Aplicações e Problemas Computacionais em Matemática Discreta

Guilherme Locca Salomão, João Victor Mendes Freire
e Renan Dantas Pasquantonio

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 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 bastante importante quando se trata 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 eficiênte.
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.
Em seguida, temos uma seção com os códigos de resolução. E, por fim, as considerações finais, onde explicamos eventuais dificuldades e algumas análises de tempo e espaço dos algoritmos.

1.5 Materiais e Métodos

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

```python
primo_atual = 3
primos = [2]
eh_primo = 1  # flag que verifica se o numero é primo ou não
while( len(primos) != 10001): # loop ocorre enquanto não a lista de numeros primos é diferente a 10001
    eh_primo = 1
    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
    for n in primos: # utilizando todos os primos anteriores
        if primo_atual%n == 0: # verificamos se o numero a ser analizado no momento não é divisivel por nenhum deles o que implicara que ele so tem um unico divisor alem de 1
            eh_primo = 0
    if eh_primo == 1: # passando nos testes o numero é adicionado a lista
        primos.append(primo_atual)
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 possivel deixa-lo instantâneo.

2.2 Special Pyhtagorian Triplet

2.2.1 Metodologia

Uma tripla pitagorica é quando a soma dos quadrados de dois numeros é igual ao quadrado de um terceiro número, ou seja $a^2 + b^2 = c^2$. Para resolver esse problema foi usada a logica 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

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

A saida do programa foi 31875000, compativel com a resposta no arquivo fornecido.

2.2.3 Considerções finais

A solução desse problema foi difícil de se traduzir para codigo poís há varias soluções possiveis porém cada uma tem sua dificuldade de implementação, entretanto, a "ideia" da solução foi facil de se chegar.

2.3 Longest Collatz Sequence

2.3.1 Metodologia

Uma sequencia collatz é uma sequêcia iterativa onde considerando a sequência iniciando em um numero n, se n for par então $n = \frac{n}{2}$, se n for impar 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

```python
start = 1 # começando pelo caso inicial
longest_number = 1 # Caso inicial da mais longa
longest_steps = 1 # Quantos passos ocorreram na mais longa
cur_steps = 0 # contador de passos
cur_number = 0 # numero inicial atual
while(start < 1000000): # verificando todos os casos ate 1000000
    cur_steps = 0
    start = start + 2 # verifica apenas os numeros impares pois esse tem as sequencias mais longas
    cur_number = start
    while (cur_number != 1): # loop ocorre enquanto não atingir o caso base
        if cur_number%2 == 0:
            cur_number = cur_number/2
            cur_steps = cur_steps+1
        else:
            cur_number = (3*cur_number)+1
            cur_steps = cur_steps+1
    if cur_steps > longest_steps: # atualiza se a quantidade de 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 porem achar detalhes que a otimizassem 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 exercicio é direta e simples, achar todos os primos e então soma-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

```
def achaPrimos(limite): # Função que ira procurar todos os primos menores que um certo limite
    primos=[True for n in range(limite+1) ] # cria uma lista com n=limite elementos e os atribui o valor logico True
    crivo=2 # cria uma chave do primeiro número primo
    saida = [] # lista de saida
    while  crivo*crivo <= limite: # enquanto o quadrado da cahve for menor que o limite, o loop ocorre
        if(primos[crivo]==True): # verifica se a chave é  um primo
            for i in range(crivo*2,limite+1,crivo): # procura todos os números multiplos da chave
                primos[i]=False # os transforma em falso, ou seja, não primos
        crivo += 1 # aumenta a chave
    for i in range(2,limite): # Traduz o indice dos elemento primos para uma lista de elementos
        if primos[i]==True:
            saida.append(i)
    return saida

primos = achaPrimos(2000000)
soma = 0
for i in primos: # realiza a soma de todos os elementos
    soma += i
print(soma)
```

