8 Recursão

Onde é introduzida a técnica de recursão e como pode ser utilizada para representar e resolver problemas de forma mais intuitiva.

Pré-requisito(s):	Objetivos (ao final você deverá ser capaz de):
• Saber modularizar programas com funções e procedimentos • Saber criar e utilizar métodos em Java • Compreender os princípios de ordenação de lista de valores	• Conhecer o conceito de recursão • Identificar quando um problema pode ser facilmente resolvido via recursão • Como escrever e utilizar métodos recursivos em java • Como o sistema trata as chamadas recursivas • Utilizar algoritmos de ordenação recursivos

No capítulo 6 vimos como decompor um problema em pedaços mais simples, estruturando-o através de módulos. Vimos que esses módulos são classificados em procedimentos ou funções de acordo com a existência de um valor de retorno ou não. Vimos como os módulos podem realizar chamadas a outros módulos mais específicos. Neste capítulo veremos como módulos podem realizar chamadas a eles mesmos. Esse processo é conhecido como chamada recursiva. O princípio norteador da técnica de indução é a definição de um caso típico de um determinado problema para o qual sabe-se a solução e então modulariza-se o problema de forma a que o mesmo seja representado em versões mais simples dele próprio. O computador fica encarregado de montar uma escadaria de solução, partindo-se de um caso n que se deseja resolver até o caso trivial ou conhecido do problema exposto. O capítulo mostra o poder de expressividade da recursão e como o sistema trata as chamadas recursivas. Dando continuidade aos algoritmos de pesquisa e ordenação apresentados no capítulo anterior, veremos alternativas

recursivas. Finalmente, o capítulo traz dois problemas interessantes como estudos de caso de recursão que comprovam o poder de expressão da técnica.

8.1 O Poder de Expressão da Recursão

O princípio fundamental da programação recursiva é que a solução de um determinado problema é definida em termos de uma solução análoga, embora mais simples, de si própria.

Essa característica dá às definições recursivas o poder de definir um conjunto infinito utilizando uma expressão finita, e aos programas recursivos o poder de especificar um número arbitrariamente grande de cálculos sem conter explicitamente estruturas de repetição.

Um módulo recursivo executa sucessivas chamadas de si mesmo até se atingir uma versão do problema para o qual se conheça a solução. Essa versão do problema é conhecida como **caso-base** (ou terminal), enquanto que a versão original do problema é chamada de **caso-geral**.

Dessa forma, para que um problema possa ser resolvido através de chamadas recursivas, regras devem ser definidas de tal forma que os casos mais gerais (mais complexos) do problema possam ser solucionados conhecendo-se apenas a solução do caso-base.

A recursão é bastante utilizada na matemática para demonstrações através da **indução**.

Um exemplo típico é a definição da função *fatorial.* O fatorial de um número inteiro n ($n!$) é computado através do seguinte produto:

$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \times \ldots \times 1$

Sabendo que o fatorial do número 0 (zero) é igual a 1 (um) (e esse é o caso-base do problema), podemos definir a função fatorial de forma recursiva:

$n! = n \times (n - 1)!, \ n \in \mathbb{N}$

Note, por sua vez, que

$(n - 1)! = (n - 1) \times (n - 2)!$

No momento em que o termo $(n - i)$ atingir o valor 0 (zero), ele é substituído automaticamente pelo valor de seu fatorial, que já é conhecido a priori por ser o caso-base do problema ($0! = 1$).

Podemos dizer, por exemplo, que

$4!$

$= 4 \times 3!$

$= 4 \times 3 \times 2!$

$= 4 \times 3 \times 2 \times 1!$

$= 4 \times 3 \times 2 \times 1 \times 0!$

$= 4 \times 3 \times 2 \times 1 \times 1$

$= 4 \times 3 \times 2 \times 1$

$= 4 \times 3 \times 2$

$= 4 \times 6$

$= 24$

Observe como o problema inicial (4!) foi decomposto em subproblemas cada vez menores, utilizando a mesma definição (!), até se chegar a um micro-problema cuja solução conhecida (0!) foi sendo utilizada de volta para resolver os subproblemas que ficaram pendentes.

