Universidade de São Paulo

Instituto de Matemática e Estatística

DCC – Leônidas O. Brandão

leo@ime.usp.br — http://www.ime.usp.br/~leo

Intrudução a Resolução de Problemas via Algoritmos

Em disciplinas como MAC-110 (licenciatura IME) e MAC-115 (engenharias), o objetivo central é iniciar o aluno na "arte de resolução de problemas via computador", mais especificamente: **como escrever algoritmos**. Um número considerável de alunos, nestas disciplinas, acabam não conseguindo aproveitamento satisfatório e acreditam que o problema: foi o computador; seu desconhecimento prévio no uso do computador; e coisas do genero. Na verdade, o problema está longe de "residir" no computador, a maior dificuldade da disciplina está em um conceito muito mais antigo que o computador: os algoritmos. Não se esqueça, sua maior preocupação nesta disciplina é: como resolver um problema via algoritmo.

1 Introdução

O objetivo básico desta apostila é introduzir o conceito de algoritmos (seção seguinte) e principalmente, apresentar algumas "dicas" sobre como escrever um algoritmo a partir de um problema dado. Para isto, na seção 2, formalizaremos um conjunto de **comandos** e regras sintáticas (e semânticas), na forma de uma **linguagem**, que denominaremos Portugol. Através desta linguagem deduziremos algoritmos a partir de alguns problemas/exemplos.

Esta dedução será construtiva: começaremos analisando casos particulares e só então generalizaremos na forma de um algoritmo. E como uma técnica auxiliar, propomos ao programador iniciante que tente construir seus algoritmos a partir de quatro questões básicas, como descritas na seção 3.

1.1 Intuição sobre o que é um Algoritmo

O conceito de algoritmo é muito mais antigo que os computadores digitais, existindo desde os primórdios da Matemática. Exemplos "arqueológicos" destes são: os algoritmos da soma, da multiplicação e da divisão de dois números (vide a seguir o exemplo da soma). Outro algoritmo muito antigo, é o algoritmo de Euclides (de Alexandria[1] [365 a.C-300 a.C]) para computar o máximo divisor comum entre dois números inteiros (vide seção 4).

Resumidamente, **algoritmo** é uma sequência de passos elementares, executados um após o outro (de cima para baixo - este é o **fluxo de execução**). Em geral, um algoritmo deve ser "aplicado" sobre um conjunto de "valores" (**dados de entrada**) para produzir um conjunto de "valores" como resposta. Uma característica que um algoritmo deve apresentar é ser determinístico: sempre que este for aplicado sobre o mesmo conjunto de entradas deverá produzir sempre a mesma saída.

No padrão de algoritmos que consideraremos, aparecem três classes de comandos:

- atribuição/variável: usados para armazenar valores e resultados aritméticos;
- comandos de seleção: o método mais simples para desviar o fluxo de execução;
- repetição ou laço: outro desvio de fluxo, desta vez para repetir comandos.

[1] Euclides foi um grande matemático grego, nascido em Alexandria (que na época fazia parte do império Grego e hoje faz parte do Egito), sua obra mais famosa é o clássico *Os Elementos*, que ainda hoje é estudado.

1.2 Um primeiro exemplo de Algoritmo

Vamos iniciar a discussão de algoritmo, com um exemplo que todos já conhecem (embora não como sendo algoritmo): o problema da soma de dois números (com vários dígitos).

Problema 1 *Dados dois números, representados pelas variáveis x e y, como você descreveria um algoritmo para alguém (que não sabe somar) obter a soma destes dois?*

Antes de prosseguir com a leitura, tente esquematizar soluções para este desafio, em duas etapas: Inicialmente, sem muito formalismo, como você explicaria verbalmente o processo para este alguém? E se você tiver que deixar por escrito (nem mesmo sabe que será o "usuário" de sua descrição), como faria?

Para a solução "escrita", vide sugestão a seguir.

Sugestão: Admita que os números sejam representados por `x` e `y` ("variáveis) e represente o i-ésimo dígito de `x` e de `y`, respectivamente, por $\mathtt{x}_i$ e $\mathtt{y}_i$. Utilize uma nova variável para realizar o "vai um" e denomíne-a por `vai_um`. (Guarde sua solução e após a leitura desta primeira seção, compare-a com a que apresentamos na sub-seção 7.1 - página 19).

A maior dificuldade em resolver "algoritmicamente" determinado problema é determinar o "bloco de repetição", isto é, o conjunto de comandos (ou intruções) a serem executadas repetidas vezes. No exemplo, da soma teríamos neste bloco, os seguintes comandos: a soma dos dígitos $\mathtt{x}_i$, $\mathtt{y}_i$ e `vai_um`; também teríamos a redefinição da variável `vai_um` (como seria isto?).

Assim, o que examinaremos aqui, será basicamente como detectar as variáveis necessárias ao algoritmo e como determinar o bloco de repetições. Faremos isto de modo contrutivo, a partir de exemplos particulares, como se fossemos resolver o problema manualmente e só então, generalizamos o processo para a obtenção de um algoritmo.

2 Comandos PORTUGOL

Nesta seção, apresentaremos brevemente, os comandos e regras (sintáticas e semânticas) que utilizaremos na descrição dos algoritmos. Denominaremos o conjuntos destas regras por **linguagem** PORTUGOL.

O nome "linguagem" advém da semelhanças destas regras com as de uma linguagem natural qualquer (evidentemente, nosso PORTUGOL, será bem mais "pobre"). Já o nome PORTUGOL, é devido aos identificadores (nomes) de comandos estarem todos em Português.

Um conceito muito importante à programação é o de **variável**, que servem para armazenar valores a serem utilizados pelo **progama** (implementação do algoritmo num computador). Uma variável é um nome (identificador) que deve ser associado à uma posição única na memória do computador (na "Random Access Memory" - RAM).

Intuitivamente, pode-se pensar que variável é o nome de uma gaveta utilizada para armazenar valores: sempre que fizer uma atribuição a esta variável, o valor atribuido será armazenado na gaveta correspondente.

- **Declaração de Variáveis:** Uma variável poderá ser de dois tipos básicos, `inteiro` ou `real`,

  `inteiro i,j`: Declara as variáveis `i` e `j` aptas para armazenarem (apenas) inteiros.

- **Atribuições:** Permite atribuirmos valores (fixo ou resultado de uma expressão aritmética) às variáveis

  `x` ← `EXPR`: A variável `x` recebe o valor associado à expressão aritmética `EXPR`.

  Se `x` armazena inteiro e `EXPR` resulta real, então o valor atribuído à `x` será truncado, por exemplo, `x` ← `5/2`, implica que `x` receberá o valor `2`.

Outro exemplo importante de atribuição, é o incremento de variável `x` ← `x+1` (ou `x*c`...), que deve ser lido como: "`x` passa a receber o valor armazenado `x` mais `1`".

