

Aplicações e Problemas Computacionais em Matemática Discreta

Guilherme Locca Salomão e João Victor Mendes Freire

7 de julho de 2019

1 Introdução

1.1 O que é Matemática Discreta

A Matemática Discreta é um ramo da matemática que não possui uma definição formal precisa e exata. Ela é descrita como o ramo que estuda conjuntos contáveis, ou seja, conjuntos finitos ou que possuem a mesma cardinalidade do conjunto dos naturais. Ao se observar o conjunto dos números reais, por exemplo, podemos escolher dois números arbitrários, como 0 e 1. Entre esses dois números, existe uma infinidade de outros números, e, portanto, não podemos considerar 0 o “primeiro” e 1 o “segundo”. O 0,5 existe entre eles. Bem como o 0,05, e o 0,005. Logo, não conseguimos mapear a cardinalidade dos números naturais para cada número real. O conjunto dos inteiros é infinito assim como os reais, mas, diferentemente deles, é enumerável. Portanto, podemos mapear a cardinalidade dos naturais neles. Assim sendo, embora contenha infinitos elementos, o conjunto dos inteiros é uma estrutura discreta, de interesse da matemática discreta, ao contrario dos reais, que é uma estrutura contínua.

1.2 Por que é importante?

Dispositivos digitais estão presentes em inúmeros contextos no mundo contemporâneo. Smartphones, computadores pessoais, servidores, semáforos, sistemas de um banco, os sistemas de um veículo e milhares de outros sistemas são digitais. Devido a isto, são construídos por circuitos lógicos, que funcionam baseados em estruturas discretas. A comunicação entre bancos, governos e até mesmo entre pessoas é segura quando se utiliza Criptografia para proteger os dados. A matemática discreta é bastanten importante em se tratando de desenvolver novos e melhores algortimos para criptografar e descriptografar dados privados. Além desse aspecto computacional, a matemática discreta oferece as ferramentas para que, por exemplo, um carteiro descubra qual a rota de entregas mais eficiente. Estes são apenas dois exemplos de impactos da matemática discreta na vida de uma pessoa comum. Existem inúmeros outros, e ainda mais

para aqueles interessados em aprofundar seus conhecimentos na área de Ciência da Computação.

1.3 Tópicos de Interesse

Alguns dos principais tópicos em matemática discreta são: teoria dos números, conjuntos, funções, relações, recorrências e teoria dos grafos.

1.4 Organização do Documento

O documento foi estruturado da seguinte forma: em cada capítulo, temos um conjunto de exercícios computacionais propostos pelo Professor. Para cada exercício, temos uma seção que introduz o problema e descreve o processo de resolução. Depois, temos uma seção com os códigos de resolução. E, por fim, temos uma de considerações finais, na qual explicamos eventuais dificuldades e algumas análises de tempo e espaço dos algoritmos.

1.5 Materiais e Métodos

Os algoritmos foram desenvolvidos em duas plataformas, sendo a primeira deles utilizando a distribuição Linux Manjaro, um editor de texto em conjunto com o terminal. Foram utilizadas as Linguagens Python, na versão 3.7, e a linguagem C. A segunda plataforma utilizou o sistema operacional MacOS, também utilizou um editor de texto (Visual Studio Code) em conjunto com o terminal. Na segunda, as atividades foram resolvidas usando apenas a linguagem Python 3.7.

2 Capítulo I: Project Euler - Conjunto 1

O primeiro conjunto de problemas contém os exercícios: 100001st Prime Number, Special Pythagorean Triplet e Distinct Powers.

2.1 10001st prime number

2.1.1 Metodologia

Um primo é definido por ter apenas dois divisores, 1 e ele mesmo, nesse sentido, a solução seria uma aplicação direta da definição

2.1.2 Resultados e discussões

```
1 primo_atual = 3
2 primos = [2]
3 eh_primo = 1 # flag que verifica se o numero é primo ou não
4 while( len(primos) != 10001): # loop ocorre enquanto não a lista de
    numeros primos é diferente a 10001
5     eh_primo = 1
```

```

6     primo_atual = primo_atual + 2 #C onsiderando que o unico numero
    primo par é o numero 2, podemos então pular todos os numero
    pares e considerar apenas os impares
7     for n in primos: # utilizando todos os primos anteriores
8         if primo_atual%n == 0: # verificamos se o numero a ser
    analizado no momento não é divisível por nenhum deles o que
    implicara que ele so tem um unico divisor alem de 1
9             eh_primo = 0
10        if eh_primo == 1: # passando nos testes o numero é adicionado a
    lista
11            primos.append(primo_atual)
12 print(primos[10000])

```

A resposta encontrada foi 104.743 como conferido no arquivo de respostas fornecido.