A figura 8.1 traz a implementação da função fatorial recursiva em Java. A função é definida na linha 1 e o caso base é representado pela decisão da linha 2. Note a invocação recursiva na linha 3. Assim como na definição matemática por indução, a invocação recursiva é feita passando-se como parâmetro o valor anterior subtraído de 1 (um).

```
1.  public static long fatorial(int n) {
2.    if (n==0) return 1;
3.    return n * fatorial(n-1);
4.  }
```

Figura 8.1 – Função fatorial recursiva em Java.

Um outro problema clássico que ilustra claramente o poder de expressividade da recursão é a série de *Fibonacci*. Como vimos, a série se inicia com dois valores inteiros, 0 e 1, e possui a propriedade de que cada número subsequente seja a soma dos dois números anteriores da série.

O problema do cálculo do i-ésimo termo da série de Fibonacci pode ser definido recursivamente da seguinte forma:

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)

Note que há dois casos-base para o cálculo do i-ésimo termo da série: o primeiro termo (índice 0) é o número 0 (zero) e o segundo termo (índice 1) é o número 1 (um). A partir daí qualquer termo n pode ser obtido efetuando-se a soma de seus dois imediatamente sucessores.

A figura 8.2 traz a implementação da função fibonacci recursiva em Java. Note a definição dos dois casos-base nas linhas 2 e 3. As duas invocações recursivas são feitas na linha 4.

```
1.  public static long fibonacci(int k) {
2.    if (k == 0) return 0;
3.    if (k == 1) return 1;
4.    return (fibonacci(k-1) + fibonacci(k-2));
5.  }
```

Figura 8.2 – Função fibonacci recursiva em Java.

Se você comparar as implementações recursivas dos dois problemas com as versões iterativas (que fazem uso explícito de laços de repetição) já vistas em capítulos anteriores, perceberá o poder de expressividade da definição recursiva de um determinado problema [*nota: não confundir expressividade, clareza, concisão com desempenho; implementações recursivas normalmente possuem eficiência computacional bem mais baixa que as das suas correspondentes versões iterativas - quando estas são possíveis*].

8.2 Anatomia de uma chamada recursiva

Um método recursivo é implementado pelo compilador através de uma pilha de chamadas do método, conhecida como **pilha de execução**. Como o próprio nome sugere, a pilha de execução de um programa empilha os dados usados em cada chamada que ainda não terminou de processar. Esses dados são armazenados na memória usando uma estrutura de dados do tipo registro, chamada de **registro de ativação**.

Um registro de ativação contém campos para armazenar:

- Valores ou referências dos parâmetros da função e de variáveis locais
- Endereço de retorno para retomar o controle pelo ativador
- Ponteiro para o registro de ativação do ativador
- Valor retornado da função

Considere como exemplo uma função recursiva em Java que calcula a potência de um número (x^n) e o programa principal que realiza uma chamada a essa função. A cada linha de código é atribuído um número pelo compilador. Se a linha representa uma chamada de função, seu número será um endereço de retorno (figura 8.3).

```
1.  ...
2.    public double potencia(double x, int n) {              (102)
3.           if (n==0)                                       (103)
4.               return 1.0;                                 (104)
5.           return x*potencia(x,n-1);                       (105)
6.    }
7.
8.    public static void main(String[] args) {
```

