# Recursividade

# Recursividade

Alguns problemas são definidos com base nos mesmos, ou seja, podem ser descritos por instâncias do próprio problema.

Para tratar estas classes de problemas, utiliza-se o conceito de recursividade.

Uma função recursiva é uma função que em sua seção de comandos chama a si mesma.

Uma grande vantagem da recursividade é o fato de gerar uma redução no tamanho do algoritmo (programa), permitindo descrevê-lo de forma mais clara e concisa.

UNIVASF

# Recursividade

Porém, todo cuidado é pouco ao se fazer funções recursivas. A primeira coisa a se providenciar é um critério de parada, o qual vai determinar quando a função deverá parar de chamar a si mesma. Este cuidado impede que a função se chame infinitas vezes.

Um exemplo de um problema passível de definição recursiva é a operação de multiplicação efetuada sobre números naturais. Podemos definir a multiplicação em termos da operação mais simples de adição.

UNIVASF

# Recursividade

No caso

A * B

pode ser definido como

A + A * (B - 1)

precisamos agora especificar um critério de parada. Qual seria?

A * 0 = 0

# Recursividade

De acordo com o que vimos até o momento, podemos definir um laço de repetição que implementaria o cálculo da operação de multiplicação entre valores naturais com base na operação de adição. Por exemplo:

```
...
int A, B, RES;
...
RES=0;
while (B<>0)
{
     RES = RES + A;
     B = B-1;
}
...
```

UNIVASF

# Recursividade

Com base no que vimos podemos, também, definir uma função recursiva que implemente a operação de multiplicação com base na operação de adição:

```
int multiplicar (int A, int B)
{
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
```

# Recursividade

Para uma melhor compreensão do que foi apresentado, devemos compreender o conceito de “registro de ativação”.

O registro de ativação é uma área de memória que guarda informações referentes ao estado atual de uma função ou do próprio programa:

- valor dos parâmetros (para funções);
- valor das variáveis locais (para funções);
- valor do contador de programa (Program Counter - PC);
- etc.

UNIVASF

# Recursividade

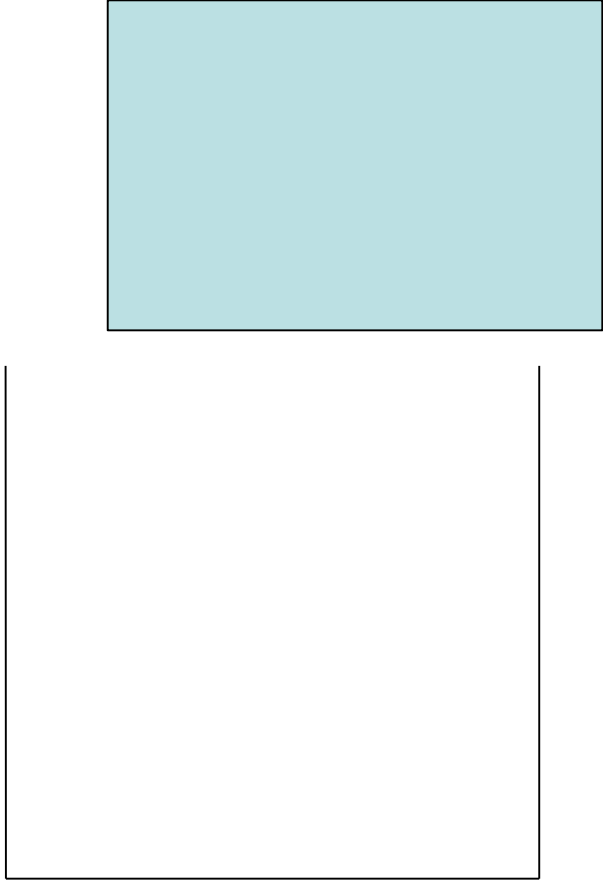
Sempre que uma função é chamada o registro de ativação de quem a invocou (da função principal, de uma outra função ou da própria função) é salvo e um novo registro de ativação é criado para a função invocada. Estes registros de ativação são empilhados em uma pilha de registros de ativação.