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

3.1.3 Conclusões finais

Para realizar esse programa, era necessário um algortimo 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

```c
#include<stdlib.h>
#include<stdio.h>

int achaNumeroTriangular(int pos, int num_anterior); //Função que encontra o número de posição N utilizando o numero da posição N-1
int achaQtdDivisores(int num); //Função que acha o numero de divisores de um número


int main(){
    int divisores=0; //contador de divisores
    int pos_atual = 0; //posição atual a ser considerada
    int num_atual=0; //Numero triangular da posição atual
    while (divisores <500) // Realiza a operação ate o contador de divisores atinga 500 ou mais
    {

    num_atual = achaNumeroTriangular(pos_atual,num_atual);
    divisores = achaQtdDivisores(num_atual);

    pos_atual++;
    }
    printf("\n resultado: %d\n",num_atual);
}

int achaNumeroTriangular(int pos, int num_anterior){
    return pos+num_anterior;
}

int achaQtdDivisores(int num){
    int cont=0;
    for(int i=1;i<=num;i++){
    if(num%i==0){ //verifica se o resto da divisão é zero o que o caracteriza como divisor
        cont++;
    }
    }
    return cont;
}
```

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

3.2.3 Considerações finais

A soluçao desse problema era bem direta e facil de se implementar, o que tornou viavel 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 operção 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

```python
x = 2**1000 # numero a ser usado
div = 10 # potencia de 10 inicial
aux = 0
aux2 = 1
soma = 0 # soma final dos digitos
while(x>0): # realiza o loop até que o numero inicial seja 0
    aux = x%div # armazena o digito a ser recebido nessa iteração
    soma = soma+ (aux/aux2) # soma o digito a soma total, é usado um auxiliar para garantir que o numero seja unitário
    x = x - aux # remove o resto do número
    aux2 =div # atualiza as potencias para poder continuar a operação
    div = div*10
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},ja que esse requer cerca de 1000 bits para armazenar, então, ao contrario da 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 on 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

```python
# Dado um número inteiro x, a função retorna a soma dos divisores inteiros de x
def soma_divisores(x):
    soma = 0;
    for i in range(1, x // 2 + 1):
        if x % i == 0:
            soma = soma + i
    return soma

# Dado um número inteiro x, a função retorna a soma dos números amigáveis no intervalo [1, x)
def soma_amigaveis(x):
    soma = 0
    a = 1
    while a < x:
        b = soma_divisores(a)
        # se b > a, para que não se calcule um número amigável que já apareceu
        if b > a and soma_divisores(b) == a:
            soma = soma + a + b
        a = a + 1
    return soma

res = soma_amigaveis(10000)
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 bastante objetiva: 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 estavamos 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 digitos tem um número, e rapidamente chegamos a conclusão que se dividissemos o número várias vezes por 10 até que sobrasse apenas 0, teriamos o número de casas. Depois, partimos para a solução que, diferentemente do problema anterior, teve que ser alterada por razões de desempenho.

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 digitos.
Tento 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 cressem 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.

```
from math import sqrt, floor, log10

# Dado um inteiro x, calcula o número de digitos
def num_digitos(x):
    return floor(1 + log10(x))

# Dado inteiro x, returna o x-ésimo valor da sequência de Fibonacci
# Solução não ideal para o problemas, pois calcula fib(x) em tempo
    exponencial
def fib(x):
    if x == 1 or x == 2:
        return 1
```

```python
    else:
        return fib(x - 1) + fib(x - 2)

# Dado inteiro x, returna o x-ésimo valor da sequência de Fibonacci
# Solução melhor, pois calcula fib(x) em tempo constante (linear se
# considerarmos o número de multiplicações que vão ocorrer para elevar
# a e b a x). Foi obtida usando o conceito de fórmula fechada para
# recorrências
def fib_fechado(x):
    raiz5 = sqrt(5)
    a = (1 + raiz5) / 2
    b = (1 - raiz5) / 2
    return (a ** x - b ** x) / raiz5

# Dado inteiro x, returna o x-ésimo valor da sequência de Fibonacci
# Solução definitiva, pois calcula fib(x) em tempo linear, e possui
# um uso de memória constante
def fib_definitivo(x):
    a = 1
    b = 0
    while x > 1:
        aux = a
        a = a + b
        b = aux
        x = x - 1
    return a

# Dado um inteiro x, retorna o índice do número de Fibonacci
# que possui x digitos
def x_digit_Fibonacci(x):
    i = 1
    while num_digitos(fib_definitivo(i)) < x:
        i = i + 1
    return i

res = x_digit_Fibonacci(1000)
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 eficiênte 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

