

Notas de Aula de Algoritmos e Programação de Computadores

FLÁVIO KEIDI MIYAZAWA

com a colaboração de

TOMASZ KOWALTOWSKI

Instituto de Computação - UNICAMP

Versão 2000.1

Estas notas de aula não devem ser usadas como única fonte de estudo. O aluno deve ler outros livros disponíveis na literatura.

Nenhuma parte destas notas pode ser reproduzida, qualquer que seja a forma ou o meio, sem a permissão dos autores.

Os autores concedem a permissão explícita para a utilização e reprodução deste material no contexto do ensino de disciplinas regulares dos cursos de graduação sob a responsabilidade do Instituto de Computação da UNICAMP.

© Copyright 2000

Instituto de Computação
UNICAMP
Caixa Postal 6176
13083-970 Campinas-SP
{fkm,tomasz}@ic.unicamp.br

10 Recursividade

Dizemos que um objeto é dito ser *recursivo* se ele for definido em termos de si próprio.

Este tipo de definição é muito usado na matemática. Um exemplo disto é a função fatorial, que pode ser definido como:

$$n! = \begin{cases} 1 & \text{se } n = 0 \text{ e} \\ n \cdot (n - 1)! & \text{se } n > 0. \end{cases}$$

Existem muitos objetos que podem ser formulados de maneira recursiva. Este tipo de definição permite que possamos definir infinitos objetos de maneira simples e compacta. Podemos inclusive definir algoritmos que são recursivos, i.e., que são definidos em termos do próprio algoritmo. Nesta seção veremos como poderemos usar esta poderosa técnica como estratégia para o desenvolvimento de algoritmos.

10.1 Projeto por Indução

Os algoritmos recursivos são principalmente usados quando a estratégia de se resolver um problema pode ser feita de maneira recursiva ou quando os próprios dados já são definidos de maneira recursiva. Naturalmente existem problemas que apresentam estas duas condições mas que não se é aconselhado usar algoritmos recursivos. Um problema que pode ser resolvido de maneira recursiva também pode ser resolvido de maneira interativa. Algumas das principais vantagens de usar recursão são a possibilidade de se gerar programas mais compactos, programas fáceis de se entender e o uso de uma estratégia para se resolver o problema. Esta estratégia é a de atacar um problema (projetando um algoritmo) sabendo (supondo) se resolver problemas menores: *Projeto por Indução*. Note que já usamos esta idéia para desenvolver o algoritmo de busca binária.

Os objetos de programação que iremos usar para trabalhar com a recursão serão as funções e os procedimentos. Assim, uma rotina (função ou procedimento) é dita ser recursiva se ela chama a si mesma. Uma rotina recursiva pode ser de dois tipos: se uma rotina \mathcal{R} faz uma chamada de si mesmo no meio de sua descrição então \mathcal{R} é uma rotina recursiva direta; caso \mathcal{R} não faça uma chamada a ela mesma, mas a outras rotinas que porventura podem levar a chamar a rotina \mathcal{R} novamente, então \mathcal{R} é uma rotina recursiva indireta.

Quando uma rotina recursiva está sendo executada, podemos visualizar uma seqüência de execuções, uma chamando a outra. Veja a figura 26.

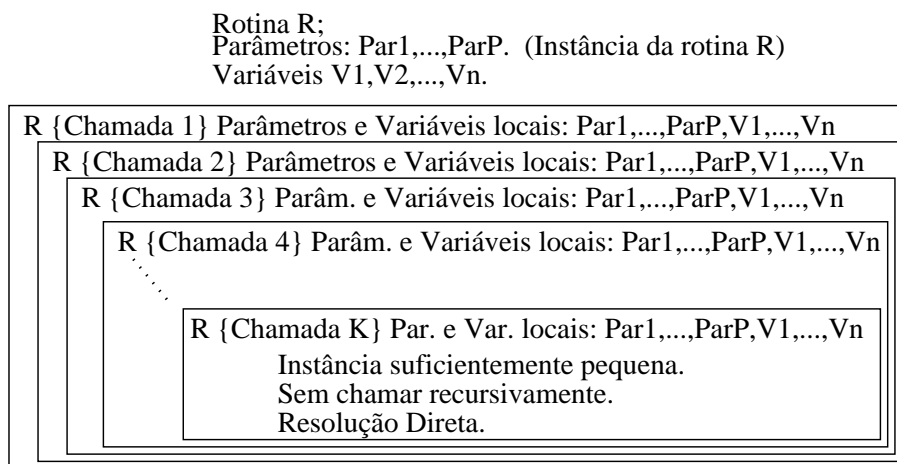
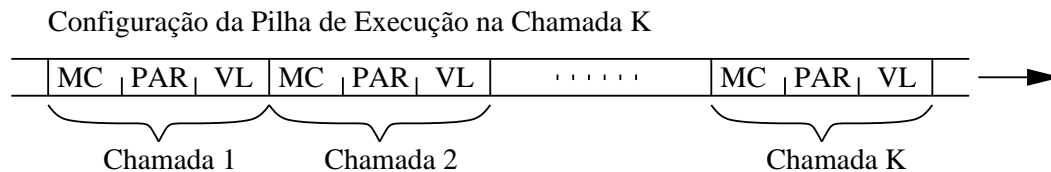


