Laços encaixados e uma introdução à sua otimização

MC102-2018s1-Aula10-180403

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

03 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, 34, 65, 22, 76, 35, 98, 47, 18, 56]
```

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

```
[12, 34, 65, 22, 76, 35, 98, 47, 18, 56]
```

Em problemas como esse, 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 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:])
```

```
[767017, 132838, 862093, 573006, 766676, 6976, 45506, 954672, 504030, 638343]
[767017, 132838, 862093, 573006, 766676, 6976, 45506, 954672, 504030, 638343]
```

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 [5]: 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 [6]: print(f'{n:7}  {min_i:6}  {S[min_i]:6}  {min_j:6}  {S[min_j]:6}  {min_dist:6}')
```

```
     10       0  767017       4  766676     341
```

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 [7]: from random import choices

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

```
[56894249, 3073965, 72567090, 10370402, 67991005, 21109721, 55403145, 89433917, 36267768, 617
[50007272, 28857364, 24879461, 13915948, 8666166, 83157594, 22510722, 74591083, 68654306, 618
```

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

```
In [8]: from time import perf_counter

        print(f"{'n':>7}  {'demora':>10}  {'i':>8}  {'S[i]':>8}  {'j':>8}  {'S[j]':>8}  {'dis
        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     0.00004         0  56894249         6  55403145   1491104
    100     0.00329        27  89091352        59  89102102     10750
   1000     0.27427       270  81486488       663  81486244       244
  10000    27.22575       638  44352442      4730  44352442         0
```

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 [ ]: from time import perf_counter

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

        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]
```

Solução

```
In [9]: from time import perf_counter

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

```python
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    0.00003         0  56894249        6  55403145   1491104
    100    0.00201        27  89091352       59  89102102     10750
   1000    0.14664       270  81486488      663  81486244       244
  10000   13.54334       638  44352442     4730  44352442         0
```

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 [11]: 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     0.00002       0  56894249       6  55403145  1491104
    100     0.00148      27  89091352      59  89102102    10750
   1000     0.14082     270  81486488     663  81486244      244
  10000    12.12684     638  44352442    4730  44352442        0
```

1.1.2 Solução

```
In [13]: 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         5  55403145         6  56894249   1491104
     100     0.00006        86  89091352        87  89102102     10750
    1000     0.00045       831  81486244       832  81486488       244
   10000     0.00531      4495  44352442      4496  44352442         0
  100000     0.07917       648    705434       649    705434         0
 1000000     0.94604       443     45255       444     45255         0
10000000    15.12531        14       100        15       100         0
```

Nosso algoritmo agora passa a ter 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 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 [ ]: primos = []
        for k in range(2, n):
            # verificar se k é primo
            # se k é primo: adicionar k à lista de primos
```

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

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

```
In [14]: # ler n
         n = int(input('Primos até quanto? '))
```

```
In [15]: primos = []
         for k in range(2, n):
             # testar se k é primo
             k_eh_primo = True
             for d in range(2, k):
                 if k % d == 0:
                     k_eh_primo = False
             # se k é primo: adicionar k à lista primos
             if k_eh_primo:
                 primos += [k]
```

```
In [16]: print(primos)
```

```
[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 [35]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

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

             primos = []
             for k in range(2, n):
                 # testar se k é primo
                 k_eh_primo = True
                 for d in range(2, k):
```

```
            if k % d == 0:
                k_eh_primo = False
        # se k é primo: adicionar k à lista primos
        if k_eh_primo:
            primos += [k]

    end = perf_counter()

    print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00003         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00049        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.05424       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     4.50629      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 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.
- O loop interno pode ser interrompido assim que concluirmos que *k não é primo*,

Vamos incorporar essas alterações ao nosso algoritmo.

```
In [34]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

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

             start = perf_counter()

             primos = [2]
             for k in range(3, n, 2):
                 # testar se k é primo
                 k_eh_primo = True
                 for d in range(3, k, 2):
                     if k % d == 0:
                         k_eh_primo = False
                         break
                 # se k é primo: adicionar k à lista primos
                 if k_eh_primo:
                     primos += [k]
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00093         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00008        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00426       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.27773      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
```