- **Comandos:** Apresentamos a seguir a sintaxe e semântica dos comandos do PORTUGOL. Tudo o que estiver entre os símbolos de colchêtes, [e], são opcionais, o que significa que estes símbolos não devem aparecer no código final: se deseja usar a opção o faça omitindos estes símbolos.

  - **Seleção:** Utilizado para (eventualmente) "desviar" o fluxo de execução do programa. Quando for necessário colocar mais que um comando (bloco de comandos) sob o controle de um comando `se`, deve-se utilizar os símbolos "abre" e "fecha parêntese", **{** e **}**, "envolvendo" o bloco de comandos.

`se (COND) comandos_1` `[senão comandos_2]`	Se `COND` (uma condição lógica) for verdadeira, execute o (cojunto de) comando(s) `comandos_1`, caso contrário, se existir o `senão`, execute `comandos_2`.

    Nos exemplos a seguir, o fluxo de execução é desviado de acordo com os valores das variáveis (`x` no primeiro e `a` e `b` no segundo).

    Exemplos: 1. `se (x=10) escreva('10')`
    2. `se (a=b) escreva(a,' diferente de ',b)`
    `senão escreva(a,' igual a ',b)`

  - **Repetição:** Comandos utilizados para controlar os blocos de repetição. Novamente, ser for necessário utilizar mais que um comando, subordinado ao `repita` ou ao `enquanto`, deverá ser utilizados os símbolos { e } "envolvendo" o bloco de comandos.

`enquanto (COND)` `  comandos`	1. Se `COND` for verdadeira, execute o (cojunto de) comando(s) `comandos`; 2. Volte ao passo 1 (note que este laço só termina quando ocorrer um `COND` falso.)

`repita`
`  comandos`
`enquanto (COND)`
	1. Execute o (conjunto de) comando(s) `comandos`;
	2. Se `COND` for verdadeira, volte ao passo 1

    Já nestes comandos os desvios de fluxo de execução são "maiores", por exemplo, no primeiro caso: Se `COND` for verdadeiro, serão executados todos os comandos em `comandos` (**bloco de repetição**) e a seguir a execução voltará para o teste de entrada (**laço**). Se `COND` for falso, todos os comandos em `comandos` serão ignorados e será executado o próximo comando após o final do bloco de repetições.

    Observe atentamente a sutil diferença entre estes dois comandos de repetição: o primeiro é um laço com condição de entrada e o segundo um laço com condição de continuação. Praticamente todas as linguagens de programação apresentam um comando equivalente ao primeiro, já o segundo pode sofrer pequenas alterações.

  - **Entrada/Saída:** Estes são comandos essenciais para que o **usuário** (pessoa que está "utilizando" o algoritmo/programa) possa, interagir com o mesmo.

`leia(lista_variáveis)`:	Para **entrada de dados**, o usuário deve digitar tantos valores quanto demandado pela lista de variáveis;
`escreva(lista_variáveis)`:	Para **saída de dados**, o computador (no qual será implementado o algoritmo/programa) deve listar os valores das variáveis da lista. Por exemplo,

    Exemplo: Programa com duas variáveis, que permite que o usuário digite dois valores e depois estes são impressos no **dispositivo de saída padrão** (usualmente o monitor de vídeo).

`inteiro x,y;`
`leia(x,y)`
`imprima(x,y)`
	o usuário deverá digitar dois valores inteiros (separados por espaços em branco), para serem atribuídos às variáveis `x` e `y`.

3 Perguntas Básicas

No restante desta apostila procuraremos mostrar como desenvolver programas utilizando a linguagem estruturada que acabamos de descrever. Os problemas que abordaremos são, em sua maioria, bastante simples, no sentido de ser possível resolvê-los manualmente (sem o auxílio de programas de computador). A dificuldade consiste em obter soluções computacionais para tais problemas, ou seja, em escrever um processo que resolva tais problemas servindo-se de uma linguagem extremamente rígida e um pouco limitada, como é o caso das linguagens de programação (PORTUGOL, C, PASCAL, $\cdots$).

Um **programa** pode ser entendido como uma sequência finita de comandos que manipulam um número limitado de variáveis e que deve devolver algum conjunto-resposta. Em geral, os problemas que admitem soluções via computador apresentam características de repetição, isto é, permitem que uma mesma sequência de passos seja repetida (um número finito de vezes) até que se obtenha o resultado desejado.

Nesta apostila, sugerimos ao leitor iniciante[2] na "arte de programar", que se concentre em quatro questões chaves à resolução de problemas via algoritmos, independente da linguagem a ser utilizada (seja ela o PORTUGOL, ou linguagens "comerciais" como C e PASCAL):

1^a **Pergunta**: Quais variáveis são necessárias?

2^a **Pergunta**: Quais comandos devem aparecer dentro de um bloco de repetição?

3^a **Pergunta**: Qual a condição para que continue a repetição?

4^a **Pergunta**: Quais os comandos necessários antes e depois do bloco de repetição?

Nem sempre é possível responder completamente estas perguntas nesta sequência. Muitas vezes, ao longo da descrição do programa surge a necessidade de novas variáveis, que devem ser acrescentadas à lista inicialmente pensada como resposta à 1^a pergunta. Por outro lado, com o passar do tempo (e aquisição de experiência na "arte de programar"), será natural que ao tentar responder a 2^a pergunta já se note a necessidade de alguns comandos que devem aparecer antes do bloco de repetição (4^a pergunta). Além disto, somente "pequenos problemas" possuem um único bloco de repetição, sendo que as perguntas 2 a 4 devem ser levadas em conta para cada novo bloco de repetição. No entanto, a experiência tem-nos mostrado que, pelo menos enquanto se está começando a aprender a descrever processos em liguagens estruturadas, estas perguntas servem como um bom roteiro ao programador iniciante.

Veremos, a seguir, o desenvolvimento de programas para alguns problemas.

3.1 Como somar um número arbitrário de valores

Problema 2 *Faça um programa que leia uma sequência de números inteiros, diferentes de zero, e calcule a sua soma.*

Se a sequência de entrada tivesse um tamanho fixo conhecido, digamos 4, a descrição do processo não seria complicada, bastando escrever

```
leia(a,b,c,d)
escreva(a+b+c+d)
```

[2] Mesmo um programador experiente responde, ao menos inconcientemente, estas questões durante seu processo de "construção" de algoritmos.

no entanto, o panorama muda radicalmente se desejamos escrever um algoritmo que funcione para um número arbitrário de valores de entrada (este número é definido pelo usuário, na "execução" da implementação do algoritmo). Neste caso, a dificuldade reside no fato de o programa ser capaz de somar uma sequência de qualquer tamanho.

Antes de continuar, reflita um pouco sobre esta questão chave à programação.