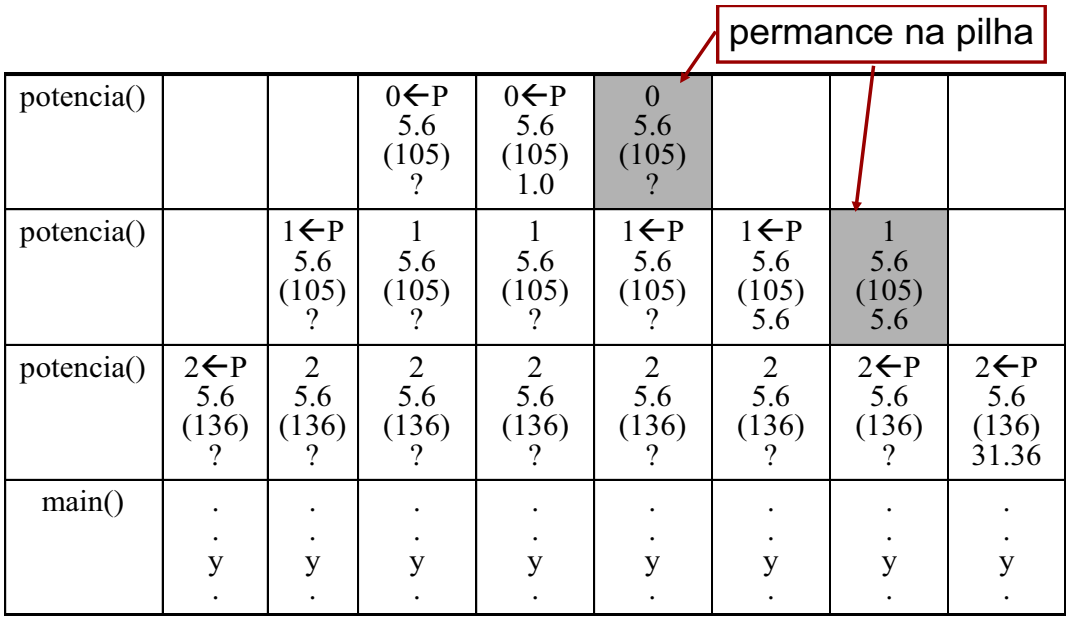
```
9.            ...
10.           y = potencia(5.6,2);                    (136)
11.   }
```

Figura 8.3 – Cálculo da potência de um número com uso de recursão.

Quando a linha 10 é executada, a função potencia é invocada pela primeira vez e depois se auto invoca por duas vezes, subtraindo por 1 (um) o valor de seu segundo argumento até que atinja o valor 0 (zero), caso-base do problema (todo número elevado a 0 é igual a 1). A ordem de execução das chamadas pode ser observada abaixo:

```
chamada 1: potencia(x,2)
chamada 2:              potencia(x,1)
chamada 3:                       potencia(x,0)
chamada 3:                       1
chamada 2:              x
chamada 1: x*x
```

A configuração da pilha de execução na memória durante o processamento do código está ilustrada na figura 8.4. Cada coluna representa a configuração da pilha em cada instante de execução. A identificação dos campos do registro também está ilustrado na figura.



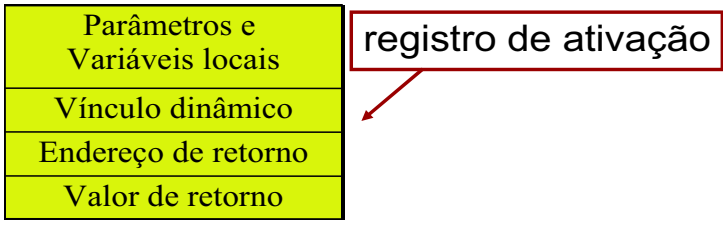


Figura 8.4 – Pilha de execução na memória durante o processamento de uma chamada recursiva e campos de um registro de ativação.

8.3 Algoritmos Recursivos de Pesquisa e Ordenação

8.3.1 Pesquisa Binária

O algoritmo de pesquisa binária de elementos em uma seqüência é caracterizado por um padrão de computação recorrente: a seqüência original é dividida ao meio e uma das subseqüências geradas passa pelo mesmo processo de divisão até que o valor procurado seja encontrado ou não haja mais possibilidade de divisão.

O problema de pesquisa binária é intuitivamente resolvido com uma abordagem recursiva. Nesse caso, temos dois casos-base claros: (1) valor é encontrado e (2) não há possibilidade de divisão. O passo recursivo realiza uma chamada do próprio método passando a metade mais à esquerda, caso a chave de busca seja menor que o valor central ou passando ametade mais à direita, caso contrário.

