Laços encaixados e uma introdução à sua otimização
MC102-2018s1-Aula10-180403

Arthur J. Catto, PhD
ajcatto@g.unicamp.br

05 de abril de 2018

1 Laços encaixados e uma introdução à sua otimização

1.1 Exemplo: *Dada uma sequência arbitrária de inteiros não-negativos, encontrar o par mais próximo*

Ler uma sequência arbitrária S de n inteiros não-negativos e ***depois*** encontrar $i, 0 \leq i < n$ e $j, 0 \leq j < n, i \neq j$, tais que a distância entre S_i e S_j, isto é, $|S_i - S_j|$, seja mínima.

1.1.1 Desenvolvimento da solução

Em nível mais abstrato, a solução deste problema tem estrutura semelhante à de vários exemplos anteriores:

```
ler a sequência S
encontrar os dois elementos mais próximos em S
exibir a solução
```

Para ``ler a sequência S'' temos pelo menos duas saídas já conhecidas:

```
In [2]: n = int(input('Número de elementos? '))
        S = []
        for i in range(n):
            S.append(int(input('Elemento #' + str(i + 1) + '? ')))
        print(S)
```

```
[12, 23, 34, 45, 56]
```

```
In [3]: S = [int(x) for x in input('Sequência de inteiros? ').split()]
        print(S)
```

```
[12, 23, 34, 45, 56]
```

No entanto, em problemas como este, em geral vamos querer trabalhar com sequências mais longas.
Isso inviabiliza a digitação de valores, especialmente quando se lembra que a execução poderá ter que ser repetida diversas vezes enquanto depuramos o algoritmo.

Uma saída mais apropriada é gerar uma sequência de valores psudo-aleatórios usando o módulo *random*.

```
In [4]: from random import choices

        n = 10
        S = choices(range(1000000), k=n)
        print(S[:10])
        print(S[-10:])
```

```
[500815, 694131, 372196, 892086, 397934, 4934, 813883, 901277, 767670, 261677]
[500815, 694131, 372196, 892086, 397934, 4934, 813883, 901277, 767670, 261677]
```

Sequências de números pseudo-aleatórios são geradas a partir de uma *semente*. Se usarmos uma mesma *semente* seremos capazes de repetir uma sequência quantas vezes quisermos.

Para gerar uma sequência "imprevisível", em geral usamos o valor do relógio da máquina como *semente*.

Se salvarmos o valor usado como *semente*, a sequência poderá ser repetida. Por exemplo,...

```
In [5]: import time
        import random
```

```
In [6]: t_inicial = time.time()
        t_inicial
```

```
Out[6]: 1522894266.992621
```

```
In [7]: random.seed(t_inicial)

        S1 = random.choices(range(1000000), k=10)
        print(S1)
```

```
[683022, 240020, 532725, 677507, 228121, 376049, 469446, 138107, 901865, 869104]
```

```
In [8]: S2 = random.choices(range(1000000), k=10)
        print(S2)
```

```
[66377, 243989, 165177, 620341, 80404, 718506, 632689, 508072, 364610, 835636]
```

```
In [9]: S1 == S2
```

```
Out[9]: False
```

Neste caso, geramos `S1` e depois `S2`, sem "re-semear" o gerador de aleatórios. Por causa disso, `S1` e `S2` são diferentes.

No entanto, quando "re-semeamos" o gerador, obtemos `S3` igual a `S1`.

```
In [10]: random.seed(t_inicial)

         S3 = random.choices(range(1000000), k=10)
         print(S3)
```

```
[683022, 240020, 532725, 677507, 228121, 376049, 469446, 138107, 901865, 869104]
```

```
In [11]: S1 == S3
```

```
Out[11]: True
```

Com isso, nossa geração de dados fica assim...

```
In [12]: import time
         import random
```

```
In [13]: t_inicial = time.time()
```

```
In [14]: random.seed(t_inicial)

         n = 10
         S = random.choices(range(1000000), k=n)
```

Uma vez obtida a sequência, vamos procurar o par mais próximo. Para isso temos que comparar todos os pares possíveis, isto é, o problema pede uma solução por enumeração exaustiva.