Uma boa técnica para iniciar o processo de programação é admitir algumas sequências de dados de entrada e pensar a partir destes dados. Então examinemos, por exemplo, a soma da seguinte sequência de entrada: `8 11 3 -7 2` $\cdots$ `0`

1. Leia o primeiro número (8);
2. Como (número lido $\neq 0$), leia o segundo número (11),
   guardando a soma destes dois (19);
3. Como (último número lido $\neq 0$), leia o próximo número (3),
   guardando com a soma anterior $(19 + 3 = 22)$;
4. Como (último número lido $\neq 0$), leia o próximo número (-7),
   guardando com a soma anterior $(22 + (-7) = 15)$;

$\vdots$

Versão 1: Sequência de passos quando usamos `8 11 3 -7 2` $\cdots$ `0`

Observando a versão 1 já podemos responder às três primeiras perguntas

Quais Variáveis são necessárias?

Vemos a necessidade de 2 variáveis: uma para guardar o último número lido e outra para guardar a soma dos números já lidos. Então, na linguagem estruturada, teríamos

```
inteiro num, soma
```

Quais comandos devem aparecer dentro de um bloco de repetição?

Podemos notar que, devem ser repetidas a leitura do próximo número da sequência e a atualização da soma. Para a leitura, basta utilizar o comando `leia(num)`, que pegará o próximo número da sequência, armazenando-o na variável `num`. Para a atualização da `soma` deveremos usar o comando de atribuição armazenando na variável `soma` o resultado da soma acumulada com o número lido: `soma` $\leftarrow$ `soma + num`.

Na linguagem estruturada Portugol, teremos:

```
1 enquanto(...)  {
2   leia(num)
3   soma ← soma + num }
```

Versão 2: Estrutura do bloco de repetição (laço)

Note que a ordem dos comandos é muito importante: imagine esta sequência, trocando-se os comandos das linhas `2` e `3` entre si.

Qual a condição para que se continue a repetição?

Como o programa deve parar ao ler o número zero, concluímos que o bloco de repetição deve ser executado enquanto o número lido for diferente de zero. Ou seja

```
enquanto(num ≠ 0) {
  leia(num)
  soma ← soma + num }
```

Versão 3: Laço com controle de entrada definido

Resta ainda respondermos à 4^a pergunta,

Quais os comandos necessários antes e depois do bloco de repetição?

Inicialmente, o valor das variáveis (quaisquer) está indefinido e portanto devemos cuidar das "inicializações" antes do primeiro teste `enquanto(num ≠ 0)`. Quanto deve valer a variável `num` neste momento inicial?

Uma solução possível é o tratamento do primeiro número da sequência como caso à parte, antes de entrarmos no comando de repetição. Deste modo, devemos ler o primeiro número fora da repetição e já o contabilizar na soma, como indicado abaixo:

```
leia(num)
soma ← num
⋮
```

Exercício 1 *Tente elaborar outra solução, que não faça um primeira leitura de* `num` *dentro do laço.*

Após a repetição devemos imprimir o conteúdo da variável `soma`, que deverá conter a soma dos elementos da sequência de entrada (isto se até agora, não cometemos algum erro na dedução do algoritmo - o que ocorre com razoável frequência na programação). Isto pode ser feito com o comando `escreva(soma)`.

Agora já podemos dispor de um programa completo,

```
inteiro num, soma
leia(num)
soma ← num
enquanto(num ≠ 0) {
  leia(num)
  soma ← soma + num }
escreva(soma)
```

Versão 4: Programa final para o problema 2, em PORTUGOL

Conceito de Simulação de um Algoritmo

Um conceito muito importante à programação é o processo de **simulação** de um algoritmo. Útil na verificação da corretude do mesmo e também para entender o seu funcionamento. Na simulação de um algoritmo/programa devemos anotar todas as alterações, de todas suas variáveis, ao longo de seu fluxo de execução para algum conjunto de dados de entrada.

Melhor que muitas explicações complicadas é um bom exemplo. Vejamos a simulação do programa acima para o conjunto de entradas `10 3 4 0`,

num	soma	comando em execução	observações
?	?		no início do programa valores das variáveis são desconhecidos
10		`leia(num)`	pega o 1^o número da entrada e guarda em `num`
	10	`soma ← num`	atribui o valor de `num` a `soma`
		`enquanto(num ≠ 0)`	como `num` $\neq 0$, o programa executará os comandos que estão dentro do bloco de repetição (linhas `5` e `6`)
3		`leia(num)`	usuário deve digitar um valor inteiro para `num`
	13	`soma ← soma + num`	atribui a `soma` o valor `soma + num`
		`enquanto(num ≠ 0)`	como `num` $\neq 0$, o programa continuará a repetição
4		`leia(num)`	
	17	`soma ← soma + num`	
		`enquanto(num ≠ 0)`	
0		`leia(num)`	
	17	`soma ← soma + num`	
		`enquanto(num ≠ 0)`	como `num` $= 0$, o programa pára a repetição e passa para o próximo comando após o laço (linha `7` da versão 4)
		`escreva(soma)`	imprime o conteúdo de soma: 17

Saída do programa: 17

Deve-se notar que após o programa "ler" o número `0`, ainda é executado o comando `soma ← soma + num` antes do fluxo (de execução) voltar ao início do comando `enquanto`. Mas isto não provoca qualquer erro, uma vez que o valor "lido" (`0`) não altera o valor em `soma`. No entanto, o que fazer se o problema consistisse em somar uma sequência de inteiros não negativos (positivos ou nulos), finalizada por um número negativo?

(Antes de seguir com a leitura, pense nesta questão.)

Uma primeira tentativa em responder à questão anterior é tentar aproveitar a versão 4 anterior, apenas trocando a condição de parada de `num ≠ 0` para `num ≥ 0`.

Porém isto não funcionaria, pois: ao digitar um finalizador (um número negativo), este será adicionado ao conteúdo de `soma` (na linha `6`), "estragando" o resultado final!

Se a solução do novo problema não pode ser "tão semelhante" à versão 4, também não será radicalmente diferente. Basta efetuarmos uma mera inversão entre as linhas `5` e `6` da referida versão (além da troca na condição de parada, acima citada),

```
inteiro num, soma
leia(num)
soma ← 0
enquanto (num ≥ 0) {
  soma ← soma + num
  leia(num) }
escreva(soma)
```

Versão 5: Soma até usuário digitar um número negativo

Note também a diferença da "inicialização" da variável `soma`. Começando com zero ela ficará com o valor desjado após a primeira execução do comando `soma ← soma + num`, ou seja, ela assumirá o valor do primeiro elemento da sequência (se este for não negativo). Note também que, para uma sequência vazia, ou seja, para a entrada que já começa com um número negativo, o resultado escrito na saída será `0`, que é o desejado.

Exercício 2 *Faça a simulação deste programa para as entradas* `10 3 4 -5` *e veja as diferenças em relação à* *versão* `4`*, prestando atenção às linhas* `5` *e* `6`*. Note que após a leitura do número* `-5`*, não é executado o comando* `soma ← soma + num`*, pois o teste do* `enquanto` *fica falso e o programa passa para o comando* `escreva(soma)`*.*

Exercício 3 *Ainda considerando o enunciado do problema 2, verifique o que há de errado com o programa a seguir:*

```
inteiro num, soma
leia(num)
soma ← 0
enquanto(num ≠ 0) {
  leia(num)
  soma ← soma + num }
escreva(soma)
```

Versão 6: Aqui existe um erro!