Figura 26: Seqüência das chamadas recursivas de uma rotina \mathcal{R} até sua última chamada recursiva.

Quando em uma determinada chamada recursiva a rotina faz referência a uma variável ou parâmetro, esta está condicionada ao escopo daquela variável. Note que em cada uma das chamadas recursivas, as variáveis e parâmetros (de mesmo nome) podem ter valores diferentes. Se a rotina faz referência à variável V_1 na terceira chamada recursiva, esta irá obter e atualizar o valor desta variável naquela chamada. A figura 27 apresenta a configuração da memória usada para a execução do programa, chamada pilha de execução, no momento em que foi feita a k -ésima chamada recursiva. A seta indica o sentido do crescimento da pilha de execução.



MC = Memória de Controle (para retorno da chamada)
 PAR = Memória relativa aos parâmetros da chamada
 VL = Memória relativa às variáveis locais da chamada

Figura 27: Configuração da pilha de execução após k -ésima chamada recursiva.

10.2 Garantindo número finito de chamadas recursivas

Note que na figura 26, da primeira chamada até a última, fizemos k chamadas recursivas. Em um programa recursivo é muito importante se garantir que este processo seja finito. Caso contrário, você terá um programa que fica fazendo “infinitas” chamadas recursivas (no caso até que não haja memória suficiente para a próxima chamada).

Para garantir que este processo seja finito, tenha em mente que sua rotina recursiva trabalha sobre uma instância e para cada chamada recursiva que ela irá fazer, garanta que a instância que será passada a ele seja sempre mais restrita que a da chamada superior. Além disso, garanta que esta seqüência de restrições nos leve a uma instância suficientemente simples e que permita fazer o cálculo deste caso de maneira direta, sem uso de recursão. É isto que garantirá que seu processo recursivo tem fim. Vamos chamar esta instância suficientemente simples (que a rotina recursiva resolve de maneira direta) de *base da recursão*.

Nas duas figuras a seguir apresentamos um exemplo de programa usando recursão direta e indireta.

```

program ProgramaFatorialRecursivo;

function fatorial(n : integer):integer;
begin
  if (n=0)
    then fatorial := 1
    else fatorial := n * fatorial(n-1);
end; { fatorial }

var n : integer;
begin
  write('Entre com um número: ');
  readln(n);
  writeln('O fatorial de ',n,
    ' é igual a ',fatorial(n));
end.

```

Figura 28: Fatorial recursivo (recursão direta).

```

program ProgramaParesImpares;
function impar(n : integer):boolean; forward;
function par(n : integer):boolean;
begin
  if n=0 then par:=true
  else if n=1 then par:=false
  else par := impar(n-1);
end; { par }
function impar(n : integer):boolean;
begin
  if n=0 then impar := false
  else if n=1 then impar:=true
  else impar := par(n-1);
end; { impar }
var n : integer;
begin
  write('Entre com um inteiro positivo: ');
  readln(n);
  if (par(n)) then writeln('Número ',n,' é par.')
  else writeln('Número ',n,' é ímpar. ');
end.

```

Figura 29: Funções par e impar (recursão indireta).

Estes dois exemplos são exemplos “fabricados” de rotinas recursivas, mas que tem o propósito de apresentar este tipo de rotina como uma primeira instância. Naturalmente existem soluções melhores para se implementar as funções acima. Discutiremos mais sobre isso posteriormente.