Aproveitando exemplos anteriores, podemos esboçar nossa solução como:

```
for i in range(n):
    for j in range(n):
        se i diferente de j e distância entre S[i] e S[j] menor que a menor já observada:
            salvar i, j, distância entre S[i] e S[j]
```

Vamos representar esse esboço em Python?

```
In [15]: min_dist = 10e10
         for i in range(n):
             for j in range(n):
                 if i != j and abs(S[i] - S[j]) < min_dist:
                     min_i, min_j, min_dist = i, j, abs(S[i] - S[j])
```

Agora só falta exibir o resultado...

```
In [16]: print(f'{n:7}  {min_i:6}  {S[min_i]:6}  {min_j:6}  {S[min_j]:6}  {min_dist:6}')

     10       1  266874       3  282097   15223
```

E se quisermos testar nosso programa com diferentes sequências?

- Podemos, por exemplo, gerar uma longa sequência S e depois extrair dela subsequências a serem estudadas.

```
In [17]: from random import choices

         S = choices(range(100000000), k=10000000)
         print(S[:10])
         print(S[-10:])

[87431599, 80651501, 52357837, 77625915, 85708491, 98761912, 88616489, 32882510, 57082936, 97
[37269349, 89051813, 34438878, 71097816, 72041190, 23154799, 11812074, 46524083, 17160639, 36
```

Para poder analisar o desempenho do nosso algoritmo para sequências de diversos tamanhos, vamos medir o tempo gasto usando a função `perf_counter` do módulo `time`.

```
In [18]: from time import perf_counter

         print(f"{'n':>7}  {'demora':>10}  {'i':>8}  {'S[i]':>8}  {'j':>8}  {'S[j]':>8}  {'di
         for n in [10, 100, 1000, 10000]:
             start = perf_counter()
             min_dist = 10e10
             for i in range(n):
                 for j in range(n):
                     if i != j and abs(S[i] - S[j]) < min_dist:
                         min_i, min_j, min_dist = i, j, abs(S[i] - S[j])
             end = perf_counter()
             print(f'{n:7}  {end - start:10.5f}  {min_i:8}  {S[min_i]:8}  {min_j:8}  {S[min_j
```

n	demora	i	S[i]	j	S[j]	dist
10	0.00003	0	87431599	6	88616489	1184890
100	0.00245	47	1423628	53	1420597	3031
1000	0.24583	167	21009646	891	21009553	93
10000	25.34181	855	15585938	6203	15585940	2

Você consegue notar algum padrão no tempo gasto pelo algoritmo?

- Sim, quando o tamanho da amostra cresce 10 vezes, o tempo gasto cresce 100!

O que você espera que vá acontecer se tivermos 100.000 elementos para examinar?

- A execução vai demorar cerca de 40 minutos!

Será possível melhorar esse desempenho?

- Como tanto i quanto j assumem valores em `range(n)` estamos calculando não só $|S[i] - S[j]|$ mas também $|S[j] - S[i]|$.
  Deve ser possível eliminar o teste redundante. Você consegue fazer isso?

```
In [21]: from time import perf_counter

         print(f"{'n':>7}  {'demora':>10}  {'i':>8}  {'S[i]':>8}  {'j':>8}  {'S[j]':>8}  {'di

         for n in [10, 100, 1000, 10000]:
             start = perf_counter()

             min_dist = 10e10
             for i in range(n):
                 for j in range(n):
                     if i != j and abs(S[i] - S[j]) < min_dist:
                         min_i, min_j, min_dist = i, j, abs(S[i] - S[j])

             end = perf_counter()
             print(f'{n:7}  {end - start:10.5f}  {min_i:8}  {S[min_i]:8}  {min_j:8}  {S[min_j
```

```
      n      demora          i      S[i]          j      S[j]       dist
     10     0.00005          0  87431599          6  88616489    1184890
    100     0.00411         47   1423628         53   1420597       3031
   1000     0.28050        167  21009646        891  21009553         93
  10000    27.01262        855  15585938       6203  15585940          2
```