```
# Dado dois inteiros low, high, calcula o número de resultados
    diferentes
# das possíveis combinações de a ** b (elevado), com a, b
    pertencentes a [low, high]
def num_combinacoes(low, high):
    lista_combinacoes = []
    for a in range(low, high + 1):
        for b in range(low, high + 1):
            aux = a**b
            if aux not in lista_combinacoes:
                lista_combinacoes.append(aux)

    return len(lista_combinacoes)

res = num_combinacoes(2, 100)
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 digitos 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 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.

```python
from math import ceil

# Dado um inteiro x, retorna True se for primo e False caso contrário
def primo(x):
    if x == 2:
        return True
    elif x % 2 == 0 or x < 2:
        return False
    else:
        divisivel = False
        i = 2
        while i <= x/2 and not divisivel:
            if x % i == 0:
                divisivel = True
            i = i + 1
    return not divisivel

```

```python
# Dado um limitante superior x, retorna a lista de tamanho x com
# True se o número com aquele índice é primo e False caso contrário
def lista_primos(x):
    lista = []
    for i in range(0, x):
        lista.append(primo(i))
    return lista

# Algotimo mais eficiente para gerar uma lista de primos
# Dado um inteiro x, retorn uma lista com True se aquele
# índice é de um primo, False caso contrário
def sieve_eratosthenes(x):
    arredonda = lambda x, primo: int(ceil(float(x) / primo))

    primos = [True] * x
    primos[0] = False
    primos[1] = False
    lista_primo = []

    for primo_atual in range(2, x):
        if not primos[primo_atual]:
            continue
        lista_primo.append(primo_atual)
        for m in range(2, arredonda(x, primo_atual)):
            primos[m * primo_atual] = False
    return primos

# Dado um valor primo x, verifica se as rotações dos dígitos
# de x estão contidas numa lista booleana de primos
def primo_circular(x, lista_de_primos):
    x = str(x) # convertendo para string para facilitar rotação
    for i in range(0, len(x)):
        rotacionado = x[i:len(x)] + x[0:i]
        if not lista_de_primos[int(rotacionado)]:
            return False
    return True

# Dado um limitante superior inteiro x, retorna quantos primos
# circulares existem em [1, x)
def quantos_primos_circulares(x):
    quantidade = 0
    primos = sieve_eratosthenes(x)
    for i in range(0, x):
        if primo_circular(i, primos):
            quantidade = quantidade + 1
    return quantidade

res = quantos_primos_circulares(1000000)
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 possuia uma raíz inteira que satisfazia a equação $numero = primo + 2 * outro\_numero^2$.

5.2.2 Resultados e discussões

```python
from math import sqrt, ceil

def sieve_eratosthenes(x):
    arredonda = lambda x, primo: int(ceil(float(x) / primo))

    primos = [True] * x
    primos[0] = False
    primos[1] = False
    lista_primo = []

    for primo_atual in range(2, x):
        if not primos[primo_atual]:
            continue
        lista_primo.append(primo_atual)
        for m in range(2, arredonda(x, primo_atual)):
            primos[m * primo_atual] = False
    return lista_primo

# Encontra o primeiro número que não obedesce a conjectura
# de Goldbach
def goldbach_conjecture():
    lista_primos = sieve_eratosthenes(1000000)
    num = 1
    não_encontrou = True
    while não_encontrou:
        num += 2
        i = 0
        não_encontrou = False
        while num >= lista_primos[i]:
            a = num - lista_primos[i]
            # quando essa condição não é satisfeita, o número não é quadrado perfeito
            if sqrt(a/2) == int(sqrt(a/2)):
                não_encontrou = True
                break
            i += 1
    return num

res = goldbach_conjecture()
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 $\sqrt{\frac{(numero-primo)}{2}}$ 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 indices primos, False para compostos).
A partir daí, partimos para buscar a solução do problema.

