Algoritmos e Programação I: Aula 12*

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
79070-900 Campo Grande, MS
`https://ava.ufms.br`

Sumário

1 Quantificadores

Na solução de problemas algoritmos ocorrem várias situações em que temos que verificar se uma dada propriedade ocorre para todos elementos de um dado conjunto. Um exemplo clássico disso é o de verificar se um dado conjunto está ordenado. Outro exemplo é o de calcular a maior nota de um dado conjunto de notas (a maior nota é maior ou igual as demais notas do conjunto). Numa outra classe de problemas, temos que efetuar alguma operação com todos os elementos de um dado conjunto. Como vimos anteriormente, um programa nada mais é que uma função, e da mesma forma que usamos conceitos de lógica para descrever a estrutura de controle de seleção, vamos descrever como podemos descrever matematicamente a repetição de um conjunto de instruções. Para isso, vamos utilizar os quantificadores universal (qualquer que seja - $\forall$) e existencial (existe - $\exists$).

Inicialmente vamos descrever matematicamente como podemos um conjunto ordenado. Para descrição dessas propriedades, vamos representar os conjuntos com uma lista. Numa lista, os elementos possuem o mesmo "nome" e são associados a um índice que corresponde a posição do elemento no conjunto. Como veremos posteriormente, o Python possui várias estruturas para representar um conjunto de elementos. Uma dessas estruturas é a `list` (lista). Na media em que for necessário, iremos apresentando as característica da estrutura `list` do Python. Vamos agora descrever um conjunto ordenado utilizando o quantificador universal. Considere a saída de um algoritmo dada por uma *lista* B, onde B é dada por

$$B = (e'_1 e'_2 \dots e'_{30})$$

*Este material é para o uso exclusivo da disciplina de Algoritmos e Programação I da FACOM/UFMS e utiliza as referências bibliográficas da disciplina.

a qual é uma permutação de uma outra *lista* A com a propriedade que, para todo i entre 1 e 29 inclusive, $e'_{i+1} \geq e'_i$. Essa afirmação é na realidade denominada uma *conjunção* de dois predicados. O primeiro, "B é uma permutação de A", é um predicado com 30 variáveis. O segundo, "para todo i entre 1 e 29 inclusive", $e'_{i+1} \geq e'_i$, contém 29 predicados da forma $e'_{i+1} \geq e'_i$, um para cada inteiro de 1 até 29. Isto é, ele serve como uma notação para o predicado

$$e'_2 \geq e'_1 \wedge e'_3 \geq e'_2 \wedge \ldots \wedge e'_{30} \geq e'_{29}$$

Pelo fato de que afirmações como "para todo"e "para cada"ocorrem frequentemente em matemática e lógica, elas são abreviadas pelo símbolo especial $\forall$, chamado quantificador universal. Neste exemplo, escrevemos a frase "para todo i entre 1 e 29 inclusive, $e'_{i+1} \geq e'_i$"da seguinte forma:

$$\forall i \in \{1, 2, \ldots, 29\} : e'_{i+1} \geq e'_i$$

onde os símbolos $\forall i \in$ são seguidos pela representação do domínio da variável i e o predicado que descreve o que é verdadeiro para *todo* valor da variável em seu domínio. O domínio de uma variável é como seu *tipo* - isto é, o conjunto de valores cujos membros podem ser atribuídos àquela variável. De uma forma mais genérica, o seguinte predicado pode ser composto dos predicados $R(i)$ e $P(i)$:

$$\forall\, R(i) : P(i)$$

onde, $R(i)$ é usado para descrever o domínio de i e $P(i)$ deve ser satisfeito para todo valor i no seu domínio para que todo o predicado $\forall\, R(i) : P(i)$ seja válido - i.e., seja **True** para todos os casos.

Predicados com o quantificador universal podem ser testados em um programa Python com a utilização de uma estrutura de repetição (laço). Em geral, para o predicado

$$\forall i \in \{1, \ldots, n\} : P(i)$$

podem ser testados pela seguinte estrutura de **repetição** (laço), onde a variável `resultado` do tipo `bool` será `True` ou `False` dependendo respectivamente se o predicado é satisfeito ou não.

```
# Python
# bool resultado
# int n

resultado = True
for i in range(1,n+1):
   if !P(i):
      resultado = False
```

onde assumimos que o predicado é verdadeiro antes da execução do laço, e então para cada valor da variável `i` o laço testa para verificar se exite um `P(i)` que seja falso.

Lembramos que uma conjunção múltipla somente é verdadeira se todos os seus operandos forem verdadeiros simultaneamente. Se apenas um é falso, então toda a conjunção é falsa. Essa é a ideia na qual o laço acima foi escrito. Os exemplos seguintes mostram alguns dos muitos usos para predicados quantificados. Eles descrevem o estado de uma computação,

e em conjunto com os respectivos laços, são usados para testar se um predicado é ou não satisfeito.

Como faremos a implementação dos quantificadores em Python, precisamos de uma estrutura de dados para armazenar uma coleção de objetos. Já descrevemos que o Python possui diferentes formas para representar um coleção de objetos e que no decorrer da disciplina iremos apresentando as diferentes estruturas. Vamos descrever de uma maneira bem informal a estrutura de dados `list`. Quando representamos um conjunto no Python com uma lista, a estrutura de dados `list`, com apenas uma variável, possibilita referenciar individualmente cada um dos elementos da lista:

```
lista = [5, 7, 4, 10]
lista[0] == 5
lista[1] == 7
lista[2] == 4
lista[3] == 10
```

e nas próximas aulas descreveremos com mais detalhes a estrutura de lista.

Cada uma das afirmações abaixo é seguida pela sua representação como um predicado que usa o quantificador universal:

a. Todo elemento e_i de e_1 a e_{21} é diferente de 9:

$$\forall\, i \in \{1, 2, \ldots, 21\} : e_i \neq 9$$

```
# Python
# list e
# bool resultado

resultado = True
for i in range(1,22):
   if not(e[i] !=9):
      resultado = False
```

b. Todos e_j entre e_m e e_n são inteiros positivos:

$$\forall j \in \{m, m+1, \ldots, n\} : e_j > 0$$

```
# Python
# list e
# bool resultado
# int m, n

resultado = True
for i in range(m,n+1):
   if not(e[j] > 0):
      resultado = False
```

c. Em uma lista $A = (e_1 e_2 \ldots e_n)$, os elementos até, mas não incluindo o j-ésimo, estão arranjados em ordem crescente:

$$\forall i \in \{1, 2, \ldots, j-2\} : e_i \leq e_{i+1}$$

```
# Python
# list e
# bool resultado
# int j

resultado = True
for i in range(1,j-1):
   if not(e[i] <= e[i+1]):
      resultado = False
```

Note que não existe exigência que qualquer uma destas instruções seja verdadeira para todos os possíveis valores das variáveis; a exigência é somente que são verdadeiras para qualquer conjunto particular de valores que caia dentro do domínio estabelecido.