Solução

```
In [20]: from time import perf_counter

         print(f"{'n':>7}  {'demora':>10}  {'i':>8}  {'S[i]':>8}  {'j':>8}  {'S[j]':>8}  {'di

         for n in [10, 100, 1000, 10000]:
             start = perf_counter()

             min_dist = 10e10
             for i in range(n):
                 for j in range(i + 1, n):
                     if i != j and abs(S[i] - S[j]) < min_dist:
                         min_i, min_j, min_dist = i, j, abs(S[i] - S[j])

             end = perf_counter()
             print(f'{n:7}  {end - start:10.5f}  {min_i:8}  {S[min_i]:8}  {min_j:8}  {S[min_j
```

```
      n      demora          i      S[i]          j      S[j]       dist
     10     0.00002          0  87431599          6  88616489    1184890
    100     0.00150         47   1423628         53   1420597       3031
   1000     0.14854        167  21009646        891  21009553         93
  10000    12.94000        855  15585938       6203  15585940          2
```

Melhorou, mas, mesmo assim, quando o tamanho da amostra cresce 10 vezes, o tempo gasto continua crescendo 100.
De uma maneira mais formal, dizemos que a complexidade desse algoritmo é da ordem de n^2 e representamos essa relação como $\mathcal{O}(n^2)$.
Esse comportamento pode inviabilizar o uso desta solução para grandes amostras.
Você consegue identificar onde está essa demora e por que isso acontece?

- O problema é que temos dois *loops* aninhados percorrendo a amostra. Para cada valor do *loop* externo, o *loop* interno faz uma varredura completa, isto é, examina da ordem de n valores. O *loop* externo também examina n valores.
  Portanto, para uma amostra de tamanho n, os dois *loops* combinados realizam da ordem de n^2 operações, o que explica o comportamento do algoritmo.

Não importa o que a gente faça, se continuarmos com dois *loops* aninhados como esses, o tempo gasto na solução será da ordem de n^2.
Para alterar esse comportamento, temos que mudar a nossa abordagem.
Você consegue pensar em algum caso particular no qual a solução do problema seria mais rápida?

- Por exemplo, se a lista estivesse ordenada, isso nos ajudaria?
  Sim, porque aí o par mais próximo seria sempre composto por dois elementos adjacentes.

- E daí?
  Daí, nós poderíamos dispensar o *loop* interno.

Como a lista pode não estar ordenada, vamos ordená-la usando uma função disponível em Python.
Um bom algoritmo de ordenação de listas tem complexidade $\mathcal{O}(n \cdot \log_2 n)$, o que é muito melhor do que $\mathcal{O}(n^2)$.
Você consegue incorporar essas alterações à nossa solução?

In [22]:
```
from time import perf_counter

print(f"{'n':>7}  {'tempo':>10}  {'i':>6}  {'S[i]':>6}  {'j':>6}  {'S[j]':>6}  {'dis
for n in [10, 100, 1000, 10000]:
    start = perf_counter()
    min_dist = 10e10
    for i in range(n):
        for j in range(i + 1, n):
            if i != j and abs(S[i] - S[j]) < min_dist:
                min_i, min_j, min_dist = i, j, abs(S[i] - S[j])
    end = perf_counter()
    print(f'{n:7}  {end - start:10.5f}  {min_i:6}  {S[min_i]:6}  {min_j:6}  {S[min_j
```

```
      n       tempo       i    S[i]       j    S[j]    dist
     10     0.00004       0  87431599       6  88616489  1184890
    100     0.00216      47  1423628      53  1420597    3031
   1000     0.15759     167  21009646     891  21009553       93
  10000    13.16114     855  15585938    6203  15585940        2
```