Agora vamos analisar melhor a função fatorial recursiva. Note que o parâmetro da função fatorial é um número

inteiro positivo n . A cada chamada recursiva, o parâmetro que é passado é diminuído de uma unidade (o valor que é passado na próxima chamada recursiva é $(n - 1)$). Assim, a instância do problema a ser resolvido (o cálculo de fatorial) vai diminuindo (ficando mais restrito) a cada chamada recursiva. E este processo deve ser finito porque existe uma condição que faz com que ele pare quando a instância ficar suficientemente pequena: caso $n = 0$.

Considere o programa da figura 29. Neste programa apresentamos um exemplo de recursão indireta. A função *par* não chama a si mesma, mas chama a função *impar* que em seguida pode chamar a função *par* novamente. A função *par* (*impar*) retorna true caso o parâmetro n seja par (ímpar).

Note que a primeira rotina que aparece no momento da compilação é a função *par*. Ela faz a chamada da função *impar*. Como esta função *impar* está definida abaixo da função *par*, a compilação do programa daria erro (por não ter sido previamente definida), caso não colocássemos uma informação dizendo a “cara” da função *impar*. Isto pode ser feito colocando antes da sua chamada uma diretiva dizendo como é a “cara” da função *impar*. Colocando apenas o cabeçalho da função ou procedimento, seguido da palavra **forward**;, estaremos dizendo ao compilador que tipo de parâmetros ela aceita e que tipo de resultado ela retorna. Com isso, poderemos fazer chamadas da rotina *impar*, mesmo que sua especificação seja feita bem depois destas chamadas.

Note também que o tamanho das instâncias das chamadas recursivas das funções *par* e *impar* também diminui de tamanho.

Projetando algoritmos por Indução

Projetar algoritmos por indução é muito parecido com o desenvolvimento de demonstrações matemáticas por indução.

Primeiro verifique se seu problema pode ser resolvido através de problemas menores do mesmo tipo que o original. Neste caso, ao projetar seu algoritmo, *assuma que você já tem rotinas para resolver os problemas menores*. Com isto em mente, construa seu algoritmo fazendo chamadas recursivas para resolver os problemas menores.

Lembrando que toda seqüência de chamadas recursivas deve parar em algum momento, construa uma condição para resolver os problemas suficientemente pequenos de maneira direta, sem fazer chamadas recursivas (base da recursão). Por fim, simule sua rotina para ver se todos os casos estão sendo cobertos.

10.3 Torres de Hanoi

Vamos ver um exemplo de programa recursivo, usando projeto indutivo, para se resolver um problema, chamado Torres de Hanoi.

Torre de Hanoi: Há um conjunto de 3 pinos: 1,2,3. Um deles tem n discos de tamanhos diferentes, sendo que os maiores estão embaixo dos menores. Os outros dois pinos estão vazios. O objetivo é mover todos os discos do primeiro pino para o terceiro pino, podendo se usar o segundo como pino auxiliar. As regras para resolver o problema são.

1. Somente um disco pode ser movido de cada vez.
2. Nenhum disco pode ser colocado sobre um disco menor que ele.
3. Observando-se a regra 2, qualquer disco pode ser movido para qualquer pino.

Escreva a ordem dos movimentos para se resolver o problema.

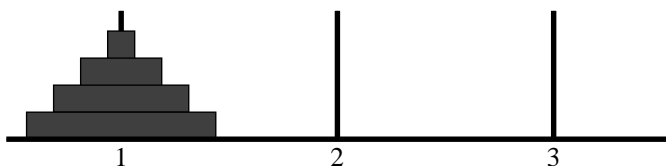


Figura 30: Torres de Hanoi com 4 discos.