Este processo é conhecido como salvamento e troca de contexto e pode ser melhor compreendido se o aplicarmos sobre um programa que se utilize da função recursiva “multiplicar” definida anteriormente.

UNIVASF

```c
#include <stdio.h>
int multiplicar (int A, int B) {
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
int main(void) {
    int A, B, RES;
    do {
        printf ("\nMultiplicando (valor natural): ");
        scanf("%d", &A);
    }while (A<0);
    do {
        printf("\nMultiplicador (valor natural): ");
        scanf("%d", &B);
    }while (B<0);
    RES = multiplicar(A,B);
    printf("\n%d * %d = %d\n",A,B,RES);
}
```

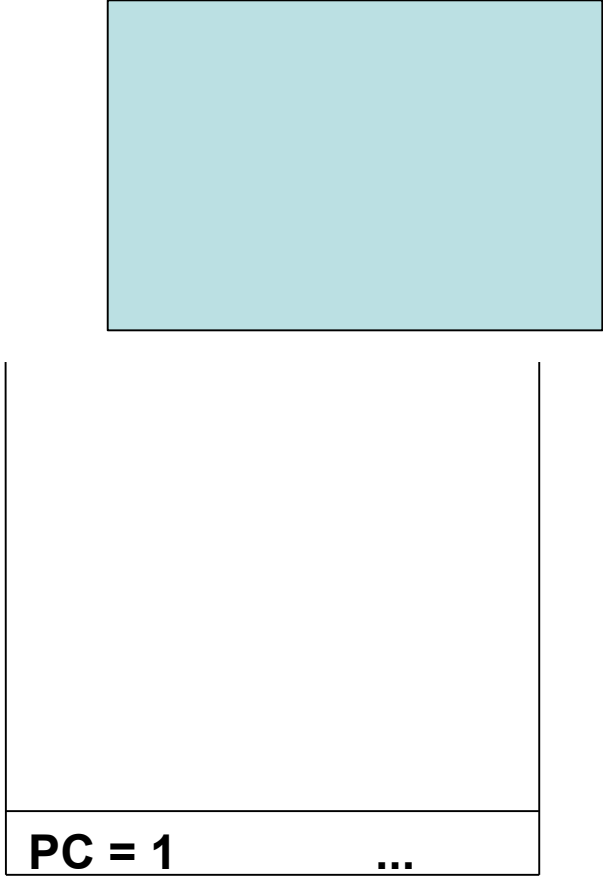
```c
#include <stdio.h>
int multiplicar (int A, int B) {
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
int main(void) {
    int A, B, RES;
    do {
        printf ("\nMultiplicando (valor natural): ");
        scanf("%d", &A);
    }while (A<0);
    do {
        printf("\nMultiplicador (valor natural): ");
        scanf("%d", &B);
    }while (B<0);
    RES = multiplicar(A,B);
    printf("\n%d * %d = %d\n",A,B,RES);
}
```



**Pilha de registros de ativação**

Para melhor contextualizar nossa explicação vamos presumir que o usuário fornecerá o valor 7 para “A” e o valor 3 para “B”.

```c
#include <stdio.h>
int multiplicar (int A, int B) {
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
int main(void) {
    int A, B, RES;
    do {
        printf ("\nMultiplicando (valor natural): ");
        scanf("%d", &A);
    }while (A<0);
    do {
        printf("\nMultiplicador (valor natural): ");
        scanf("%d", &B);
    }while (B<0);
    RES = multiplicar(A,B);
    printf("\n%d * %d = %d\n",A,B,RES);
}
```

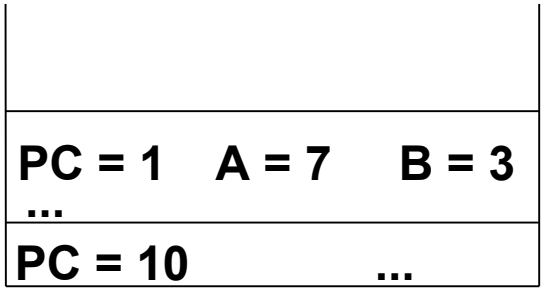



**Pilha de registros de ativação**

Inicialmente o registro de ativação da função main é colocado na pilha de registros de ativação.

```c
#include <stdio.h>
int multiplicar (int A, int B) {
    1 if (!B)
    2     return (0);
    3 else
    4     return (A + multiplicar (A, B-1));
}
int main(void) {
    1 int A, B, RES;
    2 do {
    3     printf ("\nMultiplicando (valor natural): ");
    4     scanf("%d", &A);
    5 }while (A<0);
    6 do {
    7     printf("\nMultiplicador (valor natural): ");
    8     scanf("%d", &B);
    9 }while (B<0);
   10 RES = multiplicar(A,B);
   11 printf("\n%d * %d = %d\n",A,B,RES);
}
```

