ECM404 - Estruturas de Dados e Técnicas de Programação

Exercícios - Recursividade

1. **Elaborar** um programa que some os elementos de um vetor de tamanho `N` do tipo `double` utilizando uma função recursiva. A função recursiva que realiza a soma pode ser assim definida, onde *v* é um vetor de *n* elementos:

$$soma(v,n) = \begin{cases} v[0], & \text{se } n = 1 \\ soma(v, n-1) \; + \; v[n-1], & \text{se } n > 1 \end{cases}$$

2. A seguir tem-se uma implementação em C de uma função que executa busca linear de um valor inteiro *x* em um vetor inteiro *v* de tamanho *n*:, retornado a sua posição, se encontrado, ou o valor -1, caso não seja encontrado:

```c
int busca_linear(int x, int *v, int n) {
    int pos = -1;
    int i;
    for (i = 0; i < n; i++)
        if (v[i] == x) {
            pos = i;
            break;
        }
    return pos;
}
```

Pede-se: elaborar uma versão recursiva da função `busca_linear()`. **Dica**: será necessário adicionar um parâmetro na função para controlar o ponto onde se inicia a busca.

3. A operação de elevar um número real *x* a *n*-ésima potência (valor inteiro não negativo) pode ser recursivamente definida assim:

$$pot(x,n) = \begin{cases} 1, & \text{caso } n = 0 \\ pot(x \times x, \; n \; \div 2), & \text{caso } n \text{ seja par} \\ pot(x, n-1) \times x, & \text{caso } n \text{ seja ímpar} \end{cases}$$

Onde × é a operação de multiplicação e ÷ é a operação de divisão inteira. **Pede-se**: elaborar e testar a função `pot()`, recursiva, em C.

4. A fórmula da adição é uma importante identidade binomial e é assim definida, de modo recursivo:

$$\binom{n}{m} = \begin{cases} 1, & \text{se}\, n \neq 0\, \text{e}\, m = 0 \\ 1, & \text{se}\, n = m \\ \binom{n-1}{m-1} + \binom{n-1}{m}, & \text{para os demais casos} \end{cases}$$

   Onde $n, m \in \mathbb{N}$. **Elaborar e testar** uma função recursiva em C denominada `comb(n,m)` de modo a calcular o valor de $\binom{n}{m}$. Comparar os resultados de testes com a definição: $\binom{n}{m} = \frac{n!}{m!(n-m)!}$.

5. O algoritmo de Euclides, que calcula o máximo divisor comum entre dois números inteiros *m* e *n* ($0 \leq m < n$), é assim definido recursivamente:

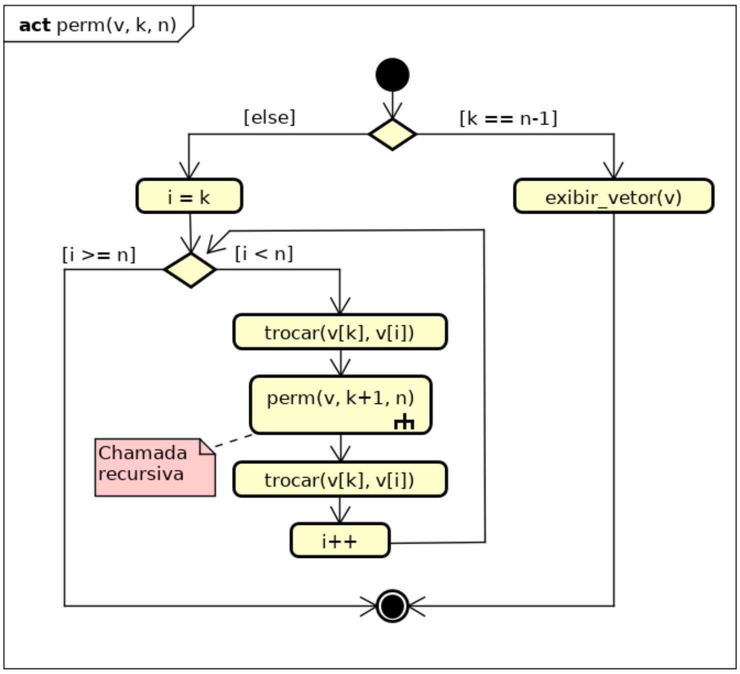
$$mdc(m, n) = \begin{cases} n, & \text{para}\, m = 0 \\ mdc(n \bmod m,\, m), & \text{para os demais casos} \end{cases}$$

   Onde *x* mod *y* é a operação que resulta o resto da divisão entre *x* e *y*. **Implementar e testar** a função recursiva `mdc()` em C.