Note que este problema consiste em passar os n discos de um pino de origem, para um pino destino, usando mais um pino auxiliar. Para resolver o problema das torres de hanoi com n discos, podemos usar a seguinte estratégia (veja figura 31):

1. Passar os $(n - 1)$ discos menores do primeiro pino para o segundo pino.

2. Passar o maior disco para o terceiro pino.
3. Passar os $(n - 1)$ discos do segundo pino para o terceiro pino.

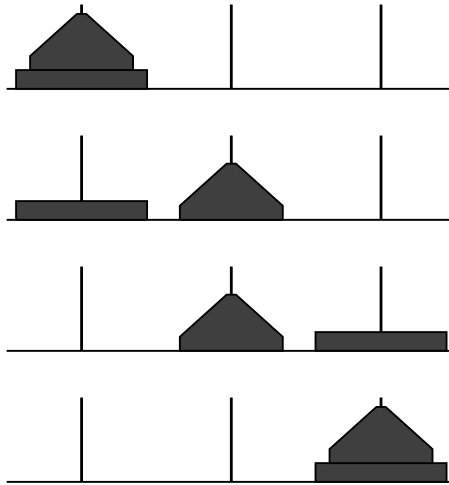


Figura 31: Estratégia do algoritmo das Torres de Hanoi.

Note que no item 1 acima, passamos $(n - 1)$ discos de um pino para o outro. Aqui é importante observar que todos estes $(n - 1)$ discos são menores que o maior disco que ficou no pino. Assim, podemos trabalhar com estes $(n - 1)$ discos como se não existisse o disco grande. Pois mesmo que algum destes $(n - 1)$ discos fique em cima do disco maior, isto não será nenhum problema, já que todos os disco em cima do maior disco são menores que este. Este mesmo raciocínio se propaga para se mover os $(n - 1)$ discos em etapas.

Observe que resolvemos o problema maior, com n discos, sabendo se resolver um problema de $n - 1$ discos. Além disso, para uma instância suficientemente pequena, com 0 discos, resolvemos o problema de maneira direta (base da recursão).

```

program ProgramaHanoi;
procedure Hanoi(n,origem,auxiliar,destino : integer);
begin
  if n>0 then begin
    hanoi(n-1,origem,destino,auxiliar);
    writeln('Mova de ',origem,' para ',destino);
    hanoi(n-1,auxiliar,origem,destino);
  end;
end;
var n : integer;
begin
  write('Entre com a quantidade de discos no primeiro pino: '); readln(n);
  writeln('A seqüência de movimentos para mover do pino 1 para o pino 3 é:');
  hanoi(n,1,2,3);
end.

```

10.4 Quando não usar recursão

Há momentos em que mesmo que o cálculo seja definido de forma recursiva, devemos evitar o uso de recursão. Veremos dois casos onde o uso de recursão deve ser evitado.

Chamada recursiva no início ou fim da rotina

Quando temos apenas uma chamada que ocorre logo no início ou no fim da rotina, podemos transformar a rotina em outra interativa (sem recursão) usando um loop. De fato, nestes casos a rotina recursiva simplesmente está simulando

uma rotina interativa que usa uma estrutura de repetição. Este é o caso da função fatorial que vimos anteriormente e que pode ser transformado em uma função não recursiva como no exemplo 7.9.

O motivo de preferirmos a versão não recursiva é porque cada chamada recursiva aloca memória para as variáveis locais e parâmetros (além da memória de controle) como ilustrado na figura 27. Assim, a função fatorial recursiva chega a gastar uma memória que é proporcional ao valor n dado como parâmetro da função. Por outro lado, a função fatorial interativa do exemplo 7.9 gasta uma quantidade pequena e constante de memória local. Assim, a versão interativa é mais rápida e usa menos memória para sua execução.

A seguir apresentamos um outro exemplo onde isto ocorre.

Exemplo 10.1 A função para fazer a busca binária usa uma estratégia tipicamente recursiva com uso de projeto por indução. A rotina deve encontrar um determinado elemento em um vetor ordenado. A idéia da estratégia é tomar um elemento do meio de um vetor ordenado e compará-lo com o elemento procurado. Caso seja o próprio elemento, a função retorna a posição deste elemento. Caso contrário, a função continua sua busca da mesma maneira em uma das metades do vetor. Certamente a busca na metade do vetor pode ser feita de maneira recursiva, como apresentamos na figura 32. Na figura 33 apresentamos a versão interativa.