**RES = multiplicar (7,3);**




**Pilha de registros de ativação**

Em nosso exemplo a primeira execução da função multiplicar ocorre na décima instrução da seção de comandos da main. Neste momento é salvo o registro de ativação do algoritmo e introduzido na pilha um novo registro de ativação referente à função chamada.

```c
#include <stdio.h>
int multiplicar (int A, int B) {
    if (!B)
        return (0);
    else
        return (A + multiplicar (A, B-1));
}
int main(void) {
    int A, B, RES;
    do {
        printf ("\nMultiplicando (valor natural): ");
        scanf("%d", &A);
    }while (A<0);
    do {
        printf("\nMultiplicador (valor natural): ");
        scanf("%d", &B);
    }while (B<0);
    RES = multiplicar(A,B);
    printf("\n%d * %d = %d\n",A,B,RES);
}
```

?

7 + multiplicar (7,2)

?

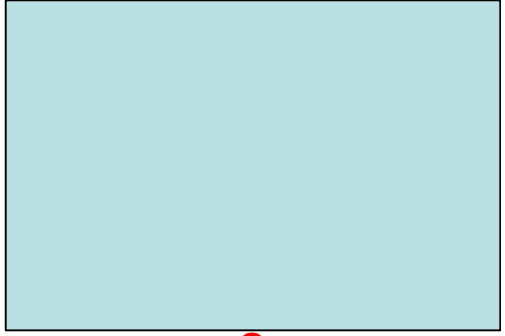
RES = multiplicar (7,3);

| |
|---|
| PC = 1 A = 7 B = 2 ... |
| PC = 4 A = 7 B = 3 ... |
| PC = 9 ... |

**Pilha de registros de ativação**

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo um registro de ativação (RA) do invocador e introduzido na pilha um novo RA.

```c
#include <stdio.h>
int multiplicar (int A, int B) {
   1 if (!B)
   2     return (0);
   3 else
   4     return (A + multiplicar (A, B-1));
}
int main(void) {
   1 int A, B, RES;
   2 do {
   3     printf ("\nMultiplicando (valor natural): ");
   4     scanf("%d", &A);
   5 }while (A<0);
   6 do {
   7     printf("\nMultiplicador (valor natural): ");
   8     scanf("%d", &B);
   9 }while (B<0);
  10 RES = multiplicar(A,B);
  11 printf("\n%d * %d = %d\n",A,B,RES);
}
```



?

7 + multiplicar (7,1)

?

7 + multiplicar (7,2)

?

RES = multiplicar (7,3);

| | | | |
|---|---|---|---|
| PC = 1 ... | A = 7 | B = 1 | |
| PC = 4 ... | A = 7 | B = 2 | |
| PC = 4 ... | A = 7 | B = 3 | |
| PC = 9 | | ... | |

Pilha de registros de ativação

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo o RA do invocador e introduzido na pilha um novo RA.

UNIVASF

# Recursividade

Devido ao valor contido no parâmetro B a quarta instrução da seção de comandos da função multiplicar é executada chamando novamente a função multiplicar. Neste momento é salvo o RA do invocador e introduzido na pilha um novo RA.

Devido ao valor contido no parâmetro B a segunda instrução da seção de comandos da função multiplicar é executada retornando o valor zero e finalizando as chamadas recursivas. Neste é desempilhado um RA e o contexto do RA da função invocadora é retomado.