2.1.3 Considerações finais

Este tem uma solução bem direta porém pouco eficiente, certas escolhas como pular a verificação de numeros pares definitivamente reduzem o tempo de execução mas ainda sim não é possível deixá-lo instantâneo.

2.2 Special Pythagorean Triplet

2.2.1 Metodologia

Uma tripla pitagórica é quando a soma dos quadrados de dois números é igual ao quadrado de um terceiro número, ou seja $a^2 + b^2 = c^2$. Para resolver esse problema foi usada a lógica de que, C tem que ser o complemento da soma de A e B, e então de forma iterativa foi testada as possibilidades dessa combinação.

2.2.2 Resultados e discussões

```

1 soma = 1000 # esse é o resultado da soma dos 3 numeros
2 solucao = 0
3 for a in range(1, soma + 1): # Sera testada cada iteração a partir
    do caso base de A ser 1 ate A ser 1000
4     for b in range(a + 1, soma + 1): # então sera testado o B>A que
    então consiga satisfazer a solução
5         c = soma - a - b
6         if a * a + b * b == c * c:
7             solucao = a * b * c
8 print(solucao)

```

A saída do programa foi 31875000, compatível com a resposta no arquivo fornecido.

2.2.3 Considerações finais

A solução desse problema foi difícil de se traduzir para código pois há várias soluções possíveis porém cada uma tem sua dificuldade de implementação, entretanto, a "ideia" da solução foi fácil de se chegar.

2.3 Longest Collatz Sequence

2.3.1 Metodologia

Uma sequência collatz é uma sequência iterativa onde considerando a sequência iniciando em um número n , se n for par então $n = \frac{n}{2}$, se n for ímpar então, $n = 3n + 1$, até chegar ao caso base de $N = 1$. Para resolver esse problema foi uma aplicação direta dessa definição e então atualizar qual seria a maior sequência sempre que uma maior for encontrada.

2.3.2 Resultados e Discussões

```
1 start = 1 # começando pelo caso inicial
2 longest_number = 1 # Caso inicial da mais longa
3 longest_steps = 1 # Quantos passos ocorreram na mais longa
4 cur_steps = 0 # contador de passos
5 cur_number = 0 # número inicial atual
6 while(start < 1000000): # verificando todos os casos ate 1000000
7     cur_steps = 0
8     start = start + 2 # verifica apenas os numeros impares pois
9     esse tem as sequencias mais longas
10    cur_number = start
11    while (cur_number != 1): # loop ocorre enquanto não atingir o
12    caso base
13        if cur_number%2 == 0:
14            cur_number = cur_number/2
15            cur_steps = cur_steps+1
16        else:
17            cur_number = (3*cur_number)+1
18            cur_steps = cur_steps+1
19    if cur_steps > longest_steps: # atualiza se a quantidade de
20    passos da sequencia atual for maior que a sequencia maior
    anterior
    longest_number = start
    longest_steps = cur_steps
print(longest_number)
```

O retorno desse algoritmo foi de 837799, igual ao resultado fornecido no arquivo de respostas

2.3.3 Considerações finais

A solução desse problema foi direta porém achar detalhes que a otimização foi difícil, a única opção de otimização que pude encontrar foi a de pular os números pares pois estes sempre retornavam as sequências mais curtas.

3 Capítulo II: Project Euler - Conjunto 2

O segundo conjunto de problemas contém os exercícios: Summation of primes, Highly Divisible Triangular Number e Power Digit Sum.

3.1 Summation of primes

3.1.1 Metodologia

A solução desse exercício é direta e simples, achar todos os primos e então somá-los. Foi utilizado o Crivo de Eratóstenes para de forma mais eficiente encontrar todos os número primos até dois milhões e então somá-los.