Note que a base da recursão foi o vetor vazio que certamente é um vetor suficientemente pequeno para resolvermos o

```

const MAX      = 100;
type TipoVet = array[1..MAX] of real;

function BuscaBin(var v      : TipoVet;
                 inicio,fim : integer;
                 x          : real):integer;
var meio: integer;
begin
  if (inicio>fim) then {vetor vazio}
    BuscaBinaria := 0 {nao achou}
  else begin
    meio := (inicio+fim) div 2;
    if (x<v[meio]) then
      BuscaBin := BuscaBin(v,inicio,meio-1,x)
    else if (x>v[meio]) then
      BuscaBin := BuscaBin(v,meio+1,fim,x)
    else BuscaBin := meio;
  end;
end; { BuscaBin }

```

```

const MAX      = 100;
type TipoVet = array[1..MAX] of real;

function BuscaBin(var v      : TipoVet;
                 inicio,fim : integer;
                 x          : real):integer;
var pos,i,inicio,fim,meio: integer;
begin
  pos := 0;
  while (inicio<=fim) and (pos=0) do begin
    meio := (inicio+fim) div 2;
    if (x<v[meio]) then fim := meio-1
    else if (x>v[meio]) then inicio := meio+1
    else pos:=meio;
  end;
  BuscaBin := pos;
end; { BuscaBin }

```

Figura 33: Busca binária não recursivo.

Figura 32: Busca binária recursivo.

problema de maneira direta.

Repetição de processamento

Quando fazemos uso de recursão com várias chamadas recursivas ocorre na maioria das vezes que cada uma destas chamadas recursivas é independente uma da outra. Caso ocorram os mesmos cálculos entre duas chamadas recursivas independentes, estes cálculos serão repetidos, uma para cada chamada. Este tipo de comportamento pode provocar uma quantidade de cálculos repetidos muito grande e muitas vezes torna o programa inviável. Um exemplo claro disto ocorre na função Fibonacci, que apesar de ter sua definição de maneira recursiva, o uso de uma função recursiva causa um número exponencial de cálculos, enquanto a versão interativa pode ser feito em tempo proporcional a n (o parâmetro da função).

$$Fibonacci(n) = \begin{cases} 0 & \text{se } n = 0, \\ 1 & \text{se } n = 1 \text{ e} \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{se } n > 1. \end{cases}$$

A função Fibonacci recursiva, apresentada a seguir, é praticamente a tradução de sua definição para Pascal.

```

function fibo(n : integer):integer;
begin
  if n=0 then fibo:=0
  else if n=1 then fibo:=1
  else fibo:=fibo(n-1)+fibo(n-2);
end;

```

Na figura 34, apresentamos as diversas chamadas recursivas da função fibonacci, descrita acima, com $n = 5$. Por simplicidade, chamamos a função Fibonacci de F .

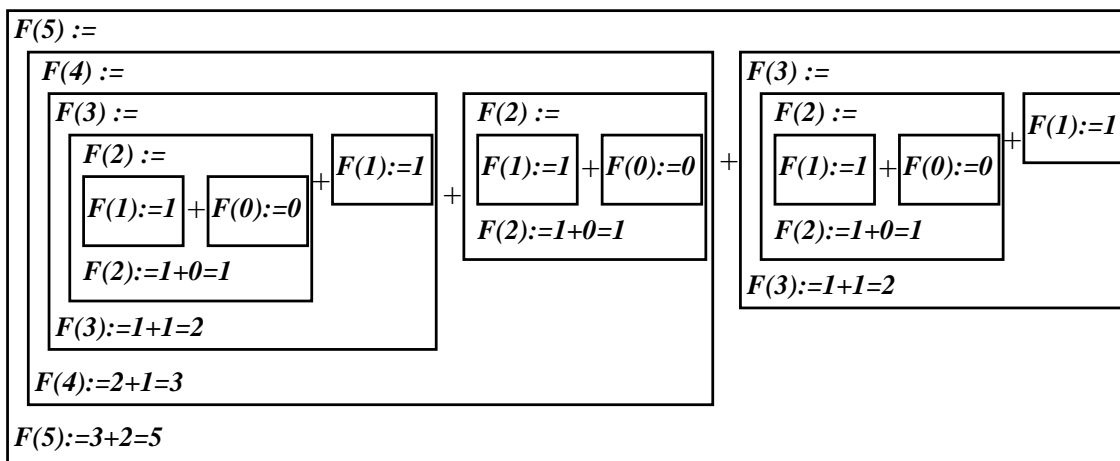


Figura 34: Chamadas recursivas feitas por Fibonacci(5).