0
?
7 + multiplicar (7,0)
?
7 + multiplicar (7,1)
?
7 + multiplicar (7,2)
?
RES = multiplicar (7,3);

| | | |
|---|---|---|
| PC = 1 ... | A = 7 | B = 0 |
| PC = 4 ... | A = 7 | B = 1 |
| PC = 4 ... | A = 7 | B = 2 |
| PC = 4 ... | A = 7 | B = 3 |
| PC = 9 | | ... |

Pilha de registros de ativação

UNIVASF

# Recursividade

Desta forma uma a uma as camadas às funções vão sendo finalizadas e seus registros de ativação desempilhados.

**RES = multiplicar (7,3);**

**RES = 21**

**?**

**7 + multiplicar (7,2)**

**7 + 14**

**?**

**7 + multiplicar (7,1)**

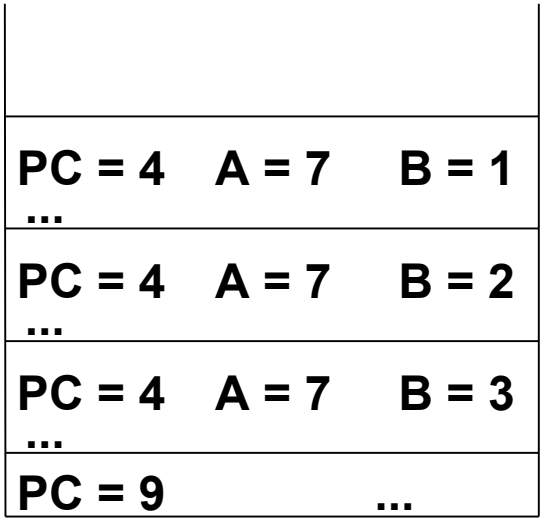
**7 + 7**

**?**

**7 + multiplicar (7,0)**

**7 + 0**

**0**




**Pilha de registros de ativação**

# Recursividade

Com base no que foi exposto, podemos visualizar algumas desvantagens da utilização de recursividade, como:

- O consumo de memória necessário para a troca de contexto.

- Redução do desempenho de execução devido ao tempo para gerenciamento de chamadas.

- Dificuldades na depuração de programas recursivos, especialmente se a recursão for muito profunda.

**Exercício:** Para uma melhor compreensão do conceito de recursividade faça agora uma função recursiva para calcular o fatorial de um número natural e construa um programa que se utilize de forma adequada da função em questão.