3.1.2 Resultados e discussões

```
1 def achaPrimos(limite): # Função que ira procurar todos os primos
2   # menores que um certo limite
3   primos=[True for n in range(limite+1) ] # cria uma lista com n=
4   # limite elementos e os atribui o valor logico True
5   crivo=2 # cria uma chave do primeiro número primo
6   saida = [] # lista de saida
7   while crivo*crivo <= limite: # enquanto o quadrado da chave
8     # for menor que o limite, o loop ocorre
9     if(primos[crivo]==True): # verifica se a chave é um primo
10      for i in range(crivo*2,limite+1,crivo): # procura todos
11        # os números multiplos da chave
12        primos[i]=False # os transforma em falso, ou seja,
13        # não primos
14      crivo += 1 # aumenta a chave
15  for i in range(2,limite): # Traduz o indice dos elemento primos
16    # para uma lista de elementos
17    if primos[i]==True:
18      saida.append(i)
19  return saida
20
21 primos = achaPrimos(2000000)
22 soma = 0
23 for i in primos: # realiza a soma de todos os elementos
24   soma += i
25 print(soma)
```

A saída do programa foi 142913828922 como no arquivo de respostas fornecido

3.1.3 Conclusões finais

Para realizar esse programa, era necessário um algoritmo para achar números primos mais eficiente que a checagem pela definição, portanto a escolha do crivo, além disso foi mais incentivado o uso de tal algoritmo por temos um valor limite.

3.2 Highly Divisible Triangular Number

3.2.1 Metodologia

Um número triangular na posição N é a soma de todos os naturais menores que N , pensando nisso, foi necessário criar uma forma para encontrar o número triangular de posição N , e então criar uma forma de testar seus divisores.

3.2.2 Resultados e discussões

```

1 #include<stdlib.h>
2 #include<stdio.h>
3
4 int achaNumeroTriangular(int pos, int num_anterior); //Função que
   encontra o número de posição N utilizando o numero da posição N
   -1
5 int achaQtdDivisores(int num); //Função que acha o numero de
   divisores de um número
6
7
8 int main(){
9     int divisores=0; //contador de divisores
10    int pos_atual = 0; //posição atual a ser considerada
11    int num_atual=0; //Numero triangular da posição atual
12    while (divisores <500) // Realiza a operação ate o contador de
   divisores atinga 500 ou mais
13    {
14
15        num_atual = achaNumeroTriangular(pos_atual,num_atual);
16        divisores = achaQtdDivisores(num_atual);
17
18        pos_atual++;
19    }
20    printf("\n resultado: %d\n",num_atual);
21 }
22
23 int achaNumeroTriangular(int pos, int num_anterior){
24     return pos+num_anterior;
25 }
26
27 int achaQtdDivisores(int num){
28     int cont=0;
29     for(int i=1;i<=num;i++){
30         if(num%i==0){ //verifica se o resto da divisão é zero o que o
   caracteriza como divisor
31             cont++;
32         }
33     }
34     return cont;
35 }

```

O retorno desse algoritmo foi de 76576500 correspondendo com a resposta fornecida

3.2.3 Considerações finais

A solução desse problema era bem direta e fácil de se implementar, o que tornou viável usar a linguagem C para isso pois ela poderia realizar em um menor tempo as operações em comparação a python. Além disso outra otimização foi o uso do numero anterior para calcular o numero da posição atual pois assim se tornava um operação simples o que acelerava o processo.

3.3 Power Digit Sum

3.3.1 Metodologia

Um número triangular na posição N é a soma de todos os naturais menores que N , pensando nisso, foi necessário criar uma forma para encontrar o número triangular de posição N , e então criar uma forma de testar seus divisores.

3.3.2 Resultados e discussões

```
1 x = 2**1000 # numero a ser usado
2 div = 10 # potencia de 10 inicial
3 aux = 0
4 aux2 = 1
5 soma = 0 # soma final dos digitos
6 while(x>0): # realiza o loop até que o numero inicial seja 0
7     aux = x%div # armazena o digito a ser recebido nessa iteração
8     soma = soma+ (aux/aux2) # soma o digito a soma total, é usado
9     um auxiliar para garantir que o numero seja unitário
10    x = x - aux # remove o resto do número
11    aux2 =div # atualiza as potencias para poder continuar a operação
12    div = div*10
13 print(soma)
```

O resultado desse algoritmo foi 1366 igual ao resultado fornecido no arquivo de respostas

3.3.3 Considerações finais

A solução desse problema é simples porém é necessário uma estrutura muito flexível para comportar o número 2^{1000} , já que esse requer cerca de 1000 bits para armazenar, então, ao contrario da minha decisão inicial, foi necessário o uso de python para essa operação.