Note que cada chamada recursiva é desenvolvida independente das chamadas recursivas anteriores. Isto provoca uma repetição dos cálculos para muitas das chamadas. Por exemplo: quando foi feita a chamada $F(3)$ (mais a direita na figura 34) para o cálculo de $F(5)$, poderíamos ter aproveitado o cálculo de $F(3)$ que tinha sido feito anteriormente para se calcular $F(4)$. Este tipo de duplicação de chamadas ocorre diversas vezes. Neste pequeno exemplo já é possível ver que a chamada de $F(1)$ ocorreu 5 vezes. De fato, uma análise mais detalhada mostraria que a quantidade de processamento feito por esta função é exponencial em n (com base um pouco menor que 2) enquanto o processamento feito pela correspondente função iterativa é linear em n .

Exercício 10.1 Seja n e k inteiros tal que $0 \leq k < n$. Usando a identidade

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

faça uma função recursiva que calcula $\binom{n}{k}$.

Qual a relação acima com o triângulo de Pascal? Faça uma função que calcula o valor de $\binom{n}{k}$ aproveitando a estratégia do cálculo do triângulo de Pascal.

Exercício 10.2 1. Uma planilha linear é formada por um vetor (com índices de 1 a $\text{MaxInd}=100$) onde cada elemento é chamado de célula.

2. Cada célula da planilha pode conter um valor real ou uma fórmula simples. Para diferenciar estas duas formas, cada célula apresenta um campo chamado Tipo que pode ser um dos seguintes caracteres: ('n','+', '-', '*', '/').
3. Caso a célula tenha Tipo igual a 'n', a célula contém um outro campo chamado valor que armazena um número real.
4. Se Tipo tem o caracter '+' (resp. '-', '*', '/'), então há dois campos (Ind1 e Ind2) que contém índices do vetor e indicam que o valor deste elemento é a soma (resp. subtração, multiplicação e divisão) dos valores associados às células dos índices Ind1 e Ind2.

Por exemplo, a fórmula simples

$$\boxed{+ \quad 4 \quad 5}$$

indica que o valor desta célula é obtido somando-se os valores das células 4 e 5.

5. Para uma ilustração desta planilha, veja a figura seguinte:

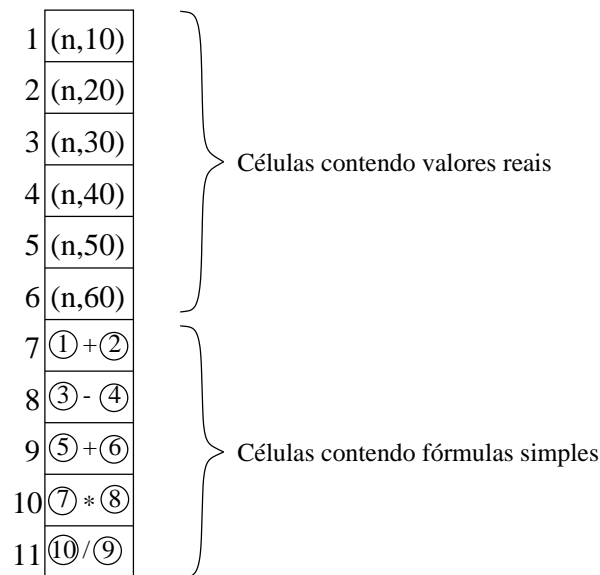


Figura 35: Planilha com valores reais e fórmulas simples.

6. Inicialmente todos as células da planilha contém um valor real igual a zero (i.e., Tipo='n').
7. Para obter o valor de uma célula o programa deve proceder da seguinte maneira:
Se a célula contém um valor real, então este é o próprio valor da célula. Caso contrário, obtemos os valores das células indicadas em Ind1 e Ind2 e aplicamos a operação correspondente.
8. Você pode considerar que uma célula é referenciada no máximo uma vez.