3.2 Como computar uma pontência de inteiros

Problema 3 *Dado* `x` *real e* `n` *natural, faça um programa que calcule* $\mathtt{x}^n$*.*

Novamente, se `n` fosse fixo e conhecido, por exemplo `n=3`, bastaria fazer

```
leia(x)
escreva(x * x * x)
```

e portanto a dificuldade deste problema se encontra na repetição arbitrária: repetir a operação de multiplicação um número variável de vezes, dependente da vontade do usuário.

Novamente, vamos iniciar no processo de resolução examinando um exemplo, `x = 2` e `n = 7`.

1. Multiplique `x` por `x`, guardando o resultado desta conta (`4`);
2. Multiplique por `x` o resultado guardado, guardando o novo resultado (`8`);
3. Repita o passo (2) outras 4 vezes (pois `n=7` e `x` já entrou 3 vezes como fator da multiplicação) obtendo o resultado final, `128`.

Quais variáveis são necessárias?

Claramente precisamos de variáveis para armazenar `x` e `n`. Além destas, será necessário outra variável (`res`) para armazenar o resultado das operações de potenciação de `x`, pois não é possível guardar o resultado destas potências na própria variável `x`: neste caso perderíamos seu valor original e não poderíamos continuar a obter novas potências deste (reflita sobre isto!).

Neste problema, notamos uma novidade em relação ao anterior, precisamos contar quantas vezes já usamos o fator `x` (original). Assim, vamos denotar esta variável pelo sugestivo nome de `cont`: a cada multiplicação somaremos `1` nessa variável. Assim, deveremos declarar as seguintes variáveis

```
inteiro x, n, res, cont
```

Quais comandos devem aparecer dentro do bloco de repetição?

Podemos observar no esquema inicial, elaborado acima, que o passo (2) é a parte repetitiva do processo: multiplicar o resultado atual de `res` por `x`, guardando este produto novamente em `res`. Feito isto, devemos então somar `1` ao contador `cont`, para indicar que mais uma multiplicação foi efetuada.

```
enquanto(...)  {
   res ← res*x
   cont ← cont+1 }
```

Qual a condição para que continue a repetição?

Já foi observado que a variável `cont` serve para contar quantas vezes o fator `x` já entrou no produto, assim devemos interromper o laço quando `cont = n`, ou de outro modo, devemos continuar este enquanto `cont < n`.

```
enquanto(cont < n) {
  res  recebe res*x
  cont ← cont+1 }
```

O que deve vir antes e depois da repetição?

Em primeiro lugar, devemos ler os valores de `x` e `n`. Precisamos, também atribuir valores iniciais adequados às variáveis `cont` e `res`. Como `cont` totaliza quanto fatores `x` já entraram no produto, o valor inicial de `cont` é zero. Assim, após a primeira execução do comando `res ← res*x`, gostaríamos de obter o valor de `x` em `res` e, por isto, devemos impor que `res` começe com o valor `1`.

Interrompido o laço, a resposta estará armazenada em `res`,

```
inteiro x, n, res, cont

leia(x, n)
cont ← 0
res ← 1
enquanto(cont < n) {
  res ← res*x
  cont ← cont+1 }
escreva(res)
```

Exercício 4 *Repare que o programa dá a resposta correta, também quando* `n = 0`, *mas é "proibido" ao usuário entrar com* `x=0`. *Como poderíamos eliminar esta restrição ?*

Exercício 5 *Sugerimos como exercício a simulação deste programa, para alguns dados de entrada à escolha do leitor.*

Exercício 6 *O seguinte programa foi proposto para calcular* x^n. *Analise o que há de errado nele.*

```
inteiro x, n, cont

leia(x, n)
cont ← 0
enquanto(cont < n) {
   x     ← x*x
   cont  ← cont+1 }
escreva(x)
```

Exercício 7 *É possível alterar o programa-solução do problema 3, dispensando a variável contadora* `cont` *? Isto é, tente deduzir uma outra versão para o referido problema, utilizando apenas três variáveis* `x`*,* `n` *e* `res`*.*

4 Algoritmo de Euclides para MDC

A seguir apresentaremos um resultado (deixando a demonstração de sua validade para o apêndice 7.2) sobre a função `mdc`: $\mathbb{N} \times \mathbb{N}^* \mapsto \mathbb{N}^*$, que determina o **máximo divisor comum (mdc)** entre dois inteiros positivos. Este resultado nos permitirá deduzir um algoritmo muito eficiente para a determinação do *mdc*, algoritmo este conhecido por Euclides (matemático grego) no longínquo século III a.C.

Proposição 1 *Dados dois inteiros não negativos a e b, com $b \neq 0$*

$$mdc(a, b) = mdc(b, a \bmod b),$$

onde `a` mod `b` *é o resto da divisão inteira de* `a` *por* `b`*.*

Se você gosta de formalismo ou não acredita que este resultado seja válido, veja a demonstração do mesmo na sub-seção 7.2, página 19.

Problema 4 *Desejamos desenvolver um programa que, para quaisquer dois valores não negativos a e b, com $b \neq 0$, calcule o máximo divisor comum entre ambos.*

Resolução Da proposição anterior sabemos que, para todo $\mathtt{a} \in \mathbb{N}$ e $\mathtt{b} \in \mathbb{N}^*$,

$$\mathtt{mdc(a,b) = mdc(b,a\ mod\ b).} \tag{1}$$

Observação 1 *O resto de uma divisão inteira é menor ou igual ao dividendo e estritamente menor que o divisor, isto é, se* `a = q b + r`*, com* $\mathtt{q} \in \mathbb{N}$ *e* $0 \leq \mathtt{r} < \mathtt{b}, \mathtt{r} \in \mathbb{N}$*, então* $\mathtt{r} \leq \mathtt{a}$*. Portanto, a expressão acima "reduz" o problema, no sentido que os valores do lado direito nunca são maiores que os do lado esquerdo, ou seja,*

$$\mathtt{r = a\ mod\ b} \leq \mathtt{min}\{\mathtt{a,b - 1}\}, \qquad (se\ \mathtt{q = a\ div\ b,\ \ a = qb + r}).$$

Estamos interessados em calcular o `mdc(a,b)`, portanto podemos perguntar

"em que situação a expressão (1) resolve trivialmente o problema"?

Vamos aplicar a expressão (1) a alguns exemplos para sentirmos o seu funcionamento.

1. Computar o `mdc` entre `21` e `14`:

$$\texttt{mdc(21,14)} \overset{(1)}{=} \texttt{mdc(14,7)} \overset{(1)}{=} \texttt{mdc(7,0)}$$

   e notamos que (1) não é mais aplicável à `mdc(7,0)`, pois neste caso `b = 0`, mas sua aplicação também não é mais necessária visto que

$$\texttt{mdc(7,0) = 7.}$$

2. Computar o `mdc` entre `12` e `5`:

$$\texttt{mdc(12,5) = mdc(5,2) = mdc(2,1) = mdc(1,0)}$$

   e analogamente ao exemplo acima, paramos numa situação com `mdc(x,0) = x`, para todo `x > 0`, ou seja,

$$\texttt{mdc(12,5) = mdc(1,0) = 1.}$$

(Matematicamente, isto significa que `12` e `5` são primos entre si.)