1.1.2 Solução

In [23]:
```
from time import perf_counter

print(f"{'n':>8}  {'tempo':>10}  {'i':>8}  {'S[i]':>8}  {'j':>8}  {'S[j]':>8}  {'dis
for n in [10, 100, 1000, 10000, 100000, 1000000, 10000000]:
    start = perf_counter()
    Sord = sorted(S[:n])
    min_dist = 10e10
    for i in range(n - 1):
        if abs(Sord[i] - Sord[i + 1]) < min_dist:
            min_i, min_dist = i, abs(Sord[i] - Sord[i + 1])
    end = perf_counter()
    print(f'{n:8}  {end - start:10.5f}  {min_i:8}  {Sord[min_i]:8}  {min_i + 1:8}  {
```

```
       n       tempo         i      S[i]         j      S[j]      dist
      10     0.00001         6  87431599         7  88616489   1184890
     100     0.00004         1   1420597         2   1423628      3031
    1000     0.00046       226  21009553       227  21009646        93
   10000     0.00539      1565  15585938      1566  15585940         2
  100000     0.07483      2767   2713006      2768   2713006         0
 1000000     1.00394        93     10888        94     10888         0
10000000    15.02493        28       258        29       258         0
```

Nosso algoritmo agora tem um bom desempenho, mesmo para grandes valores de n.

1.2 Exemplo: *Gerar todos os números primos menores que um valor dado*

Ler um inteiro n e criar uma lista P com todos os números primos menores do que n.

1.2.1 Desenvolvimento da solução

Este problema também pode ser resolvido por *enumeração exaustiva*. Vamos tentar?

```
In [ ]: # ler n
```

```
In [ ]: P = []
        for k in range(2, n):
            # verificar se k é primo
            # se k é primo: adicionar k à lista P
```

```
In [ ]: print(P)
```

Feito o primeiro esboço, podemos começar a detalhá-lo...

```
In [24]: # ler n
         n = int(input('Primos até quanto? '))
```

```
In [25]: P = []
         for k in range(2, n):
             # testar se k é primo
             k_eh_primo = True
             d = 2
             while k_eh_primo and d < k:
                 if k % d == 0:
                     k_eh_primo = False
                 else:
                     d += 1
             # se k é primo: adicionar k à lista primos
             if k_eh_primo:
                 P += [k]
```

```
In [26]: print(P)

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Lembrando do exemplo anterior...
Será que esse algoritmo funciona bem para qualquer tamanho de amostra?
Você consegue antecipar uma resposta antes de testá-lo?
Por que?

```
In [27]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

         for n in [10, 100, 1000, 10000]:
             start = perf_counter()
```

```
    P = []
    for k in range(2, n):
        # testar se k é primo
        k_eh_primo = True
        d = 2
        while k_eh_primo and d < k:
            if k % d == 0:
                k_eh_primo = False
            else:
                d += 1

        # se k é primo: adicionar k à lista primos
        if k_eh_primo:
            P += [k]

    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]          P[-4:]
      10     0.00001         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00024        25  [2, 3, 5, 7]  [79, 83, 89, 97]
    1000     0.01881       168  [2, 3, 5, 7]  [977, 983, 991, 997]
   10000     0.91895      1229  [2, 3, 5, 7]  [9941, 9949, 9967, 9973]
```

Com os dois *loops* aninhados, o tempo gasto por este algoritmo para examinar n candiadatos é da ordem de n^2, o que o torna impraticável para grandes valores de n.

Podemos melhorar esse desempenho, se nos lembrarmos de que:

- O único primo par é 2 e, portanto, podemos examinar apenas candidatos ímpares.
- Como os candidatos serão ímpares, não faz sentido tentar dividi-los por números pares, o que também reduz o número de divisores à metade.

Vamos incorporar essas alterações ao nosso algoritmo.