9. Para manipular os dados nesta planilha, você deverá fazer um programa para ler uma seqüência de linhas que pode ter um dos seguintes formatos:

Linha lida	Explicação do comando
<i>n</i> [Ind] [Val]	Coloca na célula [Ind] o valor [Val].
+ [Ind] [Ind1] [Ind2]	O valor da célula [Ind] é o valor da célula [Ind1] somado ao valor da célula [Ind2].
- [Ind] [Ind1] [Ind2]	O valor da célula [Ind] é o valor da célula [Ind1] subtraído do valor da célula [Ind2].
* [Ind] [Ind1] [Ind2]	O valor da célula [Ind] é o valor da célula [Ind1] multiplicado com o valor da célula [Ind2].
/ [Ind] [Ind1] [Ind2]	O valor da célula [Ind] é o valor da célula [Ind1] dividido pelo valor da célula [Ind2].
<i>p</i> [Ind]	Imprime na tela o valor da célula [Ind]. Este comando não altera a planilha.
.	Este comando (.) finaliza o programa.

10. Exemplo: Vamos considerar que queremos avaliar o valor da fórmula $((A + B) * (C - D)) / (E + F)$, para diferentes valores de *A, B, C, D, E, F*. Podemos considerar o valor de *A* na célula 1, o valor de *B* na célula 2, ..., o valor de *F* na célula 6. Com isso, podemos montar nossa expressão com os seguintes comandos:

Linhas de entrada	Comentários	
<i>n</i> 1 10	[1] ← 10	<i>A</i> ← 10
<i>n</i> 2 20	[2] ← 20	<i>B</i> ← 20
<i>n</i> 3 30	[3] ← 30	<i>C</i> ← 30
<i>n</i> 4 40	[4] ← 40	<i>D</i> ← 40
<i>n</i> 5 50	[5] ← 50	<i>E</i> ← 50
<i>n</i> 6 60	[6] ← 60	<i>F</i> ← 60
+ 7 1 2	[7] ← ([1] + [2])	[7] ← (<i>A</i> + <i>B</i>)
- 8 3 4	[8] ← ([3] - [4])	[8] ← (<i>C</i> - <i>F</i>)
+ 9 5 6	[9] ← ([5] + [6])	[9] ← (<i>E</i> + <i>F</i>)
* 10 7 8	[10] ← [7] * [8]	[10] ← (<i>A</i> + <i>B</i>) * (<i>C</i> - <i>F</i>)
/ 11 10 9	[11] ← [10]/[9]	[11] ← $((A + B) * (C - F)) / (E + F)$
	Neste ponto a planilha apresenta a configuração da figura 35	
<i>p</i> 11	Imprime o cálculo de $((10 + 20) * (30 - 40)) / (50 + 60)$	Imprime -2.73
<i>n</i> 1 1	[1] ← 1	<i>A</i> ← 1
<i>n</i> 5 5	[5] ← 5	<i>E</i> ← 5
<i>p</i> 11	Imprime o cálculo de $((1 + 20) * (30 - 40)) / (5 + 60)$	Imprime -3.23
.	Fim do processamento	

Exercício adicional 1: Considere agora que cada célula pode ser referenciada mais de uma vez, mas a obtenção do valor de cada célula não deve fazer uso dela mesma. Caso isto ocorra, dizemos que a planilha está inconsistente. Faça uma rotina que verifica se uma planilha está inconsistente.

Exercício adicional 2: Considere agora uma planilha bidimensional, onde cada célula deve ser especificada por dois índices. Faça o mesmo programa, com as devidas modificações, para trabalhar com este tipo de planilha.

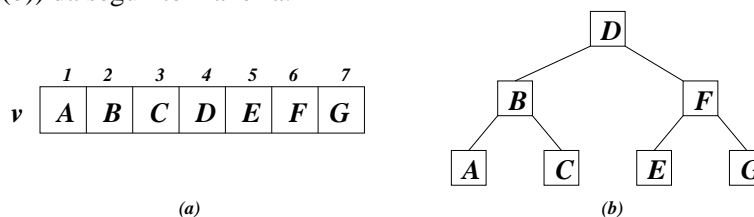
10.5 Exercícios

1. Faça uma função recursiva para encontrar um elemento em um vetor. A rotina deve ter como parâmetros: o vetor, o número de elementos do vetor (o primeiro elemento do vetor começa no índice 1), e um valor a ser procurado. A rotina retorna o índice do elemento no vetor, caso este se encontre no vetor, -1 caso contrário.
2. Faça uma rotina recursiva para imprimir os elementos de um vetor, na ordem do menor índice primeiro.
3. Faça uma rotina recursiva para imprimir os elementos de um vetor, na ordem do maior índice primeiro.
4. A função de Ackerman é definida recursivamente nos números não negativos como segue:

$$\begin{aligned} a(m, n) &= n + 1 && \text{Se } m = 0, \\ a(m, n) &= a(m - 1, 1) && \text{Se } m \neq 0 \text{ e } n = 0, \\ a(m, n) &= a(m - 1, a(m, n - 1)) && \text{Se } m \neq 0 \text{ e } n \neq 0. \end{aligned}$$