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 [33]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

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

             start = perf_counter()

             primos = [2]
             for k in range(3, n, 2):
                 # testar se k é primo
                 k_eh_primo = True
                 for d in range(3, int(k**0.5) + 1, 2):
                     if k % d == 0:
                         k_eh_primo = False
                         break
                 # se k for primo, adicionar k à lista primos
                 if k_eh_primo:
                     primos.append(k)
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00014         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00009        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00063       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.01023      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000     0.14949      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
 1000000     3.04629     78498  [2, 3, 5, 7, 11, 13]  [999931, 999953, 999959, 999961, 999979
```

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, criar uma lista de *não primos* e, a partir dela, derivar uma lista de *primos*.
Para criar a lista de *não primos*, para cada candidato k vamos colocar na lista todos os seus múltiplos ímpares.

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

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

         for n in [10, 100, 1000, 10000, 100000]:
             raiz_n = int(n**0.5)

             start = perf_counter()
             nao_primos = []

             for k in range(3, raiz_n + 1, 2):
                 for ik in range(3*k, n, 2*k):
                     nao_primos += [ik]

             primos = [2]
             for p in range(3, n, 2):
                 if p not in nao_primos:
                     primos += [p]
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00134         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00003        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00229       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.22776      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000    23.76659      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
```

O desempenho piorou bastante... o que terá acontecido?

- Você consegue ver quantos *loops* **explícitos** e **implícitos** existem agora na nossa solução?
  Será possível melhorar isso?
- Examine as *ranges* dos *loops*?
  Há algo estranho? Algo que possa ser melhorado?

Vamos exibir também o número de elementos na lista de *não primos*.
Veja se isso ajuda...

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

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

         for n in [10, 100, 1000, 10000, 100000]:
             raiz_n = int(n**0.5)

             start = perf_counter()
             nao_primos = []

             for k in range(3, raiz_n + 1, 2):
                 for ik in range(3*k, n, 2*k):
                     nao_primos.append(ik)

             primos = [2]
             for p in range(3, n, 2):
                 if p not in nao_primos:
                     primos.append(p)
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00140         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00006        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00261       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.22105      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000    23.65300      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
```

Você vê algo estranho?

Observe o comprimento da lista de *não primos*.
Faz sentido? O que terá causado isso?

Veja o que fazemos nos *loops* das linhas 10 e 11.
Para cada k na faixa $\left[3, 5, \ldots \sqrt{n}\right]$ colocamos todos os seus múltiplos na lista de *não primos*.

Pense no que acontece para $k = 3, 5$ e 7.
Agora pense no que acontece para $k = 9$.
Nós vamos acrescentar à lista todos os múltiplos de 9. Mas eles já estão lá porque todos são múltiplos de 3.
Nossa lista de *não primos* cresce desnecessariamente. E isso vai nos prejudicar severamente quando formos usá-la para buscar p na linha 16.

Veja o que acontece quando trocamos *listas* por *conjuntos* nesse mesmo algoritmo.

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

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")

         for n in [10, 100, 1000, 10000, 100000, 1000000]:
             raiz_n = int(n**0.5)

             start = perf_counter()
             nao_primos = set()

             for k in range(3, raiz_n + 1, 2):
                 for ik in range(3*k, n, 2*k):
                     nao_primos.add(ik)

             primos = [2]
             for p in range(3, n, 2):
                 if p not in nao_primos:
                     primos.append(p)
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.01075         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00005        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00065       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.00723      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000     0.09169      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
 1000000     1.04663     78498  [2, 3, 5, 7, 11, 13]  [999931, 999953, 999959, 999961, 999979
```

Além de ser menor, a representação de `nao_primos` como *conjunto* ao invés de *lista* leva uma grande vantagem na hora dos testes de pertinência na linha 16.