O algoritmo recursivo de pesquisa binária está ilustrado na figura 8.5. O algoritmo está representado como uma função (pesquisaBin) que recebe como parâmetros o vetor de elementos, a chave procurada, o índice referente ao limite inferior e o índice referente ao limite superior. Como exemplo, o método principal do programa aplica a função para o vetor ilustrado na figura 7.4 para pesquisar se o valor 15 se encontra no mesmo. Note que a primeira chamada da função (linha 18), o limite inferior de busca é o índice 0 (zero) e o superior é o índice 12.

```
1.  public class PesquisaBinariaRecursivo {
2.
3.      public static int pesquisaBin(int[] a, int chave, int
4.                                    inferior, int superior) {
5.          int n;
6.          if (inferior > superior) return -1; // caso-base 1
7.          n = (inferior + superior) / 2;
8.          if (a[n] == chave) return n; // caso-base 2
```

```
9.            if (a[n] < chave)
10.                   return pesquisaBin(a,chave,n+1,superior);
11.           else
12.                   return pesquisaBin(a,chave,inferior,n-1);
13.           }
14.
15.     public static void main(String[] args) {
16.           int[] vetor = {2, 9, 11, 15, 28, 33, 40, 51, 64, 76,
17.                          77, 82, 94};
18.           int indice = pesquisaBin(vetor, 15, 0, 12);
19.           System.out.println("Índice do valor procurado: "
20.                              +indice);
21.     }
22. }
```

Figura 8.5 – Versão recursiva do método de pesquisa binária.

8.4 Quicksort

O *quicksort* é o algoritmo de ordenação em memória interna mais rápido que se conhece para uma grande variedade de situações. É provavelmente o algoritmo de ordenação mais utilizado.

O *quicksort* adota a estratégia de **divisão e conquista**. A idéia básica de algoritmos que adotam essa estratégia é dividir o problema de ordenação de uma seqüência de n elementos em dois problemas de ordenação menores. A seguir, os problemas menores são ordenados de forma independente e os resultados das ordenações são combinados para compor a solução do problema original.

O princípio básico do *quicksort* é **particionar** o vetor original em dois pedaços, esquerdo e direito, de tal forma que as chaves dos elementos da partição esquerda sejam menores que uma determinada chave do vetor, previamente escolhida de forma aleatória, e denominada de **pivô**, e que as chaves dos elementos da partição direita sejam maiores que essa chave pivô.

O algoritmo pode ser resumido nos seguintes passos:

1. Escolha aleatoriamente um elemento do vetor e chame-o de pivô;
2. Rearranje o vetor de forma que todos os elementos em posições anteriores à do pivô possuam chaves menores que a dele, e todos os elementos posteriores possuam chaves maiores. Ao fim do processo o pivô estará em sua posição final e haverá dois sub-vetores não ordenados. Essa operação é denominada **partição**;
3. Recursivamente ordene a sublista dos elementos menores e a sublista dos elementos maiores;
4. A **base** da recursão são as listas de tamanho zero ou um, que estão sempre

ordenadas. O processo é finito, pois a cada iteração pelo menos um elemento é posto em sua posição final e não será mais manipulado na iteração seguinte.

O *quicksort* está ilustrado na figura 8.6.

```
1.  import java.util.Random;
2.
3.  public class Quicksort {
4.     public static final Random RND = new Random();
5.
6.     private static void troca(int[] v, int i, int j) {
7.          int tmp = v[i];
8.          v[i] = v[j];
9.          v[j] = tmp;
10.    }
11.
12.    private static int particao(int[] v, int begin, int end) {
13.         int indice = begin + RND.nextInt(end - begin + 1);
14.         int pivo = v[indice];
15.         troca(v, indice, end);
16.         for (int i = indice = begin; i < end; ++i) {
17.             if (v[i] < pivo) {
18.                 troca(v, indice++, i);
19.             }
20.         }
21.         troca(v, indice, end);
22.         return (indice);
23.     }
24.
25.    private static void quicksort(int[] v, int begin, int end)
26.    {
27.         if (end > begin) {
28.             int indice = particao(v, begin, end);
29.             quicksort(v, begin, indice - 1);
30.             quicksort(v, indice + 1, end);
31.         }
32.     }
33.
34.    public static void main(String[] args) {
35.         int[] vetor = {94, 28, 51, 77, 11, 33, 82, 15, 64,
36.                        76, 2, 40, 9};
37.         quicksort(vetor, 0, 12);
38.         System.out.print("Vetor ordenado: ");
39.         for (int i=0; i<13; i++)
40.           System.out.print(vetor[i]+" ");
41.      }
42. }
```

Figura 8.6 – Algoritmo recursivo de ordenação *quicksort*.