4 Capítulo III: Project Euler - Conjunto 3

O terceiro conjunto de problemas contém os exercícios: Amicable Numbers (Números Amigáveis), 1000-digit Fibonacci Number (Número de Fibonacci de 1000 Dígitos) e Distinct Powers (Potências Distintas).

4.1 Amicable Numbers

4.1.1 Metodologia

Seja $soma_divisores(n)$ a soma de todos os divisores de n (números menores que n que o dividem com resto 0). Sejam a, b inteiros diferentes. Se $soma_divisores(a) = b$, e $soma_divisores(b) = a$, dizemos que a e b são números amigáveis. O exercício em questão pergunta qual a soma de todos os números amigáveis de 1 à 10000. O exercício requer conhecimento da definição de divisível, e exigiu que se pensasse numa forma de não repetir os números encontrados.

4.1.2 Resultados e discussões

```
1 # Dado um número inteiro x, a função retorna a soma dos divisores
   inteiros de x
2 def soma_divisores(x):
3     soma = 0;
4     for i in range(1, x // 2 + 1):
5         if x % i == 0:
6             soma = soma + i
7     return soma
8
9 # Dado um número inteiro x, a função retorna a soma dos números
   amigáveis no intervalo [1, x)
10 def soma_amigaveis(x):
11     soma = 0
12     a = 1
13     while a < x:
14         b = soma_divisores(a)
15         # se b > a, para que não se calcule um número amigável que
   já apareceu
16         if b > a and soma_divisores(b) == a:
17             soma = soma + a + b
18             a = a + 1
19     return soma
20
21 res = soma_amigaveis(10000)
22 print(res)
```

A resposta encontrada nessa solução foi 31626, que é a mesma presente no site que contém as respostas para a lista.

4.1.3 Considerações finais

A solução do problema é bem direto ao ponto: para cada número de $[1, x)$, calcule a soma de seus divisores, e depois calcule a soma dos divisores desse resultado. Se forem iguais, e b for maior que a , adicione a soma deles na soma acumulada. Quanto a dificuldade de não se repetir números já calculados, basta que nós coloquemos a limitação que $b > a$, assim garantindo que, se $b < a$, $a + b$ já foi adicionado a soma quando o número que estávamos testando era a .

4.2 1000-digit Fibonacci Number

4.2.1 Metodologia

A sequência de Fibonacci é definida por uma relação de recorrência, onde $Fib(n) = Fib(n - 1) + Fib(n - 2)$, sendo $Fib(1) = Fib(2) = 1$ os números iniciais. O problema pedia para encontrar qual o índice do primeiro número da sequência de Fibonacci que possui 1000 dígitos. A busca pela solução começou pensando na forma de calcular quantos dígitos tem um número, e rapidamente chegamos a conclusão que se dividissemos o número várias vezes por 10 até que sobrasse apenas 0, teríamos o número de casas. Depois, partimos para a solução que, diferentemente do problema anterior, teve que ser alterada por razões de performance.

4.2.2 Resultados e discussões

A solução para o problema pode ser simplificada a função *num_digitos(x)*, que encontra o valor usando o *log10* de *x*. Para o problema de resolver *Fib(x)*, encontramos ao todo 3 soluções. A primeira, *fib(x)*, partiu da definição da sequência. Para valores pequenos, a solução resolve o problema, mas o tempo de solução é $O(2^n)$, que torna ela completamente inviável para números como o que possui 1000 dígitos. Tendo em vista o tempo exponencial da anterior, utilizamos os conhecimentos de recorrência, e encontramos a fórmula fechada presente em *fib_fechado(x)*. Essa função resolve o problema do *x*-ésimo Fibonacci em tempo linear. A primeira vista ele parece ideal, mas o número de vezes que se eleva os números (da ordem de *x*) faz com que os valores causem um overflow. Portanto, a velocidade é boa, mas os números crescem rápido demais para a memória disponível para um inteiro. Por fim, encontramos uma solução que resolve em tempo linear com uso constante de memória. A função *fib_definitivo(x)* resolve o problema usando três variáveis auxiliares, e o loop realiza $O(x)$ iterações.

```
1 from math import sqrt, floor, log10
2
3 # Dado um inteiro x, calcula o número de dígitos
4 def num_digitos(x):
5     return floor(1 + log10(x))
6
7 # Dado inteiro x, retorna o x-ésimo valor da sequência de Fibonacci
8 # Solução não ideal para o problemas, pois calcula fib(x) em tempo
   exponencial
9 def fib(x):
10     if x == 1 or x == 2:
11         return 1
12     else:
13         return fib(x - 1) + fib(x - 2)
14
15 # Dado inteiro x, retorna o x-ésimo valor da sequência de Fibonacci
16 # Solução melhor, pois calcula fib(x) em tempo constante (linear se
17 # considerarmos o número de multiplicações que vão ocorrer para
   elevar
18 # a e b a x). Foi obtida usando o conceito de fórmula fechada para
19 # recorrências
20 def fib_fechado(x):
21     raiz5 = sqrt(5)
22     a = (1 + raiz5) / 2
23     b = (1 - raiz5) / 2
24     return (a ** x - b ** x) / raiz5
25
26 # Dado inteiro x, retorna o x-ésimo valor da sequência de Fibonacci
27 # Solução definitiva, pois calcula fib(x) em tempo linear, e possui
28 # um uso de memória constante
29 def fib_definitivo(x):
30     a = 1
31     b = 0
32     while x > 1:
33         aux = a
34         a = a + b
35         b = aux
36         x = x - 1
```

```

37     return a
38
39 # Dado um inteiro x, retorna o índice do número de Fibonacci
40 # que possui x dígitos
41 def x_digit_Fibonacci(x):
42     i = 1
43     while num_digitos(fib_definitivo(i)) < x:
44         i = i + 1
45     return i
46
47 res = x_digit_Fibonacci(1000)
48 print(res)