6. Pode-se gerar (exibir na tela) todas as permutações de n elementos de um vetor assim, usando *n*=4 como exemplo e o vetor v = (1, 2, 3, 4):
   - Gerar 1 seguido de todas as permutações de (2, 3, 4);
   - Gerar 2 seguido de todas as permutações de (1, 3, 4);
   - Gerar 3 seguido de todas as permutações de (1, 2, 4);
   - Gerar 4 seguido de todas as permutações de (1, 2, 3).

   Perceber que, nesta descrição, está implícita uma função recursiva: para resolver um problema de tamanho 4, deve-se resolver um problema de tamanho 3, e assim por diante.

   O diagrama de atividades a seguir representa a função `perm`, que gera todas as permutações de elementos de um vetor *v* de tamanho *n* (considerar um vetor inteiro) e usando *k* para avançar nos subvetores - estude-o para se convencer que ele gera todas as permutações:



Pede-se:

(a) Traduzir este diagrama para a função C equivalente. Admitir que *v* é um vetor inteiro, *n* é o tamanho do vetor e *k* é uma variável inteira;

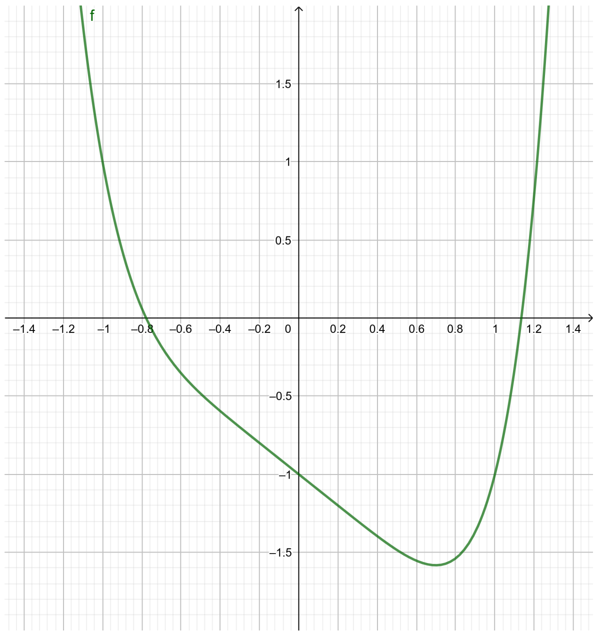
(b) Testar esta função em um programa em C. Por exemplo:

```
int main(void) {
    int v[] = {1, 2, 3, 4};
    perm(v, 0, 4);
    return 0;
}
```

Observe que o valor de *k* deve ser zero na primeira chamada. O resultado será:

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
1 4 2 3
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 3 1
2 4 1 3
3 2 1 4
3 2 4 1
3 1 2 4
3 1 4 2
3 4 1 2
3 4 2 1
4 2 3 1
4 2 1 3
4 3 2 1
4 3 1 2
4 1 3 2
4 1 2 3
```

7. O método da bissecção permite determinar uma raiz real de uma função contínua em um intervalo [*a*,*b*] da seguinte forma (acompanhe pela figura a seguir):



A figura apresenta o gráfico da função $x^6 - x - 1$. Observar que ela possui duas raízes reais, que são -0.77811 e 1.13474 e que é contínua no intervalo apresentado pela figura. O método da bissecção para determinar uma raiz real de uma função contínua em um intervalo [*a*, *b*] pode ser assim descrito:

1. Definir o intervalo [*a*, *b*] de interesse. Se a função em questão for contínua nesse intervalo e se *f*(*a*). *f*(*b*) < 0, então certamente haverá uma raiz nesse intervalo; senão, terminar sem calcular raiz alguma;
2. Calcular o ponto médio do intervalo: $c = a + \frac{(b-a)}{2} = \frac{(a+b)}{2}$;
3. Se o ponto *c* estiver próximo suficiente da solução (dentro de uma tolerância adequada), retorne *c* como uma raiz e então termine. Existem várias formas de calcular essa proximidade - uma delas é calcular a diferença entre um dos pontos extremos do intervalo com o valor de *c*. Por exemplo, se $b - c < 0.00005$, então considerar *c* uma raiz. Mas, deve-se analisar a função em questão e o intervalo antes de definir uma tolerância;
4. Se o ponto *c* não estiver próximo suficiente da solução, então:
   a. Se $f(b) * f(c) \leq 0$ então a raiz está dentro do intervalo [*c*, *b*]. Retornar ao passo 2 fazendo *a* = *c*;
   b. Senão, a raiz está dentro do intervalo [*a*, *c*]. Retornar ao passo 2 fazendo *b* = *c*;

Pede-se:

(a) Elaborar e testar a função C bissec, que deverá ter a seguinte interface:

```
int bissec(double a, double b, double* raiz);
```

Ela deverá determinar a raiz da função $x^6 - x - 1$ com tolerância de 0.00005, como descrito anteriormente. O resultado da raiz será armazenado no ponteiro `raiz`. O retorno deverá ser 0, se nenhuma raiz for encontrada ou 1, se uma raiz for encontrada e, neste caso, o valor de `raiz` poderá ser utilizado. Exemplo de utilização:

```
double raiz;
if (bissec(1, 2, &raiz))
    printf("Encontrei raiz: %10.5f\n", raiz);
else
    printf("Não encontrei raiz!\n");

if (bissec(-1, -0.5, &raiz))
    printf("Encontrei raiz: %10.5f\n", raiz);
else
    printf("Não encontrei raiz!\n");
```

(b) A função `bissec` como definida anteriormente fica restrita ao cálculo de uma única função. A linguagem C permite passar funções como parâmetro por meio de ponteiro de função. A declaração de um ponteiro para função é realizada assim:

```
typedef double (*fptr)(double);
```

Assim, `fptr` representa um "ponteiro para função" que tem um único parâmetro do tipo `double` e que retorna um `double`. Por ele é possível incluir "funções dentro de funções" dinamicamente, deixando o programa mais flexível. Assim, a função `bissec` pode ser modificada, adicionado um parâmetro pelo qual se passará uma função cuja raiz se quer determinar:

```
int bissec(fptr f, double a, double b, double* raiz);
```

Agora, na implementação de `bissec`, utiliza-se o símbolo `f` para calcular a função no ponto desejado, por exemplo:

```
int bissec(fptr f, double a, double b, double* raiz) {
    double fa = f(a);    /* chama a função f() */
    /* ... */
}
```

Depois, criar funções separadas, que se deseja calcular as raízes. Por exemplo:

```
double g(double x) {
    return pow(x, 6) - x - 1;
}
```

E então, usar `bissec` para determinar a raiz da função `g` no intervalo desejado:

```
if (bissec(g, 1, 2, &raiz))
    printf("Encontrei raiz: %10.5f\n", raiz);
else
    printf("Não encontrei raiz!\n");
```

Por fim, a função bissec pode ficar mais flexível se adicionarmos a tolerância (`epsilon`) como parâmetro:

```
int bissec(fptr f, double a, double b, double epsilon,
double* raiz);
```

E então seu uso ficará assim (exemplo):

```
if (bissec(g, 1, 2, 0.00005, &raiz))
    printf("Encontrei raiz: %10.5f\n", raiz);
else
    printf("Não encontrei raiz!\n");
```

Tarefa: implementar a função C `bissec` com as modificações apresentadas neste item. Adicionar as seguintes funções no seu programa e calcular suas raízes:

- $2x^3 - x^2 + x - 1$
- $x^3 + 4x^2 - 10$