O programa principal define estaticamente um vetor de 13 elementos inteiros a serem ordenados de forma crescente (linha 35). O método *quicksort* é então invocado na linha 37, passando como parâmetros o vetor em questão, o índice do

primeiro elemento (`begin`) e o índice do último elemento (`end`) do vetor. O método então, definido na linha 25, invoca um outro método chamado `particao` que irá definir o índice onde o vetor será dividido (linha 28). De posse desse índice, o método realiza duas chamadas recursivas, uma para cada metade do vetor particionado na posição indicada pelo índice (linhas 29 e 30). Chamadas recursivas sucessivas são realizadas até que as variáveis `begin` e `end`, que representam o início e fim de um vetor, respectivamente, armazenem o mesmo valor de índice. Quando isso ocorre significa que o vetor a ser ordenado é composto de apenas um elemento e, portanto, já está ordenado. A pilha de execução é desempilhada e o vetor original, ordenado, é impresso (linha 40).

8.5 Mergesort

A operação de unir duas listas de valores ordenadas gerando uma terceira lista ordenada é normalmente denominada de **intercalação (merge)**. Essa operação consiste em colocar na terceira lista o elemento de menor valor entre os menores das duas listas iniciais, desconsiderando esse mesmo elemento nos passos posteriores. O processo se repete até que não restem mais elementos nas listas originais.

Essa idéia é utilizada em um algoritmo de ordenação chamado de **Mergesort**. O processo consiste nos seguintes passos:

1) dividir recursivamente o vetor a ser ordenando em dois vetores até obter n vetores de um único elemento;
2) aplicar o algoritmo de intercalação tendo como entrada dois vetores de um elemento e formando um vetor ordenado de dois elementos;
3) repetir esse processo formando vetores ordenandos cada vez maiores até que todo o vetor esteja ordenando.

A figura 8.7 ilustra o algoritmo Mergesort em Java. O método `mesclar` (linha 11) realiza a intercalação de dois vetores. O método `mergesort` realiza duas chamadas recursivas, uma para cada metade do vetor a ser ordenado (linhas 6 e 7).

```
1.  public class Mergesort {
2.    private static void mergesort(int[] v, int inicio, int fim)
3.    {
4.        if (inicio < fim) {
5.            int meio = (inicio + fim) / 2;
6.            mergesort(v, inicio, meio);
7.            mergesort(v, meio + 1, fim);
8.            mesclar(v, inicio, meio, fim);
9.        }
10.   }
```

```
11.    private static void mesclar(int[] v, int inicio, int meio,
12.                                int fim) {
13.        int tamanho = fim - inicio + 1;
14.        int[] temp = new int[tamanho];
15.        for (int posicao = 0; posicao < tamanho; posicao++)
16.           temp[posicao] = v[inicio + posicao];
17.        int i = 0;
18.        int j = meio - inicio + 1;
19.        for (int posicao = 0; posicao < tamanho; posicao++) {
20.           v[inicio + posicao] =
21.              (j <= tamanho - 1) ?
22.                    ((i <= meio - inicio) ?
23.                              (temp[i] < temp[j]) ?
24.                                       temp[i++]
25.                              : temp[j++]
26.                    : temp[j++]
27.              : temp[i++];
28.        }
29.     }
30.     public static void main(String[] args) {
31.         int[] vetor = {94, 28, 51, 77, 11, 33, 82, 15, 64, 76,
32.                        2, 40, 9};
33.         mergesort(vetor, 0, 12);
34.         System.out.print("Vetor ordenado: ");
35.         for (int i=0; i<13; i++)
36.            System.out.print(vetor[i]+" ");
37.     }
38. }
```

Figura 8.7 – Algoritmo recursivo de ordenação *mergesort*.

8.6 Dois problemas recursivos

Esta seção apresenta dois problemas com solução recursiva e que não envolvem uma formulação matemática simples.