```
In [28]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

         for n in [10, 100, 1000, 10000, 100000]:

             start = perf_counter()

             P = [2]
             for k in range(3, n, 2):
                 # testar se k é primo
                 k_eh_primo = True
                 d = 3
                 while k_eh_primo and d < k:
                     if k % d == 0:
```

```
                k_eh_primo = False
            else:
                d += 2

        # se k é primo: adicionar k à lista primos
        if k_eh_primo:
            P += [k]

    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]         P[-4:]
      10     0.00002         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00013        25  [2, 3, 5, 7]  [79, 83, 89, 97]
    1000     0.00790       168  [2, 3, 5, 7]  [977, 983, 991, 997]
   10000     0.47398      1229  [2, 3, 5, 7]  [9941, 9949, 9967, 9973]
  100000    35.98243      9592  [2, 3, 5, 7]  [99961, 99971, 99989, 99991]
```

O desempenho melhorou, mas mudou a sua relação com n?
Não, sua complexidade continua sendo $\mathcal{O}(n^2)$.
Por que será?
Você consegue pensar em alguma melhoria simples?
Você está satisfeito com os limites dos *loops*?

- Não parece ser possível alterar o *loop* externo.
- Mas é possível limitar a range do *loop* interno a $\sqrt{n}$.
  Por que?

```
In [29]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

         for n in [10, 100, 1000, 10000, 100000, 1000000]:

             start = perf_counter()

             P = [2]
             for k in range(3, n, 2):
                 # testar se k é primo
                 k_eh_primo = True
                 while k_eh_primo and d <= int(k**0.5):
                     if k % d == 0:
                         k_eh_primo = False
                     else:
                         d += 2

                 # se k for primo, adicionar k à lista primos
                 if k_eh_primo:
                     P += [k]
```

```
    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]          P[-4:]
      10     0.00011         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00004        46  [2, 3, 5, 7]  [93, 95, 97, 99]
    1000     0.00031       489  [2, 3, 5, 7]  [993, 995, 997, 999]
   10000     0.00279      4966  [2, 3, 5, 7]  [9993, 9995, 9997, 9999]
  100000     0.03011     49892  [2, 3, 5, 7]  [99993, 99995, 99997, 99999]
 1000000     0.21948    499658  [2, 3, 5, 7]  [999993, 999995, 999997, 999999]
```

O desempenho já melhorou bastante e talvez seja suficiente para um grande número de aplicações.
Mas, e se quiséssemos algo mais rápido?

Podemos tentar outra abordagem. Por exemplo, a estratégia do *crivo de Eratóstenes*:

1. Criamos uma lista com todos os candidatos possíveis.
2. Percorremos a lista da esquerda para a direita e, para cada candidato ainda não riscado, riscamos todos os seus múltiplos.
3. Ao chegar ao final, todos os candidatos não riscados serão primos.

Para representar o *crivo* e os candidatos *riscados* ou não, vamos usar uma lista de *bools*.

- `crivo[i] = True` representará *i não está riscado*
- `crivo[i] = False` representará *i está riscado*

Inicialmente, todos os itens do *crivo* serão `True` e esse valor será convertido para `False` se (e quando) o item for riscado.

Vamos ver como fica...

In [30]:
```
from time import perf_counter

print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

for n in [10, 100, 1000, 10000, 100000, 1000000]:

    start = perf_counter()

    crivo = [True] * n
    for k in range(3, int(n**0.5) + 1, 2):
        if crivo[k]:
            for i in range(k*k, n, 2*k):
                crivo[i] = False

    P = [2]
    for i in range(3, n, 2):
        if crivo[i]:
            P += [i]