Simplificadamente podemos montar as seguintes tabelas para os dois exemplos acima

a	b	a mod b
21	14	7
14	7	0
$\boxed{7}$	0	

a	b	a mod b
12	5	2
5	2	1
2	1	0
$\boxed{1}$	0	

Observação 2 *Aqui é necessário ressaltar que para todo inteiro* `x > 0`*, é fácil ver que* `mdc(x,0) = x`

Quais comandos devem aparecer dentro de um bloco de repetição? e
Qual a condição para que se continue a repetição?

A observação 1 permite-nos identificar a parte repetitiva do processo (resposta à 2^a pergunta) e sua condição de parada (resposta à 3^a pergunta).

```
enquanto (não se atingiu expressão mdc(x,0)) {
   para calcular mdc(a,b)
      aplique a redução, fazendo
      com que o novo a passe a ser o b
      e o novo divisor passe a ser a mod b
   }
```

Versão 1

Talvez seja o momento de pensarmos um pouco na resposta à 1^a pergunta,

Quais Variáveis são necessárias?

Podemos notar a necessidade de variáveis inteiras para armazenar os sucessivos valores de dividendos (`a`) e divisores (`b`).

Na liguagem estruturada temos

```
inteiro a, b
```

Com isto, podemos (re)escrever uma primeira versão de algoritmo,

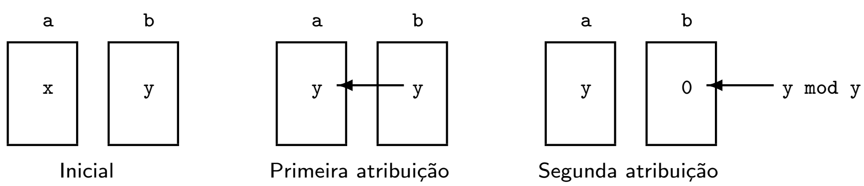
```
enquanto (b ≠ 0) {
   a ← b
   b ← a mod b
   }
```

Versão 2: Primeira tentativa (errônea) de algoritmo para MDC

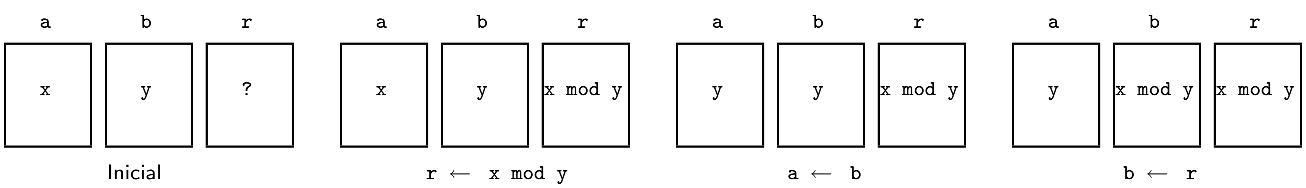
Para verificarmos quão boa está a versão acima, vamos simulá-la para `(24,14)`

a	b	
21	14	
14		[a ← b ≡ 14]
	0	[b ← a mod b ≡ 14 mod 14 ≡ 0]

e notamos um erro no algoritmo. O problema foi tentar trocar diretamente o conteúdo de duas variáveis sendo que uma dependia da outra. Estas situações estão ilustradas na figura a seguir,



Então poderíamos lançar mão de uma "área de salvamento" `r`, para armazenarmos o valor `21 mod 14` para a atribuição posterior (`b ← 21 mod 14`).



com isto o problema anterior é eliminado.

Observação 3 *Vale ressaltar que este é um "truque" muito comum em programação, pois é frequente precisarmos trocar os valores entre duas variáveis. O melhor exemplo de algoritmo onde isto ocorre (o tempo todo), são os algoritmos de ordenação: deseja-se ordenar, crescente, os valores distribuidos em uma sequência de variáveis* $\mathtt{x}_i$, $i \in \{1, 2, \ldots, n\}$, *de tal forma que ao final (tenhamos preservado todos os valores iniciais e)* $\mathtt{x}_i \leq \mathtt{x}_{i+1}$, *para todo* $i \in \{1, 2, \ldots, n-1\}$.

Falta apenas respondermos a 4^a pergunta,

Quais os comandos necessários antes e depois do bloco de repetição?

Antes de entrar no bloco de repetição as variáveis `a` e `b` devem conter os valores dos dois números dos quais se deseja calcular o `mdc`. Para isto, basta fazermos

```
leia(a)
leia(b)
```

por enquanto suporemos `a` $\geq$ `b`.

No final da repetição, quando `b = 0`, teremos que `mdc(a,b) = a`. Portanto, basta acrescentarmos, após o bloco de repetição, a linha

```
escreva(a)
```

Assim, a nova versão que podemos construir é a seguinte (já acrescentando a variável de salvamento `r`,

```
enquanto (b ≠ 0) {
   r ← a mod b
   a ← b
   b ← r
   }
```

Versão 3: Correção sobre a versão 2

Apesar desta parecer ser a versão definitiva, é aconselhável realizarmos algumas simulações para "sentirmos" se é ou não correta,

a	b	r
21	14	?
		7
14		
	7	
		0
$\boxed{7}$	0	

a	b	r
14	21	?
		14
21		
	14	
		7
14		
	7	
		0
$\boxed{7}$	0	

a	b	r
12	5	?
		2
5		
	2	
		1
2		
	1	
		0
$\boxed{1}$	0	

As simulações parecem indicar que

H.I *o algoritmo ordena* `a` *e* `b`*, isto é, se* `a < b`*, seus valores são trocados.*

H.II *o algoritmo funciona para todo* `a`$\geq$`0`*,* `b > 0`

Observação 4 *Porém não podemos ter certeza da validade de H.I e H.II, visto que estas conclusões são baseadas apenas em simulações, se quisermos ter a garantia da "corretude" do algoritmo deveríamos formalizar isto para todos os* `a` $\geq$ `0` *e* `b` $>$ `0`*, como na versão simplificada abaixo.*

1. Se `a=A<b=B`, na primeira iteração do laço teremos

a	b	r
A	B	
		A
B		
	A	

A<B $\Rightarrow$ A mod B $\equiv$ A

portanto vale H.I.

2. Usando H.I, sem perda de generalidade, vamos considerar um passo qualquer do algoritmo, sem perda de generalidade em função do raciocínio acima, com

$$\texttt{A} \geq \texttt{B} > \texttt{0}$$

Vamos observar esquematicamente as variáveis `a` e `b` em um passo qualquer do laço, admitindo que `a = A` e `b=B`$\neq$`0`,

a	b	r
A	B	
		A mod B
B		
	A mod B	

[da observação 1, página 10, `(x mod y) < y`]

e sabemos que `mdc(x,y) = mdc(y,x mod y)`, portando, `mdc(A,B)= ... = mdc(x,y)= ... = mdc(z,0)`.