8.6.1 Anagramas

Um anagrama é o nome dado a uma palavra que é formada a partir da recombinação dos caracteres de uma String original. Ou seja, o problema em questão trata da geração de todas as permutações de uma string de texto. As permutações possíveis para a string “OAN”, por exemplo, seriam: ANO, AON, NAO, NOA, OAN, ONA.

O raciocínio recursivo a ser empregado na solução do problema é que as permutações para cada caractere são seguidas por permutações dos caracteres restantes. O processo se repete até que reste apenas um caractere, onde o próprio caractere é a única permutação possível.

Dessa forma, temos determinado o caso base (um único caracteres restante) e o passo recursivo que inclui uma chamada recursiva para cada caractere na string, com um caractere diferente utilizado como o primeiro caractere da permutação.

O código Java para solução do problema está apresentado na figura 8.8.

```
1.  public class Anagrama {
2.     public static void gereAnagramas(String inicio,
3.                                      String fim) {
4.        if ( fim.length() <= 1 )
5.           System.out.println( inicio + fim );
6.        else
7.           for ( int i = 0; i < fim.length(); i++ ) {
8.              String novaString = fim.substring( 0, i ) +
9.                                  fim.substring( i+1 );
10.             gereAnagramas( inicio + fim.charAt( i ),
11.                            novaString );
12.          }
13.    }
14.    public static void main( String args[] ) {
15.       System.out.print( "Forneça a string: " );
16.       String texto = System_in.readString();
17.       Anagrama.gereAnagramas("", texto );
18.    }
19. }
```

Figura 8.8 – Solução do problema dos anagramas com recursão em Java

Observe que o caso básico está representado nas linhas 4 e 5, quando a string que será permutada for formada por apenas um caractere, no máximo. O passo recursivo ocorre na linha 10.

8.6.2 Torres de Hanoi

As *Torres de Hanoi* é um dos problemas clássicos mais conhecidos pelo profissional de Ciência da Computação. Sua resolução por métodos convencionais é extremamente complicada. Mas uma abordagem recursiva simplifica sobremaneira a solução do problema.

O problema consiste de 3 torres verticais, nos quais podem ser colocados n discos de diâmetros diferentes furados no centro. O problema é iniciado com todos os discos na torre esquerda e o objetivo é movimentar todos os discos para a torre mais à direita de acordo com as seguintes regras:

(1) só se pode movimentar um disco de cada vez;

(2) em cada torre, só se pode movimentar o disco de cima;

(3) nunca se pode colocar um disco sobre outro disco de diâmetro menor

A abordagem recursiva para o problema com n discos consiste em apenas três passos:

(P1) movimentar n - 1 discos da torre da esquerda para a torre do centro utilizando a torre da direita como auxiliar;

(P2) movimentar o último disco da torre da esquerda para a torre da direita;

(P3) movimentar os n - 1 discos da torre do centro para a torre da direita utilizando a torre da esquerda como auxiliar.

Note que esse passos levam necessariamente à movimentação de um único disco no final, que é nosso caso base.

```
1.  public class Hanoi {
2.       public static void resolverTorre(int n, char inicio,
3.                                        char fim, char temp) {
4.            if(n == 1)
5.                System.out.println(inicio+"\t"+ fim);
6.            else {
7.                resolverTorre(n - 1, inicio, temp, fim);
8.                System.out.println(inicio+"\t"+ fim);
9.                resolverTorre(n - 1, temp, fim, inicio);
10.           }
11.      }
12.      public static void main(String[] args) {
13.           resolverTorre(3, 'A', 'C', 'B');
14.      }
15. }
```

Figura 8.9 – Solução do problema das Torres de Hanoi com recursão em Java.

A solução testa o método recursivo `resolverTorre` para apenas três discos (linha 13). O algoritmo, entretanto, é geral e funciona para `n` discos. Os passos recursivos (P1) e (P3) estão representados nas linhas 7 e 9. A movimentação do último disco restante da torre (P2) está representada na linha 8. O caso base está representado na linha 5.