```

A resposta final encontrada foi que o 4782-ésimo número de Fibonacci é o primeiro a possuir 1000 dígitos.

4.2.3 Considerações finais

Este exercício exigiu bastante criatividade na hora de encontrar uma forma eficiente de resolver o problema de Fibonacci, que era o maior gargalo da solução, já que $x_digit_Fibonacci(x)$ e $num_digitos(x)$ possuem tempo de execução da ordem de n e $\log_{10}n$, respectivamente.

4.3 Distinct Powers

4.3.1 Metodologia

O problema consistia em descobrir de quantos elementos diferentes existem numa lista com os resultados das possíveis combinações de a^b , com $2 \leq a \leq 100$ e com $2 \leq b \leq 100$.

4.3.2 Resultados e discussões

```

1 # Dado dois inteiros low, high, calcula o número de resultados
  # diferentes
2 # das possíveis combinações de a ** b (elevado), com a, b
  # pertencentes a [low, high]
3 def num_combinacoes(low, high):
4     lista_combinacoes = []
5     for a in range(low, high + 1):
6         for b in range(low, high + 1):
7             aux = a**b
8             if aux not in lista_combinacoes:
9                 lista_combinacoes.append(aux)
10
11     return len(lista_combinacoes)
12
13 res = num_combinacoes(2, 100)
14 print(res)

```

A resposta encontrada foi 9183, que está de acordo com a presente na lista de soluções.

4.3.3 Considerações finais

Assim como no primeiro exercício deste conjunto, a solução foi bem direto ao ponto, calculando todas as possíveis combinações de a^b , em tempo $O(n^2)$, onde n é o número de elemento no conjunto $[low, high]$; depois, colocando na lista apenas os que ainda não estão presentes nela e contando quantos elementos existem nela no final. A solução esta longe de ser a mais eficiente, mas consegue resolver o problema quase instantâneamente em casos relativamente pequenos, como o pedido.

5 Capítulo IV: Project Euler - Conjunto 4

O quarto conjunto de problemas contém os exercícios: Circular Primes (Primos Circulares), Goldbach's other conjecture (A outra conjectura de Goldbach) e Consecutive Prime Sum (Soma de Primos Consecutivos).

5.1 Circular Primes

5.1.1 Metodologia

Um número é chamado de Primo Circular se as rotações dos seus dígitos também são primos. O problema em questão procura saber quantos números primos circulares existem até um milhão.