Ou seja, em um passo (do laço), o valor armazenado em `b` é reduzido, ao menos, de uma unidade, logo em um número finito de passos `b` atinge `0`, e neste momento a variável `a` conterá o máximo divisor comum entre `A` e `B`, visto que, do raciocínio acima `mad(A,B)= ... = mdc(z,0)`.

5 Funções

Nesta seção mudaremos um pouco o enfoque adotado até aqui: construir algoritmo a partir de quatro perguntas básicas. Nosso objetivo nesta seção é apresentar o conceito de *subrotinas*, particularmente, as *funções* computacionais. Devido às grandes semelhanças entre estas e as funções matemática, sugerimos fortemente ao leitor que procure, ao longo desta seção, fazer analogias entre este dois conceitos de funções.

A utilidade mais elementar de subrotinas é a "economia de digitação": na elaboração de programas, é comum necessitarmos escrever várias vezes uma mesma seqüência de comandos. Esta redundância é ilustrada no exemplo a seguir,

Problema 5 *Faça um programa que leia dois números reais x, y e dois inteiros positivos a, b, calculando em seguida o valor da expressão $x^a + y^b + (x-y)^{a+b}$.*

Resolução Lembrando do problema 3 (página 3), podemos notar que, a solução do presente problema é repetir três vezes a seqüência de comandos que calcula a potenciação de dois números: uma vez para o cálculo de x^a, outra para y^b e uma última para $(x-y)^{a+b}$. Daí, podemos obter um primeira solução

```
inteiro a, b, cont
real    x, y
        pot, /* para cálculo de potenciacao.  */
        soma /* soma das potencias ja calculadas.  */

leia(x, y, a, b)

/* Sequencia 1:  cálculo de x^a */
cont ← 0,  pot ← 1
enquanto (cont < a){
  pot ← pot * x
  cont ← cont + 1}

/* x^a esta guardado em pot */
soma ← pot

/* Sequencia 2:  cálculo de y^b */
cont ← 0,  pot ← 1
enquanto (cont < b){
  pot ← pot * y
  cont ← cont + 1}

/* y^b esta guardado em pot */
soma ← soma + pot

/* Sequencia 3:  cálculo de (x − y)^(a+b) */
cont ← 0,  pot ← 1
enquanto (cont < a + b){
  pot ← pot * (x - y)
  cont ← cont + 1}
/* (x − y)^(a+b) esta guardado em pot */
soma ← soma + pot

escreva(soma)
```

Versão 1: Versão ingênua para o cálculo de x^a, y^b e $(x-y)^{a+b}$

As seqüências de comandos para o cálculo de x^a, y^b e $(x-y)^{a+b}$ são muito semelhantes, diferindo apenas no controle de parada do laço (que depende do expoente) e na atualização da variável `pot` (que depende da base).

Deste modo, podemos uniformizar estas três seqüências, lançando mão de duas novas variáveis, que sugestivamente, denominaremos `base` e `exp`, respectivamente, para armazenar os valores dos expoentes e das bases. Assim, na sequência 1 faríamos <`base ← x, exp ← a`>, na sequência 2 <`base ← y, exp ← b`> e na sequência 3 faríamos <`base ← x-y e exp ← a+b`>.

Este pequeno truque, produz a seguinte versão (note os retângulos),

```
inteiro a, b, cont, [exp]
real    x, y
        pot, soma, [base]

leia(x, y, a, b)

[base ← x,  exp ← a]
cont ← 0,  pot ← 1
/* Sequência 1:  cálculo de x^a */
enquanto (cont < exp) {
  pot ← pot * base
  cont ← cont + 1 }
/* Fim:  sequência 1 */
/* x^a esta guardado em pot */
soma ← pot

[base ← y,  exp ← b]
cont ← 0,  pot ← 1
/* Sequência 2:  cálculo de y^b */
enquanto (cont < exp) {
  pot ← pot * base
  cont ← cont + 1 }
/* Fim:  sequência 2 */
/* y^b esta guardado em pot */
soma ← soma + pot

[base ← x - y,  exp ← a + b]
cont ← 0,  pot ← 1
/* Sequência 3:  cálculo de (x − y)^(a+b) */
enquanto (cont < exp) {
  pot ← pot * base
  cont ← cont + 1 }
/* Fim:  sequência 3 */
/* (x − y)^(a+b) esta guardado em pot */
soma ← soma + pot

escreva(soma)
```

Versão 2: Versão uniformizada para cômputo das exponenciais

Neste segunda versão, podemos observar mais claramente os códigos redundantes: as seqüências 1, 2 e 3 são idênticas. O inconveniente de se escrever várias vezes um mesmo trecho de programa fica mais evidente ao imaginarmos um código muito extenso.

Uma **subrotina** é um trecho de programa que recebe um **nome**, ao qual se fará referência "cada vez que for necessário uma nova cópia deste trecho". Nestas referências, diremos que a subrotina está sendo **chamada** ou **invocada**. No exemplo acima, podemos notar claramente os trechos candidatos a subrotinas: as linhas entre `/* Sequência` $\boxed{?}$`:` $\boxed{\cdots}$ `*/` e `/* Fim: sequência` $\boxed{?}$ `*/`.

Cada subrotina, além do nome, também poderá ter uma **lista de parâmetros**, ou seja, uma lista de variáveis que receberão os valores necessários para que a subrotina possa efetuar os cálculos desejados. A utilidade destes parâmetros pode ser notada no exemplo anterior: a lista de parâmetros da rotina pra calcular a potência entre dois números, deverá conter uma variável para receber o valor da base e outra para o valor do expoente.

Podemos notar no exemplo considerado, que os trechos do programa que calculam x^a, y^b e $(x-y)^{a+b}$ sempre deixam o resultado final na variável `pot`. Uma subrotina que devolve um valor, a ser utilizado no local de sua chamada, será denominada **função** (compare este conceito com o de função matemática).

Por razões práticas, vamos exigir que ao declarar uma função, deve-se declarar também os tipos dos parâmetros (domínio no caso da Matemática) e o tipo de valor retornado (contra-domínio). Estas declarações tem um efeito prático do ponto de vista da computação, fica "mais fácil" o compilador verificar se houve erro na chamada da subrotina e na passagem de valores para os seus parâmetros.

5.1 Parâmetros formais e efetivos

Como já foi observado antes, existem duas situações distintas no uso de sub-rotinas: sua declaração e sua chamada. Estas situações dão origem a duas categorias de parâmetros, os formais e os efetivos, respectivamente.