Existem vezes que a variação para o predicado quantificado pode ser omitida se a variação é clara no contexto - isto é, se existe um conjunto universo que tenha sido claramente especificado. Por exemplo, se sabemos que i é um inteiro (o qual é muitas vezes o caso), podemos escrever

$$\forall\, i : i < i + 1$$

de forma análoga, podemos rescrever os predicados

$$\forall\, R(i) : P(i)$$

como $\forall\, i : R(i) \Rightarrow P(i)$. Por exemplo,

$$\forall\, i \in \{1, \ldots, 21\} : e_i \neq 9$$

é o mesmo que $\forall\, i : 1 \leq i \leq 21 \Rightarrow e_i \neq 9$. Que pode ser lido como "se i está no intervalo de 1 a 21, então e_i não é igual a 9". Desta forma, ele é equivalente a afirmação, "para todo i na variando de 1 a 29, e_i não é igual a 9".

Um segundo quantificador de interesse é o chamado *quantificador existencial* denotado matematicamente pelo símbolo $\exists$. Ele pode ser lido como, "existe"ou "para algum". Ele é usado para abreviar múltiplas ocorrências do operador de disjunção quando diversos predicados similares estão ligados por ele.

O quantificador existencial, $\exists\, i \in \{1, \ldots, n\} : e_i = 0$ é uma notação para a $(e_1 = 0) \vee (e_2 = 0) \vee \ldots \vee (e_n = 0)$. Isto é, da mesma forma que o quantificador universal é uma notação para uma sequência de $\wedge$´s, o quantificador existencial é uma notação para uma sequência de $\vee$´s. Note que o quantificador existencial não informa qual valor (quais valores) tornam o predicado **True**, simplesmente informa que existe pelo menos um valor tal que o predicado é satisfeito. Um teste para um predicado quantificado existencialmente também pode ser construído usando um laço. Para efetuar isso para o predicado na sua forma geral

$$\exists\, i \in \{1, \ldots, n\} : P(i)$$

escrevemos um laço na seguinte forma básica:

```
# Python
# bool resultado
# int n

resultado = False
for i in range(1,n+1):
   if (P(i)):
      resultado = True
```

Esta construção segue a lógica reversa do laço representando o quantificador universal. Lembre-se que para uma disjunção de predicados ser **True**, necessitamos somente encontrar um de seus constituintes predicados como sendo **True**. O único caso na qual toda a disjunção será **False** é se todo constituinte for **False**.

Então, iniciamos o laço assumindo que toda a disjunção será `False`, e então usamos o laço para buscar um valor de `i` que satisfaça o predicado `P(i)`. Se encontramos um (ou mais) então o resultado da avaliação de todo o predicado será `True`; caso contrário, ele permanecerá `False`. Veremos alguns exemplos da utilização do quantificador existencial em um predicado, em conjunto com o laço Python correspondente que irá testar se o predicado é satisfeito pelos valores correntes das variáveis em questão. Abaixo descrevemos exemplos de sentenças com sua representação usando o quantificador existencial:

a. Existe um elemento da lista $L = (e_1 e_2 \dots e_n)$ que é zero.

$$\exists\, i \in \{1, \dots, n\} : e_i = 0$$

```
# Python
# list L
# bool resultado
# int n

resultado = False
for i in range(1,n+1):
   if L[i] == 0:
      resultado = True
```

b. Os elementos da lista $L = (e_1 e_2 \dots e_n)$ não estão em ordem crescente.

$$\exists\, j \in \{1, \dots, n\} : e_{j+1} < e_j$$

```
# Python
# list L
# bool resultado
# int n

resultado = False
for j in range(1,n):
```

```
    if L[j+1] < L[j]:
        resultado = True
```

Muitas vezes ocorre o caso em que os quantificadores universal e existencial são necessários para representar um único predicado. Por exemplo, vamos expressar simbolicamente que $B = \{e'_1 e'_2 \dots e'_{30})$ é uma permutação de $A = \{e_1 e_2 \dots e_{30}\}$. Suponha que todos elementos e_j são distintos. Podemos expressar isto como segue:

$$perm(B, A) = \forall i \in \{1, \dots, 30\} \exists j \in \{1, \dots, 30\} : e'_i = e_j.$$

ou seja, B é uma permutação de A se, para todo elemento e'_i de B existe um elemento idêntico e_j em A, embora não necessariamente na mesma posição (isto é, os índices i e j não necessitam ser idênticos para $e'_i = e_j$ ser verdadeiro). Se os elementos e_i não são todos distintos, a situação é mais complicada porque devemos expressar o conceito que todos os elementos originais de A aparecem em B. A formulação acima permitiria que elementos duplicados de A não aparecessem em B.

Além dos quantificadores $\forall$ e $\exists$, outros quantificadores são muito usados em computação. Vamos descrever cinco deles, cuja descrição em lógica matemática é a seguinte:

- **Num** $R(i)$: $P(i)$ dá o número de elementos i no intervalo dado por R que satisfaz o predicado P.
- **Min** $R(i)$: $P(i)$ retorna o valor mínimo i para o qual $P(i)$ é **True**, com i restrito ao intervalo dado por R.
- **Max** $R(i)$: $P(i)$ retorna o valor máximo i para o qual $P(i)$ é **True**, com i restrito ao intervalo dado por R.
- **Soma** $R(i)$: $f(i)$ é usada para expressar o resultado de somar todos os valores de $f(i)$ onde i toma todos os possíveis valores sobre o domínio especificado por R. Em matemática, Soma é muitas vezes abreviada por $\sum$.
- **Prod** $R(i)$: $f(i)$ é usada para expressar o resultado do produto de todos os valores de $f(i)$ onde i toma todos os possíveis valores sobre o domínio especificado por R (algumas vezes em matemática o produto é denotado pela letra Grega $\prod$).

Note que nenhum desses quantificadores resulta em um valor booleano e que diversos requerem a avaliação para cada valor do índice i em uma função ao invés de uma proposição. Como os outros quantificadores, estes cinco quantificadores podem ser usados em predicados que são implementados na forma de um laço em Python. Estes predicados e seus equivalentes laços em Python são ilustrados nos exemplos abaixo:

a. Dado um conjunto de notas {6.0 7.0 4.0 8.0}, determinar o número de notas maior que 6.0. Usando um quantificador **Num**, isso pode ser descrito como: **Num** $i \in \{0, \dots, 3\}$: $notas[i] >= 6.0$. O laço que implementa o quantificador é o seguinte:

```
# Python
# list notas
# int contador

contador = 0
```

```
for i in range (0,4):
   if notas[i] >= 6.0:
      contador += 1
```

```
# Python usando a função sum()
# list notas
# int contador

contador = sum(1 for i in range(0,len(notas)) if notas[i] >= 6.0)
```

Observe que a variável contadora (`contador`) é usada para contar o número de vezes que o predicado `notas[i] >= 6.0` for `verdadeiro`, para todos os valores de `i` no intervalo de 0 a 3. Cada vez que o valor de `i` satisfaz este predicado, `contador` é incrementado por 1.