A desvantagem da *lista* pode ser revertida se adotarmos 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, eliminamos todos os seus múltiplos.
3. Ao chegar ao final, todos os candidatos remanescentes serão primos.

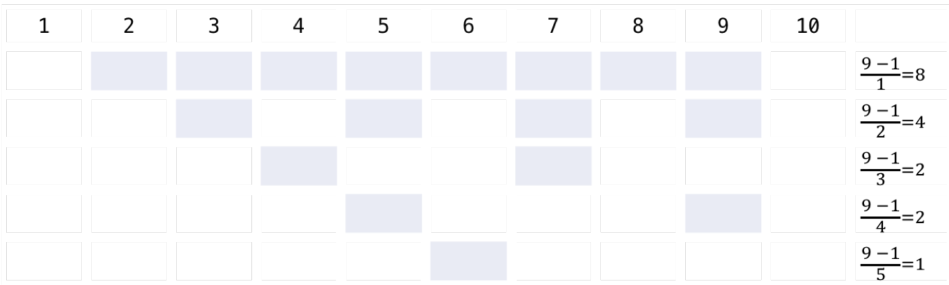
Para evitar o custo da *eliminação* dos múltiplos, vamos apenas marcá-los como *não primos*. Para isso criamos uma lista cujos elementos são todos `True` e convertemos esse valor para `False` quando o elemento é eliminado.

Vamos ver como fica...

```
In [26]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")
         for n in [10, 100, 1000, 10000, 100000, 1000000]:
             raiz_n = int(n**0.5)

             start = perf_counter()
             crivo = [True] * n
             for k in range(3, raiz_n + 1, 2):
                 if crivo[k]:
                     for i in range(k*k, n, 2*k):
                         crivo[i] = False
             primos = [2] + [i for i in range(3, n, 2) if crivo[i]]
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00347         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00002        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00009       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.00093      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000     0.01309      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
 1000000     0.12817     78498  [2, 3, 5, 7, 11, 13]  [999931, 999953, 999959, 999961, 999979
```

Finalmente, podemos acelerar nosso algoritmo ainda mais substituindo o *loop* das linhas 11-12 por uma *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 [24]: from time import perf_counter

         print(f"{'n':>8}  {'demora':>10}  {'#primos':>8}  {'primos[:6] + primos[-6:]'}")
         for n in [10, 100, 1000, 10000, 100000, 1000000]:
             raiz_n = int(n**0.5)

             start = perf_counter()
             crivo = [True] * n
             for k in range(3, raiz_n + 1, 2):
                 if crivo[k]:
                     crivo[k*k::2*k] = [False] * ((n - 1 - k * k) // (2 * k) + 1)
             primos = [2] + [i for i in range(3, n, 2) if crivo[i]]
             # primos = [2] + [i for (i, eh_primo) in enumerate(crivo) if eh_primo]
             end = perf_counter()

             print(f'{n:8}  {end - start:10.5f}  {len(primos):8}  {primos[:6]}  {primos[-6:]}
```

```
       n      demora   #primos  primos[:6] + primos[-6:]
      10     0.00254         4  [2, 3, 5, 7]  [2, 3, 5, 7]
     100     0.00002        25  [2, 3, 5, 7, 11, 13]  [71, 73, 79, 83, 89, 97]
    1000     0.00006       168  [2, 3, 5, 7, 11, 13]  [967, 971, 977, 983, 991, 997]
   10000     0.00043      1229  [2, 3, 5, 7, 11, 13]  [9929, 9931, 9941, 9949, 9967, 9973]
  100000     0.00456      9592  [2, 3, 5, 7, 11, 13]  [99923, 99929, 99961, 99971, 99989, 999
 1000000     0.05430     78498  [2, 3, 5, 7, 11, 13]  [999931, 999953, 999959, 999961, 999979
```

Agora nosso algoritmo tem desempenho de gente grande...