Formalmente, uma sub-rotina em PORTUGOL é assim declarada

```
[tipo_de_valor_a_ser_devolvido]  nome_da_subrotina (
                                 tipo_do_parâmetro_1 nome_do_parâmetro_1, ...,
                                 tipo_do_último_parâmetro nome_do_último_parâmetro )
```

Os símbolos '[' e ']' foram utilizados para indicar que esta declaração é opcional. Se ela não aparece, não vamos reconhecer a subrotina com função (pois, em matemática, função deve devolver valores, sempre)[3].

No exemplo anterior, podemos especificar a subrotina para calcular a potenciação do seguinte modo:

```
real elevado (real base, inteiro exp)
```

Essa especificação pode ser chamada de **cabecalho** ou **protótipo** da subrotina. Já os comandos que compõem a subrotina propriamente dita, são chamados de **corpo** da subrotina. No corpo podem aparecer, declarações de variáveis necessárias apenas dentro da subrotina e por isto são denominadas **variáveis locais**. Por fim, para especificar o valor devolvido pela subrotina, no caso de funções, utilizaremos um novo comando:

```
devolva(expressão)
```

que calcula a expressão e retorna, para o "local" onde foi chamada a subrotina, este valor (como em funções matemáticas: `aux` $\leftarrow$ `f(x,y)`).

Com isto podemos escrever uma função para o exemplo anterior, que denotaremos por `elevado`

[3] Já na linguagem `C`, quando não aparece o "contra-domínio", este é automaticamente assumido como sendo do tipo inteiro e portanto, em `C` todas as subrotinas são funções.

```
real elevado(real base, inteiro exp) {
    inteiro cont  /* variaveis */
    real pot      /* locais */
    cont ← 0      /* início do corpo da função */
    pot ← 1
    enquanto (cont < exp) {
      pot ← pot * base
      cont ← cont + 1 }
    devolva(pot)
    }             /* fim da função */
```

Resta especificar agora como um programa poderá chamar uma subrotina, em particular, uma função. Para isto, basta escrever o nome da subrotina seguido da lista dos valores que serão passados para os parâmetros. Esta chamada pode ser feita em qualquer trecho do programa e, no caso de retorno de um valor (função), deve-se utilizar uma variável, de mesmo tipo que o valor a ser retornado pela função.

Considerando a subrotina `elevado`, as seguintes chamadas são possíveis, onde `v, x` e `soma` são variáveis do tipo `real` e `a` é do tipo `inteiro`:

1. `v ← elevado(x, 10)`
2. `soma ← soma + elevado(x,a)`
3. `se (elevado(3,5) > x)`
4. `escreva(elevado(2,a))`
5. `v ← elevado(elevado(x,a),3)`

Desde que os valores de `x` e `a` tenham sido previamente definidos, isto é, antes de aparecer qualquer dos comandos acima, é necessário que já tenha ocorrido uma atribuição do tipo `x← EXPR_REAL` (itens 1, 2, 3 e 5) ou `a← EXPR_INTEIRA` (itens 2, 4 e 5).

Note que as seguintes chamadas de `elevado` contém erros,

(a)	(b)
`v ← 5.1` `elevado(v)`	`v ← 5.1` `elevado(2.1,v)`

Em *(a)* falta um dos dois parâmetros da função e em *(b)*, o segundo parâmetro passado é um `real` quando deveria ser um `inteiro`.

Agora já podemos "enxugar" o algoritmo para o problema 5,

```
/* Funcao de exponenciacao */
real elevado(real base, inteiro exp) {
    inteiro cont /*variaveis */
    real pot /*locais */

    cont ← 0
    pot ← 1
    enquanto (cont < exp) {
      pot ← pot * base
      cont ← cont + 1 }

    devolva(pot)
    }
```

```
/* Programa principal */
inteiro a, b
real x, y, soma {
    leia(x, y, a, b)
    soma ← elevado(x, a)
    soma ← soma + elevado(y, b)
    soma ← soma + elevado(x - y, a + b)
    escreva(soma)
    }
```

Versão 3: Forma enxuta para o cálculo de $x^a + y^b + (x - y)^{a+b}$

Em certo sentido, podemos comparar uma subrotina a uma "caixa preta": para saber usá-la não é necessário conhecer o seu interior (corpo), mas apenas o seu cabeçalho e qual o efeito provocado pela mesma (no problema anterior era retornar a potenciação de seus dois parâmetros); e para saber projetar o seu corpo não é preciso conhecer o programa que a utilizará, mas também apenas o seu cabeçalho (e objetivo).

6 Ponteiros

Em geral, quando alguém se referir, em qualquer linguagem de programação à uma variável, invocando seu nome, estará se referindo ao conteúdo desta variável.

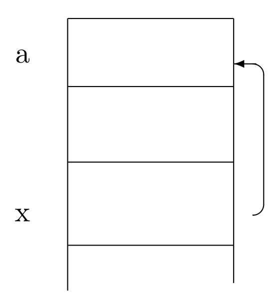
Em `C` é possível obter o endereço de uma variável `x` através da diretiva

`&x`

Pode-se definir variáveis que não armazenam diretamente qualquer dado, mas "apontam" para uma outra variável, ou seja, armzenam o endereço de uma outra variável. A estas variáveis damos o nome de **apontadores**: variáveis que guardam endereços de outras variáveis (quaisquer). Se `x` é um apontador, a variável apontada por `x` é obtida usando o caracter *.

```
/* x apontador para inteiro.  */
x = &a;
printf("%d = %d", a, *x);
```

A variável `x` guarda o endereço de `a`



Representação esquemática: `x` aponta para conteúdo de `a`

Declaração: `Tipo *nome`

Exemplo:

```
void main(void) {
int a, b, *c, **y;
    scanf("%d%d", &a, &b);
    x = &a;
    y = &x;
    *x = 5;
    x = &b;
    **y = 10;
    printf("%d=%d=%d %d", b, *x, **y, a);
}
```

Execução: Saída na tela.

```
10=10=10 5
```

Alteração de valores de Parâmetros Efetivos

Este efeito é obtido através do uso de ponteiros, visto que os parâmetros formais, quando da execução da função, são considerados variáveis locais à mesma.

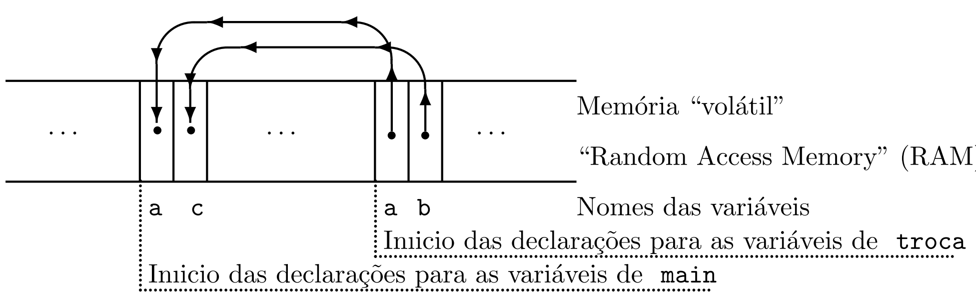
Deste modo a única possibilidade de alteração de parâmetros passados na chamada da função (**parâmetros efetivos**) é passar os endereços das variáveis que necessitam ter seus valores alterados na função. Conseqüentemente os parâmetros efetivos devem ser apontadores.