b. Dado um conjunto de notas entre 0.0 e 10.0, {6.0 5.0 3.5 4.5 7.0 4.0 8.0}, determinar o menor índice i tal que $notas[i] < 6.0$. Usando um quantificador **Min**, isso pode ser descrito como: **Min** $i \in \{1, \ldots, 7\} : notas[i] < 6.0$.

```
# Python
# list notas
# int indmin

indmin = -1
for i in range(0,7):
   if notas[i] < 6.0 and indmin == -1:
      indmin = i
```

```
# Python com a função min
# list notas
# int indmin

try:
   indmin = min(i for i in range(0,len(notas)) if notas[i] < 6.0)
except:
   menor = -1
```

Observe que a variável `inmin` é usada para armazenar o menor valor de `i` para o qual o predicado `notas[i] < 6.0`, para todos os valores de `i` no intervalo de 0 a 6. O primeiro valor de `i` satisfazendo este predicado é atribuído a `menor`; para outros valores de `i` que satisfizerem este predicado, a variável `indmin` não sofrerá alteração. Isto é garantido pela condição `indmin == -1` na instrução condicional, e `menor` é inicialmente atribuído o valor de -1 para sinalizar o fato que a ele ainda não foi atribuído qualquer valor de `i` no intervalo desejado. Se por outro lado nenhum desses valores de `i` satisfizer o predicado, então `menor` será deixado com o valor -1 para então designar essa saída.

c. Dado um conjunto de notas {6.0 5.0 3.5 4.5 7.0 4.0 8.0}, determinar o maior índice i tal que $notas[i] < 6.0$. Usando um quantificador **Max**, isso pode ser descrito como: **Max** $i \in \{1, \ldots, 7\} : Nota[i] < 6.0$.

```
# Python
# list notas
# int indmax

indmax = -1
for i in range(0,7):
   if notas[i] < 6.0:
      indmax = i
```

```
# Python com a função max()
# list notas
# int indmax

try:
   indmax = max(i for i in range(0,len(notas)) if notas[i] < 6.0)
except:
   indmax = -1
```

Observe que este laço usa uma estratégia inversa da acima. Isto é, pela ordem na qual cada valor diferente de `i` é atribuído para cada repetição subsequente do laço, o índice maior valor de `i` estará armazenado em `indmax` para a última repetição do laço que satisfaz o predicado. Também podemos implementar esse quantificador usando um laço `while` no Python como segue:

```
# Python
# list notas
# int indmax

indmax = -1
i = 0
while i < 7:
   if notas[i] < 6.0:
      indmax = i
   i += 1
```

d. Dado um conjunto de notas {6.0 5.0 3.5 4.5 7.0 4.0 8.0}, determinar a soma das notas. Usando um quantificador **Soma**, isso pode ser descrito como: **Soma** $i \in \{1, \ldots, 7\}$: $Nota[i]$

```
# Python
# list notas
# float soma
```

```
soma = 0.0
for i in range (0,7):
   soma = soma + notas[i] # soma += notas[i]
```

```
# Python usando a função sum()
# list notas
# float soma

soma = sum(notas[i] for i in range(0,7))
```

Observe que a cada vez que esse laço é repetido, outro valor (`notas[i]`) é adicionado à variável `soma`, a qual controla a soma total.

e. Dado um conjunto de números {6 5 3 4 7}, determinar o produto desses números. Usando um quantificador **Prod**, isso pode ser descrito como: **Prod** $i \in \{1, \ldots, 5\} : numeros[i]$.

```
# Python
# list numeros
# int produto

produto = 1
for i in range (0,7):
   produto = produto*numeros[i]
```

```
# Python usando a função reduce()
# list numeros
# int produto

produto = reduce(operator.mul, numeros)
```

Observe que a cada vez que esse laço é repetido, outro valor (`numeros[i]`) é multiplicado pela variável `produto`.

A utilização de predicados auxilia no projeto e implementação de programas para a solução de problemas algorítmicos. Os predicados podem ser utilizados para definir as condições exatas sob as quais o programa será executado. Também é fundamental a correspondência entre predicados quantificados e laços em programas. Em Algoritmos e Programação I vamos desenvolver um pouco essa ideia. Posteriormente, em outras disciplinas, essa formalização será utilizada para a prova da correção de algoritmos (*invariante do laço*).

Inicialmente introduzimos pré e pós-condições para programas em geral. Isto é, as pré-condições representam um predicado que descreve todos os possíveis estados da entrada antes da execução do programa, e as pós-condições representam um predicado que descreve todas as possíveis saídas que correspondem às entradas para o algoritmo que sendo tratado. Caso necessitemos, as pré e pós-condições podem ser utilizadas antes e depois de cada instrução. Isso permite uma argumentação poderosa em relação a correção do programa.

SÍMBOLO	SIGNIFICADO	SUBSTITUTO ASCII
$notas_i$	Seleção do i-ésimo elemento em uma lista	`notas[i]`
$\emptyset$	Conjunto vazio	`vazio *`
$\sum$ ou *soma*	Somatória	`sum`
$\leq$	Menor ou igual	`<=`
$\geq$	Maior ou igual	`<=`
$\neq$	Diferente	`!=`
$=$	Igual	`==`
$\forall$	Quantificador Universal	`para todo *`
$\exists$	Quantificador Existencial	`existe *`
$\wedge$	Conjunção "e"	`and`
$\vee$	Disjunção "ou"	`or`
$\sim$	Negação	`not`
{ e }	Chaves (para conjuntos)	{ e } *
$\in$	Pertence	`in`

Tabela 1: SUBSTITUTOS ASCII PARA SÍMBOLOS FORMAIS LÓGICOS

Todo o ferramental da área de lógica matemática pode ser utilizada como um método correto e conciso de descrever exatamente o que estamos tentando executar na nossa solução computacional. A lógica matemática providencia a melhor técnica conhecida para desenvolver um tal método. Sempre que necessário, utilizaremos lógica matemática como a base de nossas especificações do problema.

Bons princípios de projeto de algoritmos sugerem o uso de especificações formais por duas razões: para auxiliar no processo de solução de problemas e para providenciar orientação para leitores e usuários de nossos programas. A discussão anterior sugere que estas especificações tem a forma de declarações a respeito do estado do nosso sistema: pré-condições, pós-condições, e afirmações intermediárias colocadas estrategicamente entre as instruções do programa como comentários.

Na descrição dos passos do algoritmos que obtemos nos refinamentos, utilizamos uma linguagem de especificação baseada em lógica matemática junto com algumas expressões em Português sempre que o seu significado for claro e for mais simples que uma declaração em lógica formal. Enquanto linguagens de especificação formal rigorosamente matemáticas são usadas no projeto e manutenção de grandes sistemas, eles muitas vezes não tem a simplicidade e clareza das especificações informais.