Faça um procedimento recursivo para computar a função de Ackerman. Obs.: Esta função cresce muito rápido, assim ela deve poder ser impressa para valores pequenos de m e n .

5. Faça uma rotina recursiva para imprimir todas as permutações dos números de 1 a n , uma permutação por linha.
6. Dado uma cadeia de caracteres de comprimento n , escreva uma rotina recursiva que inverte a seqüência dos caracteres. I.e., o primeiro caracter será o último e o último será o primeiro.
7. Um vetor tem $2^k - 1$ valores inteiros (figura (a)), onde k é um inteiro positivo, $k \geq 1$. Este vetor representa uma figura hierárquica (figura (b)) da seguinte maneira:



Você pode imaginar que este vetor está representando uma árvore genealógica de 3 níveis. Infelizmente, o usuário do programa que faz uso deste vetor necessita de algo mais amigável para ver esta estrutura. Faça uma rotina recursiva que dado este vetor v e o valor k , imprime as seguintes linhas:

```

      G-----
     F-----
    E-----
   D-----
  C-----
 B-----
A-----
  
```

Note que fica bem mais fácil para enxergar a hierarquia visualizando este desenho. A profundidade que é impresso cada elemento é feita controlando os espaços definidos para a profundidade de elemento na hierarquia.

8. Escreva uma função recursiva para calcular o *máximo divisor comum*, **mdc**, de dois inteiros positivos da seguinte maneira:

$$\begin{aligned} \mathbf{mdc}(x, y) &= y && \text{se } (y \leq x) \text{ e } x \bmod y = 0; \\ \mathbf{mdc}(x, y) &= \mathbf{mdc}(y, x) && \text{se } (x < y); \\ \mathbf{mdc}(x, y) &= \mathbf{mdc}(y, x \bmod y) && \text{caso contrário.} \end{aligned}$$

9. Cálculo de determinantes por co-fatores. Seja A uma matriz quadrada de ordem n . O *Menor Complementar* M_{ij} , de um elemento a_{ij} da matriz A é definido como o determinante da matriz quadrada de ordem $(n - 1)$ obtida a partir da matriz A , excluindo os elementos da linha i e da coluna j . O *Co-Fator* α_{ij} de A é definido como:

$$\alpha_{ij} = (-1)^{i+j} M_{ij} .$$

O determinante de uma matriz quadrada A de ordem n pode ser calculado usando os co-fatores da linha i da seguinte maneira:

$$\mathbf{det}(A) = \alpha_{i1}A_{i1} + \alpha_{i2}A_{i2} + \dots + \alpha_{in}A_{in}.$$

O mesmo cálculo pode ser feito pelos co-fatores da coluna j da seguinte maneira:

$$\mathbf{det}(A) = \alpha_{1j}A_{1j} + \alpha_{2j}A_{2j} + \cdots + \alpha_{nj}A_{nj}.$$

Faça uma rotina recursiva para calcular o determinante de uma matriz de ordem n usando o método descrito acima, onde a rotina tem o seguinte cabeçalho:

function determinante(var A :*TipoMatrizReal*; n :**integer**):**real**;

onde *TipoMatrizReal* é um tipo adequado para definir matriz e n é a ordem da matriz.

Obs.: Existem na literatura outros métodos mais eficientes para se calcular o determinante.

10. Seja $v = (v_1, \dots, v_i, \dots, v_f, \dots, v_n)$ um vetor com n dígitos entre 0 e 9. Faça uma rotina recursiva para verificar se os elementos (v_i, \dots, v_f) formam um número palíndromo, $1 \leq i \leq f \leq n$. Obs.: Pode considerar que os números podem começar com alguns dígitos 0's, assim, o número 0012100 é palíndromo.