Resumo

- Recursão é uma técnica de decomposição de problema que consiste em redefinir o problema em versões mais simples dele mesmo.
- Uma solução com abordagem recursiva é formada por um caso base e pelo caso geral, que invariavelmente deve chegar ao caso base.
- O caso base consiste de um caso simples do problema original para o qual a solução é previamente conhecida. Através de um processo de indução, o caso geral consegue ser resolvido considerando o caso base.

- Um método recursivo realiza chamada a ele próprio.
- O algoritmo de ordenação de vetores mais utilizado e eficiente é um algoritmo recursivo chamado *quicksort*. Um outro exemplo de algoritmo de ordenação recursivo é o *mergesort*.

Na próxima aula...

… veremos como podemos armazenar dados e informações processados, de forma persistente em arquivos em um dispositivo de memória secundária, como nosso HD.

Referências e Sugestões de Leitura

Recursão e análise de desempenho de algoritmos recursivos são tratados de forma superficial no capítulo 2 de

ZIVIANI, Nivio. Projeto de Algoritmos com implementações em java e C++, Thomson, 2007.

Uso de recursão em Java é abordado no capítulo 15 de

DEITEL, H., DEITEL, P. Java como Programar, Prentice-Hall, 2005.

Exercícios Propostos

8.1 – O máximo divisor comum (m.d.c.) entre dois números inteiros pode ser calculado por:

$$mdc(m,n) = \begin{cases} m & se\ n = 0 \\ mdc(n, mod(m,n)) & se\ n \neq 0 \end{cases}$$

Escreva um método (recursivo) em Java para implementá-lo.

8.2 – Escreva um método (recursivo) em Java para implementar a Função de Ackermann $A(m,n)$ $(m, n \geq 0)$ definida da seguinte forma:

$$A(m,n) = \begin{cases} 1 & se\ m = 0 \\ A(m-1,1) & se\ m > 0\ e\ n = 0 \\ A(m-1, A(m,n-1)) & se\ m > 0\ e\ n > 0 \end{cases}$$

8.3 – Escreva um método (recursivo) Java para calcular o valor aproximado de $sen(x)$ utilizando a seguinte série:

$$sen(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

8.4 – O número de combinações de m objectos n a n pode ser calculado pela seguinte função:

$$C(m,n) = \begin{cases} 1 & se\ n = 0 \\ 1 & se\ n = m \\ C(m-1,n) + C(m-1,n-1) & se\ m > n\ e\ m > 0\ e\ n > 0 \end{cases}$$

Escreva um método (recursivo) Java para implementar a função *C(m,n)*.

8.5 – Seja a sucessão u_n $(n \geq 0)$ definida da seguinte forma:

$$u_n = \begin{cases} 1 & se\ k = 0 \\ 3 & se\ k = 1 \\ 5 & se\ k = 2 \\ u_{k-3} - u_{k-2} + u_{k-1} & se\ k \geq 3 \end{cases}$$

Escreva um método (recursivo) Java para gerar os *n* primeiros termos de u_n.

8.6 – Escreva um método (recursivo) Java para calcular a soma dos n (>0) primeiros números inteiros não negativos.

8.7 – Escreva um método (recursivo) Java para determinar o número de zeros que ocorre num número inteiro.

8.8 – Escreva um método recursivo Java para converter cada dígito de um número inteiro no nome correspondente (e.g. 562 corresponde a "cinco seis dois").

8.9 – Escreva um método (recursivo) Java para determinar o mínimo número inteiro num vetor de números inteiros.

8.10 – Um problema bem conhecido na área de computação é o problema das Oito Rainhas. O problema consiste em conseguir colocar 8 rainhas em um tabuleiro de xadrez de tal forma que nenhuma ataque alguma outra, ou seja, duas rainhas quaisquer não podem estar na mesma linha, nem na mesma coluna, nem na mesma diagonal. Resolva o problema recursivamente.

[*Dica: sua solução deve iniciar com a primeira coluna e procurar uma localização nessa coluna em que uma rainha pode ser colocada – inicialmente, coloque a rainha na primeira linha. A solução deve então pesquisar recursivamente as colunas restantes. Se uma coluna for alcançada e não houver nenhuma posição possível para uma rainha, o programa deve retornar para a coluna anterior e mover a rainha nessa coluna para uma nova linha.*]