Exemplo: Uma função para troca de valores de duas variáveis tipo inteiro quaisquer.

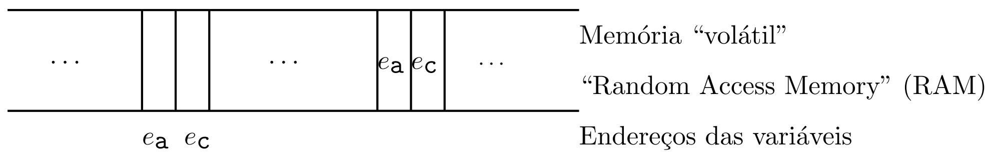
```
void troca(int *a, int *b) {

int aux;
    /* a e b devem receber os endereços das variá-
       veis as quais deseja-se trocar os conteúdos.  */
    aux = *a; /* Note que nem a nem b tem seus con- */
    *a = *b;  /* teúdos alterados, quem os tem são: */
    *b = aux; /*   o aux e os (futuros) parâmetros  */
    }         /*   efetivos.                        */
```

```
void main(void) {
inta, b, c, ...
   ⋮
   a = 5; c = 10;
   /* passamos os endereços das variáveis */
   troca(&a,&c);
   ⋮
   }
```



(a) Variáveis `a` e `b` (de `troca`) apontam, respectivamente, para `a` e `c` (de `main`)



(b) Apontadores vistos como endereço: e_a endereço da variável `a` e e_c da variável `c`

7 Apêndices

7.1 Resolução do Problema 1 (página 1)

Atribuições $\left\{ \begin{array}{l} \texttt{i} \leftarrow \texttt{0} \\ \texttt{vai\_um} \leftarrow \texttt{0} \end{array} \right.$

`repita`

Bloco de repetições $\left\{ \begin{array}{l} \texttt{i} \leftarrow \texttt{i+1} \\ \texttt{z}_i \leftarrow \texttt{(x}_i\texttt{+y}_i\texttt{+vai\_um) mod 10} \\ \texttt{vai\_um} \leftarrow \texttt{(x}_i\texttt{+y}_i\texttt{+vai\_um) div 10} \end{array} \right.$

`até que` $\texttt{x}_i$`=0 e` $\texttt{y}_i$`=0 /* estamos admitindo (infinitos) 0 á esquerda */`

`resultado em z`

Estamos usando os operadores binários `mod` e `div`, respectivamente, como o resto da divisão inteira e seu quociente ($\langle$ `a mod b = r` $\Leftrightarrow$ `b q + r = a` $\rangle$ e $\langle$ `a div b = q` $\rangle$). Mas se a linguagem de programação a ser utilizada não dispusesse deste tipo de operadores[4], poderíamos substituí-los facilmente por comandos de seleção (tipo `se`), uma vez que $\texttt{x}_i\texttt{+y}_i\texttt{+vai\_um}$ resulta um número pequeno, entre `0` e `19`: $\texttt{x}_i$ e $\texttt{y}_i$ são dígitos (`0,1,...,9`) e `vai_um` é `0` ou `1`.

7.2 Demonstração da Proposição 1

Apresentaremos abaixo uma demonstração para a proposisão 1, página 10.

Demonstração Seja $\bar{\texttt{q}} = \texttt{mdc(a,b)}$, então $\frac{\texttt{a}}{\bar{\texttt{q}}}$ e $\frac{\texttt{b}}{\bar{\texttt{q}}}$ são números inteiros positivos e para todo $\texttt{k} \geq \texttt{1}$, $\frac{\texttt{a}}{\bar{\texttt{q}}+\texttt{k}}$ e $\frac{\texttt{b}}{\bar{\texttt{q}}+\texttt{k}}$ não podem ambos ser inteiros, pois por construção, $\bar{\texttt{q}}$ é o <u>máximo</u> divisor comum à `a` e `b`.

Demonstraremos através dos dois ítens seguintes que

$$\bar{\texttt{q}} = \texttt{mdc(b, a mod b)}.$$

1. $\bar{\texttt{q}}$ é dividor de `b` e de `a mod b`.

   Por construção $\bar{\texttt{q}}$ é divisor de `b` e colocando

   $$\texttt{r = a mod b}, \text{ i. e., } \texttt{a = q b + r}, 0 \leq \texttt{r<b}, \texttt{q} \in \mathbb{N}. \tag{2}$$

   $\bar{\texttt{q}}$ também divide `r`, pois

   $$\frac{\texttt{r}}{\bar{\texttt{q}}} = \frac{\texttt{a}}{\bar{\texttt{q}}} - \texttt{q}\frac{\texttt{b}}{\bar{\texttt{q}}} \quad \text{(e } \bar{\texttt{q}} \text{ divide } \texttt{a} \text{ e } \texttt{b}\text{)},$$

[4] Tanto o Pascal quanto o C dispõe dos operadores `div` e `mod`, em Pascal existem e são precisamente estes e em C são, respectivamente, `/` e `%`.

logo $\frac{\mathtt{a}}{\bar{\mathtt{q}}} - \mathtt{q}\frac{\mathtt{b}}{\bar{\mathtt{q}}} \in \mathbb{N}$ ($\mathtt{0} \leq \mathtt{a\text{-}qb<b}$).

2. $\bar{\mathtt{q}}$ é o maior divisor comum de `b` e `r = a mod b`.

   Suponhamos por absurdo que exista um $\mathtt{k} \geq \mathtt{1}$, para o qual $\bar{\mathtt{q}} + \mathtt{k}$ seja divisor de `b` e de `r`, ou seja, $\frac{\mathtt{b}}{\bar{\mathtt{q}}+\mathtt{k}}$ e $\frac{\mathtt{r}}{\bar{\mathtt{q}}+\mathtt{k}}$ são inteiros e como da equação (2)

$$\frac{\mathtt{r}}{\bar{\mathtt{q}}+\mathtt{k}} = \frac{\mathtt{a}}{\bar{\mathtt{q}}+\mathtt{k}} - \mathtt{q}\frac{\mathtt{b}}{\bar{\mathtt{q}}+\mathtt{k}}$$

   segue que $\frac{\mathtt{a}}{\bar{\mathtt{q}}+\mathtt{k}}$ também é inteiro (não é possível um número <u>não inteiro</u> somado a um <u>inteiro</u> resultar <u>inteiro</u>), neste caso $\bar{\mathtt{q}}+\mathtt{k}$ seria um divisor comum entre `a` e `b`, encerrando uma contradição com a hipótese de $\bar{\mathtt{q}}$ ser o maior dos divisores comuns destes números. Logo, $\bar{\mathtt{q}}$=`mdc(b, a mod b)`.

   Portanto `mdc(a,b)` = $\bar{\mathtt{q}}$ = `mdc(b, a mod b)`. ■