5.1.2 Resultados e discussões

A solução começou fazendo uma função que retorna True se um dado número é primo, e False se não. Depois, partimos para fazer uma função que preenche uma lista de tamanho n com True/False dependendo se aquele índice é um número primo ou não. O próximo passo foi descobrir se as rotações com os dígitos de um número são primos ou não. Devido as facilidades de se usar strings em Python, converter o número pra uma string e fazer algumas manipulações de fatiamento tornaram essa tarefa bem simples. Por fim, a resolução final consistem em repetir as verificações para todos os primos obtidos na função *lista_primos*. Infelizmente, a solução é completamente inviável para números muito grandes, como o 1000000 pedido pelo exercício. Então, buscamos por formas mais eficientes de se separar os números primos, e encontramos o Crivo de Eratóstenes, um algoritmo bastante eficiente para encontrar números primos. Felizmente, esse algoritmo fez com que nossa solução resolve num tempo viável, embora encontramos versões que agilizam ainda mais ao nem considerar primos que contém certos dígitos como 2 e 5.

```
1 from math import ceil
2
3 # Dado um inteiro x, retorna True se for primo e False caso contrário
```

```

4 def primo(x):
5     if x == 2:
6         return True
7     elif x % 2 == 0 or x < 2:
8         return False
9     else:
10        divisivel = False
11        i = 2
12        while i <= x/2 and not divisivel:
13            if x % i == 0:
14                divisivel = True
15            i = i + 1
16        return not divisivel
17
18 # Dado um limitante superior x, retorna a lista de tamanho x com
19 # True se o número com aquele índice é primo e False caso contrário
20 def lista_primos(x):
21     lista = []
22     for i in range(0, x):
23         lista.append(primo(i))
24     return lista
25
26 # Algoritmo mais eficiente para gerar uma lista de primos
27 # Dado um inteiro x, retorn uma lista com True se aquele
28 # índice é de um primo, False caso contrário
29 def sieve_eratosthenes(x):
30     arredonda = lambda x, primo: int(ceil(float(x) / primo))
31
32     primos = [True] * x
33     primos[0] = False
34     primos[1] = False
35     lista_primo = []
36
37     for primo_atual in range(2, x):
38         if not primos[primo_atual]:
39             continue
40         lista_primo.append(primo_atual)
41         for m in range(2, arredonda(x, primo_atual)):
42             primos[m * primo_atual] = False
43     return primos
44
45 # Dado um valor primo x, verifica se as rotações dos dígitos
46 # de x estão contidas numa lista booleana de primos
47 def primo_circular(x, lista_de_primos):
48     x = str(x) # convertendo para string para facilitar rotação
49     for i in range(0, len(x)):
50         rotacionado = x[i:len(x)] + x[0:i]
51         if not lista_de_primos[int(rotacionado)]:
52             return False
53     return True
54
55 # Dado um limitante superior inteiro x, retorna quantos primos
56 # circulares existem em [1, x)
57 def quantos_primos_circulares(x):
58     quantidade = 0
59     primos = sieve_eratosthenes(x)
60     for i in range(0, x):

```

```

61         if primo_circular(i, primos):
62             quantidade = quantidade + 1
63         return quantidade
64
65 res = quantos_primos_circulares(1000000)
66 print(res)

```

A resposta encontrada foi 55.

5.1.3 Considerações finais

Esse problema foi bastante complicado, principalmente no momento de obtenção da solução mais eficiente para encontrar números primos. Acabamos aprendendo como usar funções *lambda* em Python, e entendemos melhor como fatiar Strings.

5.2 Goldbach's other conjecture

5.2.1 Metodologia

A solução do problema se iniciou com a reutilização do Crivo de Eratóstenes do exercício anterior, devido a sua grande eficiência em encontrar número primos. Depois, a solução surgiu da ideia de simplesmente testar o primeiro número que não possuía uma raiz inteira que satisfazia a equação $numero = primo + 2 * outro_numero^2$.

5.2.2 Resultados e discussões

```

1 from math import sqrt, ceil
2
3 def sieve_eratosthenes(x):
4     arredonda = lambda x, primo: int(ceil(float(x) / primo))
5
6     primos = [True] * x
7     primos[0] = False
8     primos[1] = False
9     lista_primo = []
10
11     for primo_atual in range(2, x):
12         if not primos[primo_atual]:
13             continue
14         lista_primo.append(primo_atual)
15         for m in range(2, arredonda(x, primo_atual)):
16             primos[m * primo_atual] = False
17     return lista_primo
18
19 # Encontra o primeiro número que não obedece a conjectura
20 # de Goldbach
21 def goldbach_conjecture():
22     lista_primos = sieve_eratosthenes(1000000)
23     num = 1
24     não_encontrou = True
25     while não_encontrou:
26         num += 2
27         i = 0

```

```

28     não_encontrou = False
29     while num >= lista_primos[i]:
30         a = num - lista_primos[i]
31         # quando essa condição não é satisfeita, o número não é
        quadrado perfeito
32         if sqrt(a/2) == int(sqrt(a/2)):
33             não_encontrou = True
34             break
35         i += 1
36     return num
37
38 res = goldbach_conjecture()
39 print(res)

```

A resposta encontrada foi 5777, que condiz com a lista de respostas.