    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]         P[-4:]
      10     0.02260         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00002        25  [2, 3, 5, 7]  [79, 83, 89, 97]
    1000     0.00016       168  [2, 3, 5, 7]  [977, 983, 991, 997]
   10000     0.00105      1229  [2, 3, 5, 7]  [9941, 9949, 9967, 9973]
  100000     0.01105      9592  [2, 3, 5, 7]  [99961, 99971, 99989, 99991]
 1000000     0.12093     78498  [2, 3, 5, 7]  [999959, 999961, 999979, 999983]
```

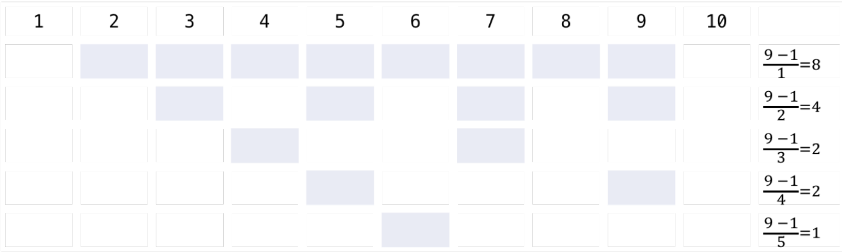
É possível transformar a iteração das linhas 15-18 em uma *list comprehension*:

```
In [31]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

         for n in [10, 100, 1000, 10000, 100000, 1000000]:

             start = perf_counter()

             crivo = [True] * n
             for k in range(3, int(n**0.5) + 1, 2):
                 if crivo[k]:
                     for i in range(k*k, n, 2*k):
                         crivo[i] = False
             P = [2] + [i for i in range(3, n, 2) if crivo[i]]

             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]         P[-4:]
      10     0.00234         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00002        25  [2, 3, 5, 7]  [79, 83, 89, 97]
    1000     0.00009       168  [2, 3, 5, 7]  [977, 983, 991, 997]
   10000     0.00090      1229  [2, 3, 5, 7]  [9941, 9949, 9967, 9973]
  100000     0.01182      9592  [2, 3, 5, 7]  [99961, 99971, 99989, 99991]
 1000000     0.12185     78498  [2, 3, 5, 7]  [999959, 999961, 999979, 999983]
```

Finalmente, podemos acelerar nosso algoritmo ainda mais substituindo o *loop* das linhas 12-13 por outra *list comprehension*...

- Os elementos selecionados pela `range` da linha 11 são `crivo[k*k]`, `crivo[k*k + 2*k]`, ...
- Numa faixa `[esq...dir)` há `(dir - 1 - esq) // larg` intervalos de largura `larg`.
  - Por exemplo, veja a figura abaixo, supondo $esq = 1$, $dir = 10$ e $larg = 1, 2, \ldots 5$



- Adaptando para o nosso caso e incluindo o elemento inicial, ao todo serão afetados `(n - 1 - k*k) // (2*k) + 1` elementos, cujos valores devem passar para `False`.

Depois de implementar essa alteração, chegamos à nossa versão definitiva.

1.2.2 Solução

In [32]:
```
from time import perf_counter

print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'P[:4]':12}  {'P[-4:]'}")

for n in [10, 100, 1000, 10000, 100000, 1000000]:

    start = perf_counter()

    crivo = [True] * n
    for k in range(3, int(n**0.5) + 1, 2):
        if crivo[k]:
            crivo[k*k::2*k] = [False] * ((n - 1 - k * k) // (2 * k) + 1)

    P = [2] + [i for i in range(3, n, 2) if crivo[i]]

    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(P):8}  {P[:4]}  {P[-4:]}')
```

```
       n      demora   #primos  P[:4]         P[-4:]
      10     0.00194         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00001        25  [2, 3, 5, 7]  [79, 83, 89, 97]
    1000     0.00006       168  [2, 3, 5, 7]  [977, 983, 991, 997]
   10000     0.00034      1229  [2, 3, 5, 7]  [9941, 9949, 9967, 9973]
  100000     0.00377      9592  [2, 3, 5, 7]  [99961, 99971, 99989, 99991]
 1000000     0.04960     78498  [2, 3, 5, 7]  [999959, 999961, 999979, 999983]
```

E, agora, nosso algoritmo tem desempenho de gente grande...