5.3.2 Resultados e discussões

```python
from math import ceil

def sieve_eratosthenes(x):
    arredonda = lambda x, primo: int(ceil(float(x) / primo))

    primos = [True] * x
    primos[0] = False
    primos[1] = False
    lista_primo = []

    for primo_atual in range(2, x):
        if not primos[primo_atual]:
            continue
        lista_primo.append(primo_atual)
        for m in range(2, arredonda(x, primo_atual)):
            primos[m * primo_atual] = False
    return lista_primo, primos

# Dado um inteiro x, retorna a maior sequência de soma de primos,
    tal que a soma é inferior a x
def maior_soma_primos(x):
    maior = 0
    lista_primo, é_primo = sieve_eratosthenes(x)
    consecutivo = 0
    for i in range(len(lista_primo)):
        soma = lista_primo[i]
        consec = 1
        for j in range(i + 1, len(lista_primo)):
            soma += lista_primo[j]
            consec += 1
```

```
            if soma >= len(é_primo):
                break
            if é_primo[soma] and consec > consecutivo:
                maior = soma
                consecutivo = consec
    return maior

res = maior_soma_primos(1000000)
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 sequencia 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 heurisica 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

```
import sys, time

# Classe que gerencia o problema
class KnightsTour:
    def __init__(self, width, height):
        self.w = width
        self.h = height

        self.board = []
        self.generate_board()

    # Função que gera o tabuleiro utilizando self.w e self.h
    def generate_board(self):
        for i in range(self.h):
            self.board.append([0]*self.w)

    # Função opcional para imprimir o tabuleiro
    def print_board(self):
```

```python
        print("  ")
        print("------")
        for elem in self.board:
            print(elem)
        print("------")
        print("  ")

    # Gera uma lista de possiveis posições, dada uma lista cur_pos com [x, y]
    def generate_legal_moves(self, cur_pos):
        possible_pos = []
        move_offsets = [(1, 2), (1, -2), (-1, 2), (-1, -2),
                        (2, 1), (2, -1), (-2, 1), (-2, -1)]

        for move in move_offsets:
            new_x = cur_pos[0] + move[0]
            new_y = cur_pos[1] + move[1]

            if (new_x >= self.h):
                continue
            elif (new_x < 0):
                continue
            elif (new_y >= self.w):
                continue
            elif (new_y < 0):
                continue
            else:
                possible_pos.append((new_x, new_y))

        return possible_pos

    # Para ser mais eficiente, é mais fácil visitar os vizinhos sozinhos
    # no começo, já que eles ficam nas bordas do tabuleiro, e talvez não
    # possam ser alcançados depois
    def sort_lonely_neighbors(self, to_visit):
        neighbor_list = self.generate_legal_moves(to_visit)
        empty_neighbours = []

        for neighbor in neighbor_list:
            np_value = self.board[ neighbor[0] ][ neighbor[1] ]
            if np_value == 0:
                empty_neighbours.append(neighbor)

        scores = []
        for empty in empty_neighbours:
            score = [empty, 0]
            moves = self.generate_legal_moves(empty)
            for m in moves:
                if self.board[ m[0] ][ m[1] ] == 0:
                    score[1] += 1
            scores.append(score)

        scores_sort = sorted(scores, key = lambda s: s[1])
        sorted_neighbours = [s[0] for s in scores_sort]
        return sorted_neighbours
```

```python


    # Função recursiva do Passeio do Cavalo (Knight's Tour)
    # n: profundidade da árvore de busca
    # path: caminho atual
    # to_visit: vértice para visitar
    def tour(self, n, path, to_visit):
        self.board[to_visit[0]][to_visit[1]] = n
        path.append(to_visit) #append the newest vertex to the current point
        print("Visitando: ", to_visit)

        if n == self.w * self.h: #if every grid is filled
            self.print_board()
            print(path)
            print("Finalizado!")
            end = time.time()
            print("Tempo de execução: ", end - start)
            sys.exit(1)

        else:
            sorted_neighbours = self.sort_lonely_neighbors(to_visit)
            for neighbor in sorted_neighbours:
                self.tour(n+1, path, neighbor)

            #If we exit this loop, all neighbours failed so we reset
            self.board[to_visit[0]][to_visit[1]] = 0
            try:
                path.pop()
                print("Voltando para: ", path[-1])
            except IndexError:
                print("Nenhum caminho encontrado")
                sys.exit(1)

kt = KnightsTour(8, 8)

start = time.time()
kt.tour(1, [], (0,0))
kt.print_board()
```

6.3 Considerações finais

O problema Knight's Tour foi bastante complicado de resolver, principalemente pelo uso de árvores de busca. Tivemos que recorer 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.

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