A Tabela 1 resume como os símbolos lógicos devem ser digitados quando eles são representados dentro de programas `Python`. Lembrando que os símbolos destacados com um asterisco (`*`) na coluna do lado direito não são parte da linguagem `Python`; eles devem ser usados somente dentro de comentários.

2 Iteração e Laços

A escolha de um mecanismo apropriado para controlar uma sequência de operações num computador é a chave para desenvolver uma solução algorítmica efetiva para um problema. Frequentemente, o detalhamento da especificação do problema indica a solução. Por exemplo, a disjunção (`or`) que aparece nas especificações (pré- ou pós-condições) implica que uma seleção (instrução `if`) deve ser feita, enquanto uma conjunção (operador `and`) sugere o uso de uma sequência de computações independentes. Um quantificador (para todo, existe,

soma, etc) pede por uma instrução de iteração (laço `for` ou `while`). À medida que exploramos os diferentes problemas e suas soluções, tentaremos demonstrar como as especificações são úteis para ajudar a determinar o tipo de estrutura de controle que deve ser usado.

Como destacamos anteriormente, em computação temos três tipos de estruturas de controle: sequencial, seleção e repetição (iteração). Vamos agora descrever soluções de problemas computacionais que utilizam estruturas de controle iterativas, ou seja, necessitam que um conjunto de expressões sejam repetidas. A estrutura de controle que permite a iteração (repetição) de uma ou mais instruções é a mais complexa das três estruturas. Quando incluímos na nossa solução uma estrutura de controle de repetição temos que olhar muito atentamente se a estrutura está gerando a solução correta. Inicialmente vamos tratar a solução de problemas que utilizam apenas variáveis simples. Trataremos os problemas da soma de uma sequência de números, da potência inteira de um dado número e a contagem de caracteres num texto.

As estruturas de controle de repetição (iteração) são formuladas num algoritmo ou programa como laços. Analisaremos a formulação usada num laço, e como sua especificação correspondente assegura a precisão da solução. O laço permite-nos escrever de maneira compacta uma sequência de expressões que será repetida diversas vezes.

Definição: Um *laço* é uma sequência de instruções que podem ser executadas 0, 1, ou um número finito n de vezes.

O laço é um dos mais importantes e fundamentais conceitos para o desenvolvimento de algoritmos. Por que os programas precisam de laços? Existem duas razões principais:

1. Economia de instruções

2. Imprevisibilidade de tamanho de entrada

Ambas as razões para laços podem ser ilustrados no seguinte exemplo: Suponha que precisamos desenvolver um algoritmo que compute `soma` dos números inteiros de 1 a 6.

```
...
# int soma
...
soma = 1 + 2 + 3 + 4 + 5 + 6
print(soma)
```

Esta abordagem é, claramente, deselegante quanto inflexível. Se tivermos que estender este problema um pouco e requerer que `soma` seja a soma de, digamos, `1+...+100`, esta abordagem torna-se insatisfatória. Ainda mais difícil, e beirando ao impossível, seria uma extensão do problema sendo que *soma* deveria ser a soma de um número indefinido (ainda assim finito) de inteiros `1+...+n`. Isto é, o valor de `n` é uma entrada do problema, e o programa deve estar preparado para computar a soma `soma` para qualquer valor particular de `n`. Nesse caso, o programa deve funcionar para qualquer valor de `n`, podendo ser sempre executado, seja com `n == 6`, `n == 100` ou `n == 1000`.

Como descrevemos acima, os laços estão estreitamente relacionados com os quantificadores que aparecem nas afirmações. Por exemplo, $\forall\, i \in \{1, \ldots, n\} : \ldots$ indica que devemos usar um laço para obter o estado no qual essa condição é verdadeira pois precisamos assegurar que ela seja verdadeira *para todos* os casos descritos. Similarmente, $\exists\, i \in \{1, \ldots, n\} : \ldots$ indica que devemos usar um laço para achar algum valor de i para o qual a condição seja verdadeira. A conexão entre laços e quantificadores é também óbvia para quantificadores tais como **Soma**, que nos indica que devemos somar uma sequência de elementos.

Desta forma, nossa primeira sugestão no desenvolvimento de laços é examinar a pós-condição para cada passo e ver se ele contém ou não um quantificador. Uma segunda sugestão vem implicitamente de estados em que devemos executar alguma tarefa para todos elementos na entrada. Desta forma, um laço para percorrer por todos os elementos na entrada é subentendido.

Dado que reconhecemos a necessidade de um laço, como podemos projeta-lo? É fundamental para o nosso processo de desenvolvimento reconhecer o objetivo do laço. Como é sempre o caso em nossas soluções computacionais, este objetivo é dado pela pós-condição para o passo do algoritmo que contém o laço. Isto é, quando o laço termina nosso programa deve estar em um estado descrito pela pós-condição. Desta forma o primeiro passo no desenvolvimento do laço é o desenvolvimento da pós-condição, mas este é o método que já estávamos utilizando.

Para cada laço devemos desenvolver três partes que, combinadas, irão satisfazer o objetivo e garantir o progresso para a finalização. Estas três partes são:

- A(s) instrução(ões) de inicialização
- A condição de finalização
- A(s) ação(ões) executada(s) dentro de laço

O propósito da(s) instrução(ões) de inicialização é estabelecer as condições de entrada no laço. Normalmente essas condições estão associadas a condição de finalização e as ações dentro do laço. Exemplos típicos são encontrados na necessidade de inicializar uma variável contadora ou somadora com 0 antes de começar o laço, lendo o primeiro valor de um fluxo de dados de entrada, etc. Poderíamos escrever pré- e pós-condições formais para inicialização de laços, mas usualmente é suficiente considerar o laço inteiro como uma estrutura na qual a instrução de inicialização é a primeira parte.

O segundo elemento do laço é a condição de finalização. No laço `while`, a iteração das ações do laço continua até que a instrução condicional `b` seja falsa (embora isto implique que deveremos chamá-la de "condição de permanência" que não é uma terminologia padrão). Isto implica que devemos escolher nossa condição de finalização para assegurar que todos os dados são processados de acordo com a pós-condição. De fato, como sabemos que a condição de finalização é `False` na saída do laço, a negação da condição de finalização (esta negação será verdadeira desde que a condição seja `False`) deve ser parte da nossa pós-condição.

A instrução `while` permite que qualquer repetição de um grupo de instruções seja escrito apenas uma vez. Sua forma geral é dada por:

```
expressões de inicialização
while b:
    s1
    s2
   ...
    sn
```

onde `b` é uma expressão condicional qualquer e `s1, s2,.., sn` é uma sequência qualquer de instruções. Como descrito acima, as expressões de inicialização garantem o início correto do laço.

Quando usamos a instrução `while` em um laço, temos que tomar certos passos iniciais para que o laço permaneça sob controle. As condições necessárias para um laço funcione corretamente são que as instruções de inicialização, expressão condicional `b`, e as instruções

`s1, s2, ..., sn` sejam escritas de tal forma que (1) o término apropriado do laço seja garantido e (2) a pós-condição do laço seja satisfeita. Isto é, a execução o laço deve sempre terminar após um número finito de passos, e após o término, o laço deve satisfazer sua pós-condição. Por esta razão, esses tipos de laços são geralmente chamados de laços controlados.