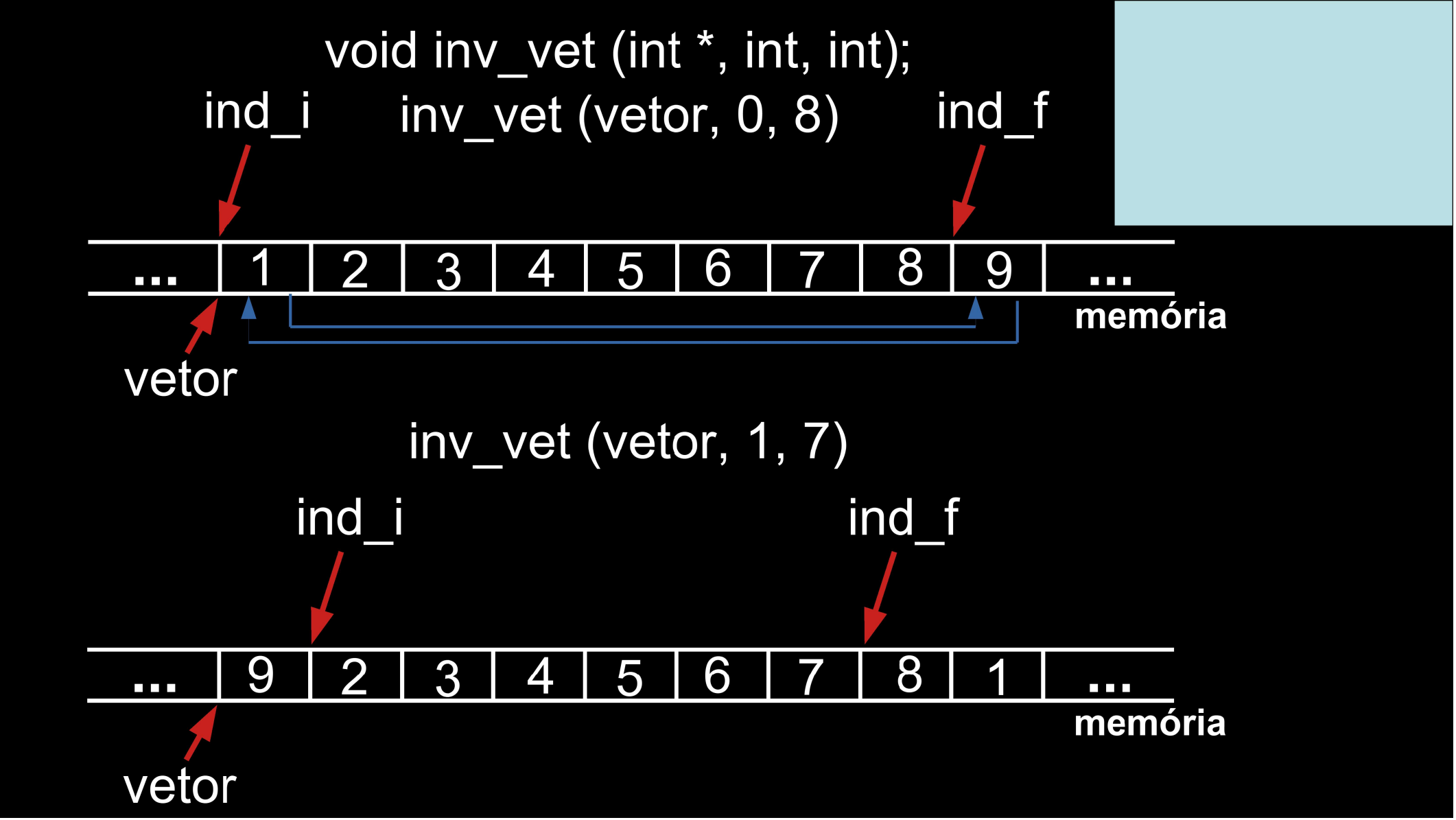
UNIVASF

```c
#include <stdio.h>
int fatorial (int num)
{
    if (!num)
        return (1);
    else
        return (num * fatorial(num-1));
}
int main()
{
    int n;
    printf ("Digite o numero que voce deseja saber o fatorial: ");
    scanf ("%d", &n);
    if (n>=0)
        printf ("\nO fatorial do numero %d eh %d",n,fatorial(n));
    else
        printf ("\nNao existe fatorial de numeros negativos!");
}
```

# Recursividade

**Exercício:**

Para uma melhor compreensão do conceito de recursividade construa, na linguagem de programação C, uma função recursiva que receba como parâmetro dados referentes a um vetor com elementos inteiros e inverta a ordem de seus elementos.


void inv_vet (int *, int, int);
ind_i
inv_vet (vetor, 0, 8)
ind_f
... 1 2 3 4 5 6 7 8 9 ...
memória
vetor
inv_vet (vetor, 1, 7)
ind_i
ind_f
... 9 2 3 4 5 6 7 8 1 ...
memória
vetor

```c
void inv_vet (int *vet, int ind_i, int ind_f)
{
    int aux;
    if (ind_i<ind_f)
    {
        aux = vet[ind_i];
        vet[ind_i] = vet[ind_f];
        vet[ind_f] = aux;
        inv_vet (vet, ind_i+1, ind_f-1);
    }
}
```

# Recursividade

Um outro exemplo muito utilizado de problema que possui uma definição recursiva é a geração da série de Fibonacci:

{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …}

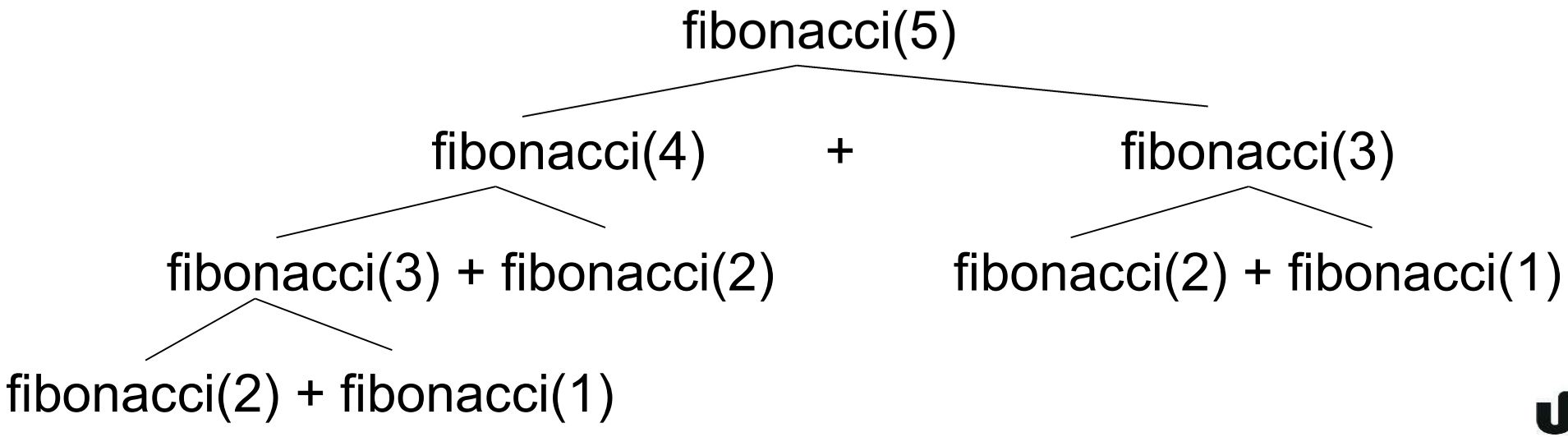
Uma função recursiva que recebe a posição do elemento na série e retorna seu valor é:

```
unsigned int fibonacci(unsigned int i)
{
    if (i==1)
       return 0;
    if (i==2)
       return 1;
    return (fibonacci(i-1) + fibonacci(i-2))
}
```

# Recursividade

Fora os problemas mencionados anteriormente, relacionados à troca de contexto na recursão, qual seria outro problema proveniente da recursão evidenciado na função recursiva apresentada para o cálculo do valor de um elemento da série de Fibonacci com base na sua posição?

O cálculo do mesmo valor n vezes.



# Recursividade

Como vimos, mesmo problemas que possuem uma definição recursiva podem ser solucionados de forma imperativa. Um exemplo disso é o cálculo imperativo do valor de um elemento da série de Fibonacci com base na sua posição, apresentado abaixo:

```
unsigned int fibonacci(unsigned int i) {
    if (i==1)
        return 0;
    if (i==2)
        return 1;
    else {
        unsigned int a, b;
        for(a=0, b=1; i-2; b+=a,a=b-a,i--);
        return b;
    }
}
```

UNIVASF

# Recursividade

Assim como a série de Fibonacci existem outras sequências definidas por recorrência, ou seja, onde um valor da sequência é definido em termos de um ou mais valores anteriores, o que é denominado de relação de recorrência.

**Exercício:**

Estabeleça a relação de recorrência presente na sequência abaixo.

S = { 1, 2, 6, 24, ... }

# Recursividade

A sequência S é definida por recorrência por

1. S(1) = 1
2. S(n) = n * S(n-1) para n>=2

# Recursividade

**Exercício:**

Com base na relação de recorrência estabelecida no exercício anterior, considerando o princípio da modularização, construa um programa que receba uma lista de inteiros positivos, representando posições de elementos na sequência e retorne na saída padrão os respectivos valores da sequência. A lista de posições será finalizada pelo valor zero. Não é necessário validar as entradas.

| Exemplo de entrada: | Exemplo de saída: |
|---|---|
| 4 | 24 |
| 2 | 2 |
| 0 | |

UNIVASF

```c
#include <stdio.h>
unsigned int func (unsigned int);
int main() {
    unsigned int num;
    do {
        scanf("%u", &num);
        if (num)
            printf("%u\n",func(num));
    }while(num);
    return 0;
}
unsigned int func (unsigned int n) {
    if (n==1)
        return 1;
    return (n*func(n-1));
}
```