1 Técnicas de Resolução de Problemas

Antes de resolver um problema:

1. Reflita.
2. Leia com atenção o que se pede.
3. Faça exemplos no papel.
4. Observe o comportamento dos resultados e evite que o computador faça cálculos desnecessários.

Exemplo: Somar os elementos em uma Progressão Aritmética.

$$soma = 1 + 3 + 5 + 9 + 11 + 13 + 15 + 17 + 19$$

Primeira solução: Loop

```
1 soma = 0
2 for elem in l:
3     soma += elem
```

Esta solução é linear, ou seja, são feitas $O(n)$ adições para uma lista de n elementos.

Segunda solução: Recursão

```
1 def soma(l):
2     if len(l) == 0: return 0
3     else: return l[0] + soma(l[1:])
```

Esta solução continua realizando $O(n)$ adições para uma lista de n elementos.

Terceira solução: Fórmula da soma da P.A.

$$S_n = \frac{(a_1 + a_n) \cdot n}{2}$$

Portanto,

```
1 soma = ((l[0] + l[-1]) * len(l)) / 2
```

Nesta solução, existe apenas uma conta a se calcular. Portanto, o tempo de execução é constante, ou seja, é $O(1)$ independente do tamanho da lista de entrada.

Quarta solução: Mas e se não decorarmos a fórmula? Se não sabemos a fórmula, podemos observar a solução. Queremos somar:

$$1 + 3 + 5 + 9 + 11 + 13 + 15 + 17 + 19$$

Mas sabemos que:

$$1 + 19 = 20$$
$$3 + 17 = 20$$
$$5 + 15 = 20$$
$$7 + 13 = 20$$
$$5 + 11 = 20$$

Ou seja, a soma do primeiro com o último é igual a soma do segundo com o penúltimo, e assim sucessivamente. Portanto, podemos ir somando os elementos de dois em dois, sabendo que o resultado será sempre o mesmo. No final, como somamos de dois em dois, teremos $\frac{n}{2}$ somas. Portanto, o resultado da soma é:

$$soma = (a_1 + a_n) \cdot \frac{n}{2}$$

Uma dica importante é evitar repetições desnecessárias, como fizemos na soma da P.A. Esta dica também se aplica ao problema a seguir:

Fechem as portas! (Maratona de Programação 2006) Madame Beauvoir possui uma mansão onde ela recebe todos os seus descendentes (netos e bisnetos) durante as férias. Sua mansão possui exatamente N quartos (cada quarto é numerado de 1 a N), onde N é também a quantidade de netos e bisnetos (cada descendente é também numerado de 1 a N).

Como toda criança, os descendentes de Mme. Beauvoir são bastante travessos. Todo dia é a mesma confusão: eles acordam de manhã cedo antes dela e se encontram no grande jardim. Cada descendente, um de cada vez, entra na mansão e troca o estado das portas dos quartos cujos números são múltiplos do seu identificador. Trocar o estado de uma porta significa fechar uma porta que estava aberta ou abrir uma porta que estava fechada. Por exemplo, o descendente cujo identificador é igual a 15 vai trocar o estado das portas 15, 30, 45, etc.

Considerando que todas as portas estão inicialmente fechadas (todos os descendentes fecham as portas antes de descerem para o jardim) e que cada descendente entra exatamente uma vez na mansão (a confusão é tão grande que não sabemos em que ordem), quais portas estarão abertas após a entrada de todos os descendentes na mansão?

Entrada: Leitura de um número inteiro N ($0 \leq N \leq 25000000$), indicando o número de portas e descendentes.

Saída: Impressão de uma linha, com a sequência crescente de números correspondente aos identificadores dos quartos cujas portas estarão abertas.

Exemplos:

Entrada	Saída
1	1
2	1
3	1
4	1 4

Primeira solução: Lista com o estado das portas

```
1 def portas(n):
2     l = n * [False]      # n portas, todas iniciam fechadas
3     for neto in range(1, n+1):
4         for porta in range(neto, n+1, neto):
5             l[porta-1] = not l[porta-1]
6
7     for porta in range(len(l)):
8         if l[porta]: print(porta+1)
```

Na linha 1, criamos uma lista com o estado de n portas, todos inicialmente fechadas (representado pelo booliano `false`). Na linha 2, iteramos sobre cada um dos descendentes. Na linha 3, para cada descendente, iteramos sobre cada porta que será modificada por ele (repare no incremento do contador). Na linha 4, alteramos o estado de cada porta a ser modificada por cada descendente (o comando **`not`** alterna o valor de uma variável booliana de `True` para `False` e vice-versa).

Os dois *loops* aninhados indicam que o algoritmo é quadrático (ou seja, $O(n^2)$).

Segunda solução: Para buscar uma solução linear, vamos analisar a saída do programa. Para $N = 100$, temos as seguintes portas abertas:

$$1, 4, 9, 16, 25, 36, 49, 64, 100$$

Ou seja, a resposta contém os quadrados perfeitos até 100. Mas por quê?

O estado de cada porta será alterado por seus "divisores". Todo número possui um número par de divisores:

$$divisores(15) = 1, 3, 5, 15$$
$$divisores(24) = 1, 2, 3, 4, 6, 8, 12, 24$$

Para cada divisor x de y, existe outro número z tal que $x \cdot z = y$. E se o estado de uma porta é alterado por um número par de netos, ela terminará fechada.

Entretanto, apenas os quadrados perfeitos possuem um número ímpar de divisores:

$$divisores(16) = 1, 2, 4, 8, 16$$
$$divisores(25) = 1, 5, 25$$

Apenas portas abertas um número ímpar de vezes terminarão abertas. Portanto, podemos criar uma função para verificar se um número é quadrado perfeito:

```
1 def quadPerf(x):
2     r = (int)(x**0.5)
3     return r*r == x
4
5 def portas2(n):
6     for porta in range(1, n+1):
7         if quadPerf(porta): print(porta)
```

Ao invés de dois *loops* aninhados, agora temos apenas um. O algoritmo passa a ser linear, ou seja, $O(n)$.

Terceira solução: Ainda podemos melhorar. Sabemos que a quantidade de número na saída para n portas será $\sqrt{n}$. Portanto, podemos fazer um *loop* com apenas esta quantidade de elementos:

```
1 def portas3(n):
2     porta = 1
3     while porta*porta <= n:
4         print(porta*porta)
5         porta += 1
```

Este último algoritmo, portanto, é $O(\sqrt{n})$.