Quando a instrução `while` é executada, a instrução condicional `b` é avaliada primeiro. Se o resultado for `True`, a sequência `s1, s2, ..., sn` será executada uma vez. Então a instrução condicional `b` é avaliada novamente; se for `True`, a sequência `s1, s2, ..., sn` é repetida mais uma vez. Este par de ações é repetido até o teste revelar que a expressão condicional `b` seja `False`, quando isso ocorre, a repetição é finalizada e a instrução seguinte ao final do bloco da instrução `while` é executada. Por exemplo, no problema da soma de `1+2+3+4+5+6` temos:

```
# pós: soma == Soma i em {1,...,6}: i
```

que sugere que o laço seja executado de `1` até `6`. Então, nossa condição de finalização na instrução `while` é:

```
...
soma = 0
numero = 1
while numero <= 6:
   soma = soma + numero
   numero = numero + 1
...
```

Quando sabemos o número de vezes que um laço será repetido, podemos usar a instrução `for`, que é uma forma restrita da instrução `while`. Na discussão dos quantificadores, vimos muitos exemplos do laço `for`. A solução do problema anterior pode ser escrita usando um laço `for` da seguinte maneira:

```
...
soma = 0
for numero in range(1,7):
   soma = soma + numero
...
```

Note que tanto no laço `while` como no laço `for` a instrução `numero <= 6` descreve exatamente a condição sobre a qual o laço continua sua repetição para realizar a equivalente da soma dos números `1,...,6`. Nesses casos, a variável `numero` está sendo usada para controlar o número de vezes que os laços serão repetidos.

No problema de computar o número caracteres de uma dada palavra, usamos a função `len()`. Uma outra forma de resolver esse problema é usando um laço e contar cada um dos caracteres da palavra. Nesse caso, o laço será executado enquanto houver caracteres ainda não "contados" na palavra:

```
...
while (existirem caracteres ainda não contados na palavra):
   ...
```

e saímos do laço quando não houver mais caracteres a serem contados.

A(s) ação(ões) que serão repetidas dentro do laço tem um propósito duplo; elas devem fazer progresso na execução do objetivo especificado pela pós-condição do laço, e elas devem assegurar progresso para encontrar a condição terminação do laço. No caso da soma dos números, a `soma` e o `numero` são os objetivos do laço e `numero` também garante o progresso para a finalização do laço.

Para desenvolver um laço para resolver um problema, devemos atender o objetivo do laço (a pós-condição), a inicialização do laço, as ações executadas dentro laço e como o laço irá terminar. Observamos que as ações dentro do laço são diretamente responsáveis para assegurar o término correto do laço. A necessidade de manipular esses quatro elementos é o que faz com que os laços sejam mais complexos que a manipulação de expressões condicionais. O nosso trabalho será facilitado com a utilização das pré- e pós-condições e afirmações que podem ser expressas usando lógica matemática.

Usando laços para controlar o fluxo de um programa, podemos realizar uma economia e generalizar blocos de instruções que de outro modo seria difícil ou impossível realizar. Vamos ilustrar isso de uma maneira mais detalhada olhando os passos iterativos no problema da soma de `1 + 2 + 3 + ...+ n`, de calcular a potência de um número inteiro e computar o número de caracteres diferentes do espaço numa frase.

3 Solucionando Problemas Algoritmos com Repetição

As três estruturas de controle, sequencial, seleção e repetição são usadas nesta disciplina para projetar algoritmos para solucionar problemas usando um computador. Agora que introduzimos a estrutura de controle de repetição, vamos utilizá-la nas próximas aulas para resolver novos problemas cuja solução não era possível, ou que a solução havia ficado limitada em função do tamanho da entrada.

Vamos agora abordar problemas que necessitam da estrutura de controle de repetição para serem solucionados. Na realidade, esses problemas usam as três estruturas de controle (sequencial, seleção e repetição), pois na estrutura de repetição, o controle do laço necessita de uma estrutura condicional.

3.1 Somando números

Quando o famoso matemático Carl Friedrich Gauss era um jovem estudante, seu professor deu como um exercício somar os 100 primeiros positivos inteiros `1, 2, 3,..., 100`. Qual é o valor desta soma,

$$1 + 2 + 3 + \cdots + 100 = ?$$

Gauss resolveu o problema rapidamente idealizando a fórmula de uma progressão aritmética (`1 + 100 == 101, 2 + 99 == 101,....`).

Considerando que o computador pode efetuar repetidamente (e rapidamente) um bloco de instruções, vamos projetar uma solução que não usa a fórmula de uma progressão aritmética, pois o objetivo nesse exercício é ilustrar como a estrutura de controle de repetição funciona. Inicialmente, vamos entender melhor o problema de calcular a soma de `1 + 2 + 3 + ... + 100`.

```
          0 # inicialização da soma
    +     1
---------
          1 # soma
```

```
   +    2
---------
        3 # soma
   +    3
---------
        6 # soma
   +    4
---------
       10 # soma
   +    5
---------
       15 # soma
        .
        .
        .
```

onde podemos ver que o procedimento envolve duas operações:

1. Um contador que é incrementado por um a cada passo.
2. A soma dos inteiros de 1 até o contador.

```
        0   # inicialização da soma
   +    1   # inicialização do contador
---------
        1   # soma = soma + contador
   +    2   # contador = contador + 1
---------
        3    # soma = soma + contador
   +    3   # contador = contador + 1
---------
        6   # soma = soma + contador
   +    4    # contador = contador + 1
---------
       10    # soma = soma + contador
        .
        .
        .
```

Vemos no exemplo acima que repetidamente um valor de um `contador` é incrementado, e seu valor adicionado de forma cumulativa. Em função isso, necessitamos que a linguagem possibilite escrever uma ou um grupo de instruções que possam ser executadas repetidamente, mas com uma interpretação diferente para cada repetição. Esta característica é providenciada pelas instruções `for` e `while` na linguagem Python, ambas permitem que qualquer instrução ou grupo de instruções sejam repetidamente executadas, mas com uma interpretação diferente a cada repetição. Isto é o que precisamos para simplificar a construção deselegante para computar a soma dos valores de 1 a 100.

