Ficha de Trabalho N.º 8

Versão 2024/25

Objetivos: Conceção e codificação de soluções para problemas, usando recursividade.

Conceitos Necessários à Resolução da Ficha

3.1 Conceito de Recursividade

Muitos problemas são descritos através de relações de recorrência.

Diz-se que algo é recursivo quando se define em função de si próprio.

Em programação, entende-se por recursividade a capacidade que uma linguagem tem de permitir que uma função se invoque a si própria.

A utilização de um subprograma recursivo permite desencadear um número arbitrariamente grande de repetições de instruções sem, contudo, usar explicitamente estruturas de controlo repetitivo: aplicando a relação de recorrência.

Os algoritmos recursivos são apropriados para resolver problemas que por natureza são recursivos. Alguns têm soluções recursivas simples, concisas, elegantes e para os quais é difícil desenhar soluções não recursivas com tanta clareza e simplicidade.

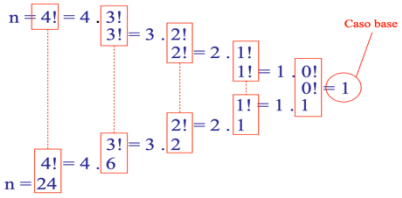
Exemplo mais comum de recursão: função Factorial

$0! = 1$ — Caso base

$1! = 1 . 0! = 1$
$2! = 2 . 1! = 2 . 1$
$3! = 3 . 2! = 3 . 2 . 1$
$4! = 4 . 3! = 4 . 3 . 2 . 1$

Regra Geral:
$n! = n * (n-1)!$
$fact(n) = n * fact(n-1)$

Ex. Factorial de 4



Caso Base ou Condição de Paragem em Recursão

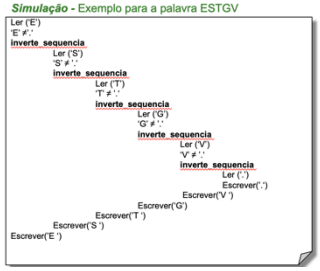
- Como uma função recursiva pode chamar-se a si mesma **indefinidamente**, é essencial a existência do caso base, ou condição de paragem.
- No caso do factorial, o caso base é o zero, cujo valor do Factorial é 1. A partir dele, são encontrados todos os outros valores.

```
// versão recursiva do programa factorial
long FactorialRec(int n)
{
    if (n == 0) return 1; // caso base, onde a recursão acaba
    else
        return n * FactorialRec(n - 1);
}
```

Exemplo - Ler uma sequência de caracteres, terminando com um '.', e escrevê-la por ordem inversa.

```
void InverterSeq()
{
    char letra;
    scanf("%c",&letra);
    if (letra != '.')
        InverterSeq();
    printf("%c", letra);
}
```

Simulação - Exemplo para a palavra ESTGV



Calcular $\sum_{i=1}^{n} i$

```
void Main()
{
    int num=0;
    printf("Calcular o Somatório de 1 a: ");
    scanf("%d", num);
    printf("Solução Recursiva - O somatório de 1 a %d é %d", num, RecursiveSomat(num));
    printf("Solução Iterativa - O somatório de 1 a %d é %d", num, IterativeSomat(num));
}
```

```
// Somatório: versão recursiva
long RecursiveSomat(int n)
{
    if ( n == 1 ) // caso base (n=1)
        return 1;
    else
        return n + RecursiveSomat(n - 1);
}
```

```
// Somatório: versão iterativa
long IterativeSomat(int n)
{
    int soma=0;
    for (int i = 1; i <= n; i++)
    {
        soma = soma + i;
    }
    return soma;
}
```

Notas sobre recursividade

A recursividade é uma forma de implementar um ciclo através de sucessivas chamadas à mesma função.

Regras para escrita de funções recursivas

- A primeira instrução de uma função recursiva deve ser a implementação do critério de paragem: a condição ou condições que se devem verificar para que a função deixe de chamar-se a si própria. Estes casos chamam-se **casos base**.
- A chamada recursiva da função deve ser sempre relativa a um subconjunto e só deve vir depois de escrito o critério de paragem.
- A recursividade permite ganhos quanto ao código escrito e, proporciona, por isso, maior legibilidade, mas resulta em perdas (geralmente pouco significativas) ao nível da performance.

Problemas Propostos

1) Calcule o valor do somatório apresentado a seguir, usando uma função recursiva (o valor de *n* deve ser pedido ao utilizador).

$$\sum_{i=1}^{n} i$$

2) Escreva uma função recursiva que permita calcular o fatorial de um valor inteiro *n* (pedido ao utilizador). Faça um programa para testar a sua função.

3) Escreva um programa em C que inclua uma função recursiva chamada potencia que permita calcular x^n (*x* real, *n* inteiro positivo).

4) Calcule o valor do somatório apresentado a seguir, usando uma função recursiva. $\sum_{i=1}^{n} \frac{r^i}{i!}$

Os valores de r (real) e n (inteiro) devem ser pedidos ao utilizador.

5) Elabore funções recursivas que permitam calcular o valor de cada um dos somatórios das alíneas a seguir. Escreva a instrução que invoque a função elaborada.

a) $x = \sum_{i=1}^{n} [(i)!]^i$ b) $x = \sum_{i=1}^{n} i*(i+1)!$ c) $x = \sum_{i=1}^{n} \frac{(i!)}{i}$ d) $x = \sum_{i=2}^{n} \frac{(i!)}{(i-1)!}$ e) $x = \sum_{i=1}^{n} \frac{(i!)^2}{i+1}$

6) Elabore várias funções para calcular o Fibonacci de um número:

a) uma, recursiva simples, aplicando diretamente a própria definição de número de Fibonacci;

b) outra, iterativa;

c) ainda outra, recursiva pro, que usa um vetor para armazenar os números de Fibonacci anteriores que depois vão ser utilizados para o cálculo do seguinte.

Para cada uma das funções criadas, calcule e mostre o tempo de execução para o cálculo do Fibonacci de um número elevado (e.g. 35 ou 40), permitindo efetuar uma avaliação comparativa do tempo de execução e consequentemente do desempenho do algoritmo respetivo.

Possivelmente, vários fatores externos podem alterar o tempo de execução (outros processos que estejam a correr no processador), pelo que, um melhoramento que se pode implementar é fazer várias medições para o tempo de execução de cada função e tomar valores médios e mostrar o desvio máximo também.

Obs. Para calcular o tempo de execução de cada tipo de implementação pode usar-se:

```
#include <time.h>
#include <sys/time.h>
```

Depois, colocar a definição dos dados:

```
struct timeval t1, t2;
double elapsedTime;
```

Quando se pretender iniciar a contagem, colocar o código:

```
// iniciar o cronómetro
gettimeofday(&t1, NULL);
```

Colocar depois o código de que se pretende calcular o tempo de execução, neste caso a chamada à função:

```
//
// Código....
//
```

Quando se pretender finalizar a contagem e mostrar o tempo decorrido:

```
// Calcular e imprimir o tempo decorrido em milisegundos
elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0;
elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0;

printf("Texto informativo do processo que foi executado Terminado em %.9f milisegundos. \n\n",
elapsedTime);
```