5.2.3 Considerações finais

A solução desse exercício foi bastante direto ao ponto, e exigiu apenas perceber a condição que necessitava que $\sqrt{\frac{(\text{numero} - \text{primo})}{2}}$ tem que ser um quadrado perfeito.

5.3 Consecutive Prime Sum

5.3.1 Metodologia

O problema pedia que, dado um inteiro x , fosse encontrada a maior sequência de números primos consecutivos de forma que a soma seja a maior possível e seja um primo inferior ao dado x . A solução se iniciou reutilizando o Crivo de Eratóstenes, que foi utilizado no primeiro exercício desse conjunto, levemente modificado, de forma a retornar tanto uma lista de primos de 1.. x e a lista de primalidade (True para índices primos, False para compostos). A partir daí, partimos para buscar a solução do problema.

5.3.2 Resultados e discussões

```

1 from math import ceil
2
3 def sieve_eratosthenes(x):
4     arredonda = lambda x, primo: int(ceil(float(x) / primo))
5
6     primos = [True] * x
7     primos[0] = False
8     primos[1] = False
9     lista_primo = []
10
11     for primo_atual in range(2, x):
12         if not primos[primo_atual]:
13             continue
14         lista_primo.append(primo_atual)
15         for m in range(2, arredonda(x, primo_atual)):
16             primos[m * primo_atual] = False
17     return lista_primo, primos
18

```

```

19 # Dado um inteiro x, retorna a maior sequência de soma de primos,
    tal que a soma é inferior a x
20 def maior_soma_primos(x):
21     maior = 0
22     lista_primo, é_primo = sieve_eratosthenes(x)
23     consecutivo = 0
24     for i in range(len(lista_primo)):
25         soma = lista_primo[i]
26         consec = 1
27         for j in range(i + 1, len(lista_primo)):
28             soma += lista_primo[j]
29             consec += 1
30             if soma >= len(é_primo):
31                 break
32             if é_primo[soma] and consec > consecutivo:
33                 maior = soma
34                 consecutivo = consec
35     return maior
36
37 res = maior_soma_primos(1000000)
38 print(res)

```

A resposta encontrada foi 997651.

5.3.3 Considerações finais

A solução deste exercício se mostrou bastante complicada. Depois de muito tempo buscando uma forma eficiente de encontrar a maior sequência que ao mesmo tempo resulta na maior soma, acabamos busca ajuda na internet. Procurando sugestões de algoritmos e até mesmo algumas soluções, fizemos adaptações para seguir o mesmo estilo de soluções anteriores, e acabamos chegando num algoritmo bastante rápido para a solução do problema.

6 Capítulo V: Knight's Tour

6.1 Metodologia

O problema do Passeio do Cavalo, ou Knight's Tour, consiste em, dado um tabuleiro e uma posição inicial de um cavalo, fazer com que se visite todas as outras casas, sem repetição. Deve-se respeitar as regras de movimentação do xadrez. A regra de Warnsdorff é uma heurística bastante utilizada na resolução do problema. Ela diz:

- Inicie de uma posição qualquer do tabuleiro
- Sempre mova para um quadrado adjacente não visitado, isto é que o cavalo pode visitar, que tenha grau mínimo, ou seja, que tenha o menor número de vizinhos não visitados.

Para resolução desse problema, foi sugerido o seguinte algoritmo:

1. Seja P uma posição inicial do tabuleiro
2. Marque P com o movimento número 1

3. Para cada número de movimento de 2 até o número de posições do tabuleiro faça:
 - (a) Seja S o conjunto das posições acessíveis de P
 - (b) Seja P a posição em S com mínima acessibilidade
 - (c) Marque a posição P com o número do movimento atual
4. Retorne a matriz marcada em que cada posição terá o número do movimento