Para resolver esse problema usando a "força bruta" (e não usando a técnica eficiente descoberta por Gauss), temos que a descrição do problema é direta. Vamos generalizar e calcular a soma de

$$1 + 2 + 3 + \cdots + n$$

para qualquer inteiro positivo n, e neste caso temos que a descrição do problema fica:

```
# Este programa lê um número inteiro positivo, computa e imprime a
# soma dos números inteiros maiores ou iguais a 1 e menores ou iguais
# ao número lido. soma =  1 + 2 + 3 + ... + n
```

As especificações de entrada e saída do problema são:

Entrada	**Saída**
`n` (último número)	Valor de `1 + 2 + 3 + .. + n`

Para este problema, a entrada consiste dos inteiros positivos `n`, `contador` para gerar os números, e `soma` para armazenar o valor da soma dos números `1 + 2 + ... + n`.

```
# descrição das variáveis utilizadas
# int n, contador, soma
```

e as pré e pós-condições ficam:

```
# pré: n

# pós: soma = Soma i em {1,...,n}: i
```

Vamos agora descrever a subdivisão.

```
# Algoritmo: somador
# Passo 1. Leia o último número n
# Passo 2. Compute a soma de 1 +..+ n
# Passo 3. Imprima a soma
# fim algoritmo
```

Necessitamos refinar o Passo 2. Como vimos acima, para computarmos a soma de 1 a `n`, começamos com 1 como o valor de `contador` e com 0 como o valor inicial da `soma`. A cada passo, o valor do `contador` é adicionado a `soma`, atribuindo um novo valor a `soma`, e o valor do `contador` é incrementado por 1. Estes passos são repetidos até que tenhamos somado todos os números entre 1 e `n`. Ou seja, a variável `contador` inicia com o valor 1 e repetidamente vai aumentado em uma unidade. A variável `soma` inicia com o valor 0 e repetidamente soma o valor de `contador`.

```
contador = 1              # contador == 1 - inicialização do contador
soma = 0                  # soma == 0 - inicialização da soma
i = 1
soma = soma + contador    # soma == 0 + 1
contador = contador + 1   # contador == 1 + 1
i = 2                     # i = i + 1
```

```
soma = soma + contador    # soma == 1 + 2
contador = contador + 1   # contador == 2 + 1
i = 3                     # i = i + 1
soma = soma + contador    # soma == 3 + 3
contador = contador + 1   # contador == 3 + 1
i = 4                     # i = i + 1
soma = soma + contador    # soma == 6 + 4
contador = contador + 1   # contador == 3 + 1
...
i = n
soma = soma + contador    # soma == soma + n
contador = contador + 1   # contador == n + 1
# contador >  n  - Finalize!
,,,
```

onde a variável `i` conta o número de vezes que o bloco

```
contador = contador + 1
soma = soma + contador
```

repete. No caso da variável `contador`, ela gera repetidamente os números `1,.., n`, que coincide com o número de vezes que o bloco acima é executado. Esse tipo de variável é denominada como uma variável contadora. Já a variável `soma`, adiciona, cumulativamente, os valores da variável `contador`. Denominamos esse tipo de variável como acumuladora (somadora). Quando o valor de `contador` excede `n`, a computação do laço é interrompida e o valor da variável `soma` é a resposta desejada. Usando a definição do quantificador **Soma** descrito acima (e as pós condições) temos que:

```
soma == Soma i em {1,...,n}:i
```

e a descrição dessa estrutura de repetição usando a instrução `while` simplifica a descrição da soma de `1 + 2 + ...+ n` e pode ser feita como:

```
while contador <= n:
   a. Adicione numero a soma
   b. Incremente contador por 1
```

onde as instruções `a` e `b` são repetidas enquanto a condição `contador <= n` permanecer verdadeira. Desta forma, quando o valor de `contador` exceder o de `n`, a repetição é finalizada e a próxima instrução após o bloco `while` sera executada Com isso, podemos descrever o Passo 2 como:

```
# Passo 2. Compute a soma de 1 +..+ n
# Passo 2.1. Inicialize as variáveis contador e soma
# Passo 2.2. Enquanto contador <= n
# Passo 2.2.1.  Adicione contador a soma
# Passo 2.2.2.  Incremente contador por 1
```

sendo que no nosso exemplo, as instruções de inicialização são `contador = 1`, `soma = 0`, e o valor de `n`, obtido anteriormente no Passo 1 do algoritmo. A expressão condicional é dada por `contador < = n`, e as instruções do bloco de repetição são `contador = contador + 1` e `soma = soma + contador`. Desta forma, o Passo 2 pode ser implementado como o laço controlado em Python da seguinte forma:

```
# Passo 2. Compute a soma de 1 +..+ n
# Passo 2.1. Inicialize as variáveis contador e soma
contador = 1
soma = 0
# Passo 2.2. Enquanto contador <= n
while contador <= n:
# Passo 2.2.1.  Adicione contador a soma
   soma = soma + numero
# Passo 2.2.2.  Incremente contador por 1
   contador = contador + 1
```

Quando for conhecido o número de vezes que um laço será repetido, podemos usar a instrução `for`, que é uma forma restrita da instrução `while`. Na discussão dos quantificadores, vimos muitos exemplos do laço `for`. Nesse caso, o Passo 2 pode ser implementado em Python usando um laço `for` da seguinte maneira:

```
# Passo 2. Compute a soma de 1 +..+ n
# Passo 2.1. Inicialize a variável soma
soma = 0
# Passo 2.2. Para contador em [1,n]
for contador in range(1,n+1):
# Passo 2.2.1.  Adicione contador a soma
   soma = soma + contador
```

Note tanto no laço `while` como no laço `for` as instruções `contador <= n` e numero in range(1,n+1), descrevem exatamente a condição sobre a qual o laço continua sua repetição para realizar a soma das n notas.

O tipo de generalização realizado por introduzir a variável `n` permite-nos resolver o problema da soma de uma sequência contígua de inteiros para qualquer intervalo. Sem o `n`, a solução estaria sempre tendendo a computar a soma de um número fixo de números. Isto seria uma solução bem menos satisfatória para este problema e para a maior parte das situações de solução de problemas em geral, pois não satisfaz a característica geral de um algoritmo. Podemos generalizar ainda mais e ler o valor inicial da sequência. Nesse caso, `numero` seria inicializado com esse valor.

Uma implementação em Python usando `while` para o problema é dada a seguir.

```
# -*- coding: utf-8 -*-
# Programa: somador.py
# Programador:
# Data: 08/03/2016
# Este algoritmo lê n (último número) e calcula o valor da soma
#        1 + 2 + 3 + ... + n
# para algum inteiro positivo n. Ele usa a variável numero como
# um contador e a variável soma para sua soma.
```

```
# início do módulo principal
# descrição das variáveis utilizadas
# int n, contador, soma

# pré: n

# Passo 1. Leia o último número
n = int(input())
# Passo 2. Compute a soma de 1 +..+ n
# Passo 2.1. Inicialize as variáveis contador e soma
contador = 1
soma = 0
# Passo 2.2. Enquanto contador <= n
while contador <= n :
# Passo 2.2.1.  Adicione contador a soma
   soma = soma + contador
# Passo 2.2.2.  Incremente contador por 1
   contador = contador + 1
# Passo 3. Imprima a soma
print(soma)

# pós: soma == Soma i em {1,...,n}: i
# fim do módulo principal
```

Vamos agora generalizar o exemplo anterior. Nosso problema agora é ler dois números inteiros, `inicio` e `fim` e calcular e imprimir a somas dos números inteiros maiores ou iguais a `inicio` e menores ou iguais a `fim`.

