Um padrão útil

In [ ]:

```
i = 0
while i < n:
    i += 1
```

Invariante de laço

Um **invariante de laço**, ou simplesmente **invariante**, é uma **condição lógica** que

1. É válida no ínicio da execução de um laço
2. Se mantém válida após cada execução do corpo do laço

Observe que um invariante também será válido no momento em que ele termina

Invariantes são usados para provar que um programa é correto (faz o que está especificado para fazer)

No entanto, também são úteis para entender e **projetar** algoritmos

Vejamos como exemplo

In [ ]:

```
i = 0
# 1. inv
while i < n:
    # 2. inv
    i += 1
```

Temos como invariantes

- `i` é um inteiro
- `i >= 0`

Não são invariantes

- `i == 1200`
- `i == 0`
- `i > 0`
- `i <= 0`
- `i < n`

Um invariante útil em provas

In [ ]:

```python
# pré-condição: n >= 0
i = 0
while i < n:
    # invariante: i <= n
    i += 1
```

Se `n` é um número natural (`>=0`), um invariante **interessante** é

- `i <= n`

Observe que um while termina quando sua respectiva condição é falsa, no exemplo, quando `i >= n` .

Logo, disto e do invariante `i<=n` conclui-se que ao termino do while temos

- `i == n`

Outro exemplo

In [ ]:

```python
i = 0
while i < n:
    i += 5
```

Temos como invariantes:

- `i >= 0`
- `i` é múltipo de `5` , ou seja, `i % 5 == 0`

Não é invariante

- `i <= n`

Quantas vezes itera?

- `n // 5 + 1` ?

- se `n` é múltiplo de `5` , itera `n //5` , senão `itera n//5 + 1`

Outros invariantes:

- se `n` é múltiplo de `5` , então `i <= n//5`
- se `n` não é múltiplo de `5` , `i <= n//5 + 1`

Laços como aproximações à solução

Idéia para projetar algoritmos: a cada repetição do corpo de um laço, o programa se aproxima à solução desejada

In [ ]:

```
i = 0
while i < n:
    i = i + 1
```

No script anterior, o efeito final é deixar `i==n`

- A cada iteração do laço `i` se aproxima mais a `n`
- O valor de `i` para a próxima iteração é calculado em função do valor que tinha `i` na iteração anterior (`i += 1`)
- O invariante `i<=n` junto com a condição de terminação do `while` garantem que `i==n`

Método geral

- Analise se algum esquema de repetição se adequa ao problema
- Identifique variáveis cujos valores podem ser calculados progressivamente, aproximando-se à solução a cada repetição
  - **O valor da próxima iteração deve depender do valor da iteração anterior**
- Estabeleça invariantes que caracterizem a aproximação
- Codifique o corpo do lação (`while` ou `for`) de tal forma que o invariante seja estabelecido no início e preservado a cada iteração

Exemplo: Fatorial

Queremos calcular o fatorial de n (n>=0).

```
n! = 1*2*3*4*...*n
```

Para incluir o caso em que `n==0` é conveniente analisar assim

```
n! = 1*1*2*3*4*...*n
```

- Podemos fazer por aproximações, primeiro calculando fatorial de `0` , depois de `1` , de `2` e assim sucessivamente até chegar no fatorial de `n`
  - Usaremos o esquema de repetir `n` vezes
- Precisaremos de variáveis

- `i` para o controle do laço
- `fat` onde vamos colocando a aproximação e que no final terá o resultado desejado

- Os invariantes
  - `fat == i!`
  - `i <= n` (daqui para frente teremos ele sempre em mente, porém implicitamente)

In [1]:

```
def fatorial(n):
# pré-condição: n >= 0
# retorna: n!
    i = 0
    fat = 1
    while i < n:
    # invariante : fat = i!
        i = i+1
        fat = fat * i
    return fat

fatorial(5)
```

Out[1]:

```
120
```

Exemplo: Cálculo do número de Euler e

Aproximações da constante e podem ser obtidas a partir da seguinte série

$$e = \sum_{i}^{\infty} \frac{1}{i!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Queremos escrever uma função que, dado n, calcule a aproximação de e usando n termos desta série

In [2]:

```
def eAprox(n):
# pré-condição: n>=0
    i = 0
    e = 0
    while i < n:
    # invariante: e = 1/0! + 1/1! + 1/2! + 1/3! + ... + 1/(i-1)!
        e = e + 1/fatorial(i)
        i += 1
    return e

eAprox(20)
```

Out[2]:

```
2.7182818284590455
```

Esta implementação de `eAprox` é ineficiente. A cada iteração calculamos `i!` (do zero)

Observe que a cada iteração, poderiamos calcular `i!` a partir do valor anterior `(i-1)!` apenas multiplicando por `i`, evitando a chamada `fatorial(i)`