6.2 Resultados e discussões

```
1 import sys, time
2
3 # Classe que gerencia o problema
4 class KnightsTour:
5     def __init__(self, width, height):
6         self.w = width
7         self.h = height
8
9         self.board = []
10        self.generate_board()
11
12    # Função que gera o tabuleiro utilizando self.w e self.h
13    def generate_board(self):
14        for i in range(self.h):
15            self.board.append([0]*self.w)
16
17    # Função opcional para imprimir o tabuleiro
18    def print_board(self):
19        print(" ")
20        print("-----")
21        for elem in self.board:
22            print(elem)
23        print("-----")
24        print(" ")
25
26    # Gera uma lista de possíveis posições, dada uma lista cur_pos
27    # com [x, y]
28    def generate_legal_moves(self, cur_pos):
29        possible_pos = []
30        move_offsets = [(1, 2), (1, -2), (-1, 2), (-1, -2),
31                        (2, 1), (2, -1), (-2, 1), (-2, -1)]
32
33        for move in move_offsets:
34            new_x = cur_pos[0] + move[0]
35            new_y = cur_pos[1] + move[1]
36
37            if (new_x >= self.h):
38                continue
39            elif (new_x < 0):
40                continue
41            elif (new_y >= self.w):
```



```

42         elif (new_y < 0):
43             continue
44         else:
45             possible_pos.append((new_x, new_y))
46
47     return possible_pos
48
49     # Para ser mais eficiente, é mais fácil visitar os vizinhos
50     # sozinho, já que eles ficam nas bordas do tabuleiro, e
51     # talvez não
52     # possam ser alcançados depois
53     def sort_lonely_neighbors(self, to_visit):
54         neighbor_list = self.generate_legal_moves(to_visit)
55         empty_neighbours = []
56
57         for neighbor in neighbor_list:
58             np_value = self.board[ neighbor[0] ][ neighbor[1] ]
59             if np_value == 0:
60                 empty_neighbours.append(neighbor)
61
62         scores = []
63         for empty in empty_neighbours:
64             score = [empty, 0]
65             moves = self.generate_legal_moves(empty)
66             for m in moves:
67                 if self.board[ m[0] ][ m[1] ] == 0:
68                     score[1] += 1
69             scores.append(score)
70
71         scores_sort = sorted(scores, key = lambda s: s[1])
72         sorted_neighbours = [s[0] for s in scores_sort]
73         return sorted_neighbours
74
75     # Função recursiva do Passeio do Cavalo (Knight's Tour)
76     # n: profundidade da árvore de busca
77     # path: caminho atual
78     # to_visit: vértice para visitar
79     def tour(self, n, path, to_visit):
80         self.board[to_visit[0]][to_visit[1]] = n
81         path.append(to_visit) #append the newest vertex to the
82         #current point
83         print("Visitando: ", to_visit)
84
85         if n == self.w * self.h: #if every grid is filled
86             self.print_board()
87             print(path)
88             print("Finalizado!")
89             end = time.time()
90             print("Tempo de execução: ", end - start)
91             sys.exit(1)
92
93         else:
94             sorted_neighbours = self.sort_lonely_neighbors(to_visit)
95
96             for neighbor in sorted_neighbours:

```

```

95         self.tour(n+1, path, neighbor)
96
97         #If we exit this loop, all neighbours failed so we
reset
98         self.board[to_visit[0]][to_visit[1]] = 0
99         try:
100             path.pop()
101             print("Voltando para: ", path[-1])
102         except IndexError:
103             print("Nenhum caminho encontrado")
104             sys.exit(1)
105
106 kt = KnightsTour(8, 8)
107
108 start = time.time()
109 kt.tour(1, [], (0,0))
110 kt.print_board()

```

6.3 Considerações finais

O problema Knight's Tour foi bastante complicado de resolver, principalmente pelo uso de árvores de busca. Tivemos que recorrer a internet para buscar uma solução. Encontramos uma bastante simples e elegante, que encapsula todo o problema numa única classe e em 5 métodos. A solução resolve o problema de forma recursiva, o que por um lado facilita partir da definição do problema e o entendimento dela. Por outro lado, ela consome mais recursos por colocar as chamadas na pilha de execução. A solução encontrada resolve um tabuleiro 8x8 em 0.0032131s, resolve um 16x16 em 0.0397682s e um 31x31 em 0.0807571s. Infelizmente, ao realizar o salto de 31x31 para 32x32, a solução recursiva não consegue resolver o problema, pois estoura a pilha de recursão (RecursionError: maximum recursion depth exceeded while calling a Python object).

7 Referências

- Discrete Mathematics - Wikipédia, consulta em 30 de junho de 2019
- Sieve of Eratosthenes - Wikipédia, consulta em 6 de julho de 2019
- Lista de soluções (cedida pelo professor), última consulta em 7 de julho de 2019
- Site com a solução do Passeio do Cavalo, consulta em 7 de julho de 2019