No descrição do problema, exploramos os limites computacionais e matemáticos para que este algoritmo funcione corretamente. Por exemplo, podemos generalizá-la para a soma dos números inteiros entre `início` e `fim` onde `início` e `fim` são dois números inteiros. Temos que deixar claro se os números `inicio` e `fim` serão considerados na soma e também descrever a respeito dos casos não usuais, tais como quando `início` (e/ou `fim`) for um número negativo. Para esse problema, vamos considerar apenas valores maiores ou iguais a zero (Naturais). Temos também que analisar qual os valores máximos e mínimos para `início` e `fim`.

```
# Este programa lê dois números inteiros, início e fim, computa e imprime
# a soma da sequência dos números inteiros maiores ou iguais a inicio e
# menores ou iguais a fim.
```

Após a descrição do problema, a próxima fase é o das especificações de entrada e da saída. Relembramos que nessa fase, a partir das descrições do problema desenvolvemos especificações mais precisas a respeito da saída desejada e da entrada necessária para a obtenção da saída. Estas especificações definem mais precisamente o estado conhecido do programa no início de qualquer execução do programa (chamadas especificações de entrada) e qual estado o programa deve ter no final da execução (chamada especificações de saída).

Entrada	**Saída**
dois números inteiros, início, fim.	A soma dos números inteiros entre início e fim, inclusive.

Para este problema, a entrada consiste dos inteiros positivos `inicio`, `fim`, definindo os limites onde será feita a soma, `contador` para contar os números, e `soma` para armazenar o valor da soma dos números `inicio, inicio + 1 + inicio + 2 + ... + fim`. Quando da implementação do algoritmo pode surgir a necessidade da utilizarmos variáveis auxiliares para armazenar valores intermediários.

```
# descrição das variáveis utilizadas
# int inicio, fim, contador, soma
```

e as pré e pós-condições ficam:

```
# pré: inicio fim

# pós: soma == Soma i em {inicio, inicio+1,..., fim}: i
```

Exemplo:

```
# formato da entrada
1 # inicio
100 # fim

# estado das variáveis após a entrada
início == 1
fim == 100

soma = 1 + 2 + 3 + 4 + ... + 97 + 98 + 99 + 100
# estado da variável soma após o cálculo
soma ==  5050

# formato da saída
A soma de 1 a 100 é 5050
```

Uma vez definida a entrada (`pré:`) e a saída (`pós:`), a próxima fase é o **refinamento**. Nessa fase devemos descrever o algoritmo que soluciona o problema em passos elementares. Lembramos que um computador pode ler e imprimir dados, realizar operações aritmética e lógicas e controlar o fluxo dos passos (instruções). Usando o exemplo anterior, os passos do nosso Algoritmo ficam:

```
Algoritmo: Gauss
# Passo 1. Leia o início e o final da sequência
# Passo 2. Calcule a soma dos números entre início e final
# Passo 3. Imprima a Soma
# fim Algoritmo
```

Como destacamos anteriormente, o refinamento do nosso problema deve ser expresso de acordo com as funcionalidades de um computador. A única diferença com relação ao programa `somador.py` é que a entrada e dada por dois números inteiros não negativos, e que a variável `contador` começa com `inicio` e termina com `fim`. Nesse caso, o refinamento do Passo 2 fica

```
# Passo 2. Compute a soma de inicio +..+ fim
# Passo 2.1. Inicialize as variáveis contador e soma
# Passo 2.2. Enquanto contador <= fim
# Passo 2.2.1.  Adicione contador a soma
# Passo 2.2.2.  Incremente contador por 1
```

Após o refinamento do algoritmo, podemos finalizar a Codificação.
numbers

```
# -*- coding: utf-8 -*-
# Programa: gauss.py
# Programador:
# Data: 12/09/2016
# Este programa lê dois números inteiros, início e fim, computa e imprime
# a soma da sequência dos números inteiros maiores ou iguais a inicio e
# menores ou iguais a fim.
# início do módulo principal
# descrição das variáveis utilizadas
# int inicio, fim, soma, contador

# pré: início fim

# Passo 1. Leia o início e o final da sequência de números
inicio = int(input('Entre com o número inicial: '))
fim = int(input('Entre com o número final  : '))
# Passo 2. Calcule a soma dos números entre início e final
# Passo 2.1. Inicialize Soma e contador
soma = 0
contador = inicio
# Passo 2.2. Enquanto contador for menor ou igual a final faça
while contador <= fim:
# Passo 2.2.1. Adicione contador a Soma
   soma = soma + contador
# Passo 2.2.2. Incremente contador com 1
   contador = contador + 1
# Passo 3. Imprima a Soma
print('A soma de {0:d} a {1:d} é {2:d}'.format(inicio, fim, soma))

# pós: soma == Soma i em {inicio,..., fim}: i
# fim do módulo principal
```

Após a interpretação sem erros do programa, o próximo estágio é o Teste e Verificação. Você pode executá-lo (de acordo com o seu sistema operacional) e testar o sue funcionamento. Nesse nosso exemplo, como só vimos a estrutura de controle sequencial, assumimos que o usuário só fornecerá números que sejam válidos. De qualquer forma, teste o seu programa para números bem grandes e para divisor igual a 0 e verifique o que ocorre. Abaixo, um exemplo de uma execução do programa.

Exemplo 1

```
# formato da entrada
Entre com o número inicial: 1
```

```
Entre com o número final  : 100

# formato da saída
A soma de 1 a 100 é 5050
```

Exemplo 2

```
# formato da entrada
Entre com o número inicial: 10
Entre com o número final  : 10

# formato da saída
A soma de 10 a 10 é 10
```

O Python possui várias funções que associadas com o laço `for` possibilitam uma implementação quase "direta" das pós condições. No caso da soma

```
Soma i em {inicio,..., fim}: i
```

usando a função `sum()` com um laço `for` em Python, podemos escreve-la como:

```
sum(i for i in range(inicio,fim+1)
```

e com isso, podemos reescrever o Passo 2 da seguinte forma:

```
1  # -*- coding: utf-8 -*-
2  # Programa: gauss.py
3  # Programador:
4  # Data: 12/09/2016
5  # Este programa lê dois números inteiros, início e fim, computa e imprime
6  # a soma da sequência dos números inteiros maiores ou iguais a inicio e
7  # menores ou iguais a fim.
8  # início do módulo principal
9  # descrição das variáveis utilizadas
10 # int inicio, fim, soma
11
12 # pré: início fim
13
14 # Passo 1. Leia o início e o final da sequência de números
15 inicio = int(input('Entre com o número inicial: '))
16 fim = int(input('Entre com o número final  : '))
17 # Passo 2. Calcule a soma dos números entre início e final
18 soma = sum(i for i in range(inicio,fim+1))
19 # Passo 3. Imprima a Soma
20 print('A soma de {0:d} a {1:d} é {2:d}'.format(inicio, fim, soma))
21
22 # pós: soma == Soma i em {inicio,..., fim}: i
23 # fim do módulo principal
```

Apesar de mais simples, o programa acima exige uma abstração maior (entendimento da função `sum()`. Deixe para usar esses recursos depois que você estiver bem mais familiarizado com os laços `for` e `while`.