Podemos usar mais uma variável, chamada `fat`, com invariante `fat = i!`

In [3]:

```
def eAprox(n):
    i = 0
    fat = 1
    e = 0
    while i < n:
    # invariante: fat = i!
    # invariante: e = 1/0! + 1/1! + 1/2! + 1/(k-1)! + 1/k! +... + 1/i-1!
        e = e + 1/fat
        i += 1
        fat = fat * i
    return e

eAprox(20)
```

Out[3]:

```
2.7182818284590455
```

Podemos ainda melhorar um pouco a eficiência do algoritmo evitando uma multiplicação a cada iteração

Observe que a cada iteração, o cálculo de `1/i!` poderia se beneficiar do valor obtido na iteração anterior, ou seja,
`1/(i-1)!`

Podemos adicionar uma variável chamada `termo` com invariante de laço `termo = 1/i!`

In [4]:

```
def eAprox(n):
    i = 0
    termo = 1
    e = 0
    while i < n:
    # invariante: termo = 1/i!
    # invariante: e = 1/0! + 1/1! + 1/2! + ... + 1/i-1!
        e = e + termo
        i += 1
        termo = termo / i
    return e

eAprox(20)
```

Out[4]:

```
2.7182818284590455
```

Busca sequencial

Dada uma sequência de valores, procurar o primeiro elemento que satisfaz uma dada propriedade.

Exemplo: propriedade do 3025

Repare a seguinte característica do número `3025` :

$30 + 25 = 55$ e $55^2 = 3025$

Queremos saber qual é o menor número entre `1000` e `9999` que possui a mesma característica.

Definamos primeiro uma função `prop3025` que verifica se um número possui a mesma característica do `3025` .

In [5]:

```
def prop3025(n):
    return (n // 100 + n % 100) ** 2 == n

prop3025(3025)
```

Out[5]:

```
True
```

Para encontrar o menor número que satisfaz a propriedade do 3025, podemos realizar **uma busca sequencial** começando em `1000` e avançando de um em um até achar no número desejado.

In [7]:

```
i = 1000
while True :
    #invariante: para todo k=1000..i-1, not prop3025(k)
    if prop3025(i):
        resp = i
        break
    i += 1

i
```

Out[7]:

```
2025
```

In [8]:

```
i = 1000
while not prop3025(i) :
    #invariante: para todo k=1000..i-1, not prop3025(k)
    i += 1

i
```

Out[8]:

```
2025
```

Exemplo: mínimo múltiplo comum

Podemos usar busca sequencial, começando com `i = 1` e avançando de um em um, procuramos o primeiro `i` que seja múltiplo de `m` e `n` ao mesmo tempo.

In [9]:

```
def mmc(m, n):
# pré-condição: m e n >= 1
# retorna: mínimo múltiplo comum de m e n
    i = 1
    while i % m != 0 or i % n != 0:
    # invariante: para todo k=1..i-1, k não é múltiplo comum de m e n
        i += 1
    return i

mmc(20, 12)
```

Out[9]:

```
60
```

Podemos melhorar a eficiência consideravelmente

- Iniciar a busca a partir do maior entre `m` e `n`
- Se, por exemplo, o maior for `m`, restringir a buscar somente dentre os múltiplos de `m`

In [10]:

```
def mmc(m, n):
    if m < n:
        m, n = n, m
    i = m
    while i % n != 0:
    # invariante: i é múltiplo de m
    # invariante: para todo k=m..i-1, k não é múltiplo comum de m e n
        i += m
    return i

mmc(20, 12)
```

Out[10]:

```
60
```

Sequencias de valores terminadas por uma marca

Exemplo: ler valores inteiros até a leitura de um negativo

Observe que é um problema de busca sequencial. Dentre os lidos, procuramos pelo primeiro negativo.

In [11]:

```
val = int(input())
while val >= 0:
    # invariante: todos os lidos antes do último val são >= 0
    val = int(input())

print("terminou")
```

```
12
24
8
0
41
-12
terminou
```

Exemplo: soma de uma sequência de números

Dada como entrada uma sequência de números finalizada por um número negativo, calcular a soma de todos os números (incluindo o último, que é negativo)

In [12]:

```
val = int(input())
soma = val
while val >= 0:
    # invariante: soma = soma de todos os lidos até o momento
    val = int(input())
    soma += val

print("resultado =", soma)
```

```
34
12
35
1
4
-5
resultado = 81
```

Se não queremos incluir o último número (o negativo) na soma:

In [13]:

```
val = int(input())
soma = 0
while val >= 0:
    # invariante: soma = soma de todos os lidos até antes do último val
    soma += val
    val = int(input())

print("resultado =", soma)
```

```
34
12
35
1
4
-5
resultado = 86
```