analise do algoritmo do paper
experimentos - paper
fit quadratico
Tempo de CPU(s)
0.0
0.1
0.2
0.3
0.4
0.5
200
400
600
800
1000
Tamanho da instancia (NxN)

O algoritmo pedestre como esperado, teve um fit cúbico.



A seguinte figura sobrepõe os 2 gráficos. É notável que o algoritmo do paper se mantém muito mais rápido que o pedestre para as instâncias testadas.



Além das instâncias de matrizes 0-1, testamos instâncias com valores máximos até 10000000000. Ainda assim o Algoritmo do paper ganhava de lavada do algoritmo pedestre e parecia manter um crescimento próximo a O(n^2). As instâncias realmente grandes que provavelmente mostrariam que a complexidade não é O(n^2) não puderam ser rodadas, pois levariam dias para produzir os resultados.

A medição dos tempos foi realizada utilizando o módulo timeit de Python 3.