3.2 Potência

Vimos no exemplo anterior que a estrutura de controle de repetição pode ser utilizada para somar os números de uma sequência. Agora analisaremos o caso de computar a potência inteira de um dado número real, ou seja multiplicar um número por ele mesmo um dada quantidade de vezes. Dado um número real x e um número inteiro positivo, calcular o valor da potência x^n.

Projetar e implementar um algoritmo para ler os valores de x e n, computar e imprimir o valor de x^n.

Exemplo 1

```
# formato da entrada
2.0
2

# formato da saída
4.0
```

Exemplo 2

```
# formato da entrada
1.1
3

# formato da saída
1.331
```

A descrição deste problema é bem simples:

```
# Este programa lê um número real x e uma potência inteira n >= 0,
# computa o valor de x elevado a n e imprime o resultado.
```

Temos que as especificações de entrada e saída são:

Entrada	**Saída**
um número real x indicando a base e um número inteiro $n > 0$ indicando a potência	o valor da potência de x elevado a n

Nesse exemplo as variáveis são compostas de tipos básicos. Necessitamos do tipo `float` para armazenar os valores das variáveis `numero` e `potencia`, e do tipo`int` para armazenar o valor da variável `pot`. Quando da implementação do algoritmo pode surgir a necessidade da utilizarmos variáveis auxiliares para armazenar os valores intermediários.

Exemplo

```
Leia o valor de número: 2.0
Leia o valor da potência: 2

# estado das variáveis após a leitura
numero == 2.0
pot == 2

# cálculo da potência
potencia = 1.0
potencia = potencia*numero
potencia = potencia*numero

# estado das variáveis após o cálculo da potência
numero == 2.0
pot == 2
potencia == 4.0

# formato da saída
2.0^2 = 4.000000
```

e as pré e pós-condições ficam:

```
# pré: numero pot

# pós: potencia == Prod i em {1,..., pot}: numero
```

Nos passos do Algoritmo desse problema temos um exemplo de uma variável que armazena o produto acumulado. A cada iteração i a variável armazenará o valor de `numero` multiplicado por `numero` i vezes.

```
Algoritmo: Potência
# Passo 1. Leia os valores de numero e pot e inicialize potência
# Passo 1.1. Leia o valor do número
# Passo 1.2. Leia o valor do expoente
# Passo 2. Calcule o valor da potência
# Passo 3. Imprima o valor da potência
# fim do algoritmo
```

Temos que refinar o Passo 2. Como sabemos o número de vezes que a multiplicação será efetuada, podemos projetar a nossa solução com um laço `for`:

```
# Passo 2. Calcule o valor da potência
# Passo 2.1. Inicialize o valor de potência
# Passo 2.2. Para i no intervalo [1,pot]
# Passo 2.2.1. Multiplique potencia por numero
```

Com isso, uma implementação em Python do nosso algoritmo fica:

```
# -*- coding: utf-8 -*-
# Programa: potencia.py
# Programador:
# Data: 23/03/2016
# Este programa lê um número real x e uma potência inteira n >= 0,
# computa o valor de x elevado a n e imprime o resultado.
# início do módulo principal
# descrição das variáveis utilizadas
# float numero, potencia
# int pot

# pré: numero pot
# Passo 1. Leia os valores de numero e pot e inicialize potência
# Passo 1.1. Leia o valor do número
numero = float(input('Leia o valor de numero: '))
# Passo 1.2. Leia o valor do expoente
pot = int(input('Leia o valor da potência: '))
# Passo 2. Calcule o valor da potência
# Passo 2.1. Inicialize o valor de potência
potencia = 1
# Passo 2.2. Para i no intervalo [1,pot]
for i in range(0,pot):
# Passo 2.2.1. Multiplique potencia por numero
   potencia = potencia*numero
# Passo 3. Imprima o valor da potência
print('{0:f}^{1:d} = {2:f}'.format(numero, pot, potencia))

# pós: potencia == Prod i em {1,..., pot}: numero
# fim do módulo principal
```

Usando uma IDE, implemente e execute o programa para vários valores de `numero` e `pot`. A potência inteira de um número real pode ser calculado em Python usando a instrução `**`, no nosso caso:

```
potencia = numero**pot
```

Na sua implementação, compare os dois resultados.

3.3 Computando o numero de caracteres de uma string

Utilizando a estrutura de dados string, podemos resolver vários problemas que envolvem o análise de caracteres tais como computar o número de caracteres duplos de um dado texto, buscar uma dada palavra num texto, formatar um texto, contar o número de caracteres de um texto, etc. As linguagens de programação dispões de funções e métodos para manipular strings. Vamos abordar alguns problemas para entender melhor como essas funções e módulos funcionam.

Para computar o tamanho de uma palavra nos exemplos e exercícios que tratamos anteriormente sobre strings, utilizamos a função `len()` no Python. Agora que temos mais informações sobre strings, iremos projetar um algoritmo e um programa para computar

o número de caracteres visíveis, diferentes do caracteres espaço de uma dada string, sem utilizar a função `len()`). Considere um texto como uma sequência de caracteres visíveis (pode conter espaços em branco) e representada por uma variável do tipo abstrato de dados `string`. No problema, a entrada é dada por um texto formado por caracteres visíveis e a saída consiste em escrever para o texto lido o respectivo número de caracteres visíveis, excetuando os caracteres espaço.

Exemplo 1

```
# formato da entrada
algoritmos dados

# formato da saída
15
```

Exemplo 2

```
# formato da entrada
estrutura e python

# formato da saída
16
```

```
# -*- coding: utf-8 -*-
# Programa: tamanho.py
# Programador:
# Data: 08/07/2016
# Este programa lê um texto, conjunto de no máximo 40
# caracteres visíveis contíguos, calcula e imprime o número de
# de caracteres diferentes do espaço na palavra.
# início do módulo principal
# descrição das variáveis utilizadas
# int    tamanho
# string texto, caractere

# pré: palavra

# Passo 1. Leia um texto
texto = input()
# Passo 2. Compute o tamanho do texto (sem espaços)
# Passo 2.1. Inicialize a quantidade de caracteres
tamanho = 0
# Passo 2.3. Conte os caracteres do texto diferentes de espaço
for caractere in texto:
   if caractere != ' '
      tamanho = tamanho + 1;
# Passo 3. Imprima o número de caracteres do texto
print(tamanho)

```

```
27 # pós: tamanho and tamanho == len(texto)-texto.count(' ')
28 # fim do módulo principal
```

Podemos utilizar um recurso da linguagem `Python` para simplificar o passo 2.

```
# Passo 2. Compute o tamanho da palavra
tamanho = sum(1 for caractere in texto if letra != ' ')
```