Universidade Federal do Espírito Santo

Departamento de Computação e Eletrônica - CEUNES
Estrutura de Dados I
Prof. Oberlan Romão

Recursão

Recursão é um método de resolução de problemas que envolve quebrar um problema em subproblemas menores e menores até chegar a um problema pequeno o suficiente para que ele possa ser resolvido trivialmente. Normalmente recursão envolve uma função que chama a si mesma (recursão direta). Embora possa não parecer muito, a recursão nos permite escrever soluções elegantes para problemas que, de outra forma, podem ser muito difíceis de programar. Toda função recursiva deve obedecer a três leis importantes:

1. Deve chamar a si mesmo, recursivamente;
2. Deve ter um caso básico (chamado de **caso base**);
3. Deve mudar o seu estado e se aproximar do caso básico (chamado de **passo recursivo**).

Exercícios

1. (`ex1.c`) Teste o seguinte programa, fornecendo o valor 6 e outros valores como entrada. As mensagens escritas mostram em que ordem cada execução da função fatorial é iniciada e terminada.

```c
#include <stdio.h>

int fatorial(int n){
    printf("Iniciando a funcao fatorial(%d)\n", n);
    if (n <= 1){
        printf("Finalizando a funcao fatorial(%d) com valor 1\n", n);
        return 1;
    }
    int fat = fatorial(n - 1);
    printf("Finalizando a funcao fatorial(%d) com valor %d\n", n, n*fat);
    return n*fat;
}

int main() {
    int n, fat;
    scanf("%d", &n);
    fat = fatorial(n);
    printf("%d! = %d\n", n, fat);

    return 0;
}
```

Você pode utilizar o site C Tutor para facilitar a visualização da execução desse código, nesse caso, troque a leitura do valor de `n` (`scanf("%d", &n);`) por `n = 6`.

Obs.: A função poderia, é claro, ser feita sem todos os comandos de escrita, e o retorno no final da função poderia ser `return n*fatorial(n-1)`, dispensando o uso da variável `fat`.

2. (`ex2.c`) Analise as funções recursivas abaixo e descubra qual o problema de cada uma. Após corrigir, execute o código para ter certeza que tudo funcionou. Caso não esteja conseguindo descobrir onde está o problema, utilize o site C Tutor para te auxiliar.

```c
#include <stdio.h>

int f1(int n){
    /* Retorna a qtd de números naturais pares menores ou iguais a n */
    if (n % 2 == 0)
        return 1 + f1(n - 2); //Por que n - 2?
    else
        return f1(n - 1); //Por que n - 1?
}

int f2(int n){
    /* Retorna a soma dos números naturais menores ou iguais a n */
    return (n == 1) ? 1 : (n + f2(n + 1)); // Operador ternário em C
}

int main() {
    printf("f1(%d) = %d\n", 10, f1(10));
    printf("f2(%d) = %d\n", 10, f2(10));
    return 0;
}
```

Note a utilização do operador ternário em C (ele é muito similar ao Operador Ternário de Python). Para saber mais: `https://www.geeksforgeeks.org/conditional-or-ternary-operator-in-c-c/`

3. (`ex3.c`) A *sequência de Fibonacci* foi descrita pelo matemático italiano Leonardo Pisano, também conhecido por Leonardo Fibonacci, quando estudava o crescimento de uma população de coelhos. Mais tarde verificou-se que a sequência aparece em muitas outros campos do conhecimento. A sequência é definida recursivamente como

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

Os primeiros valores da séria são: $0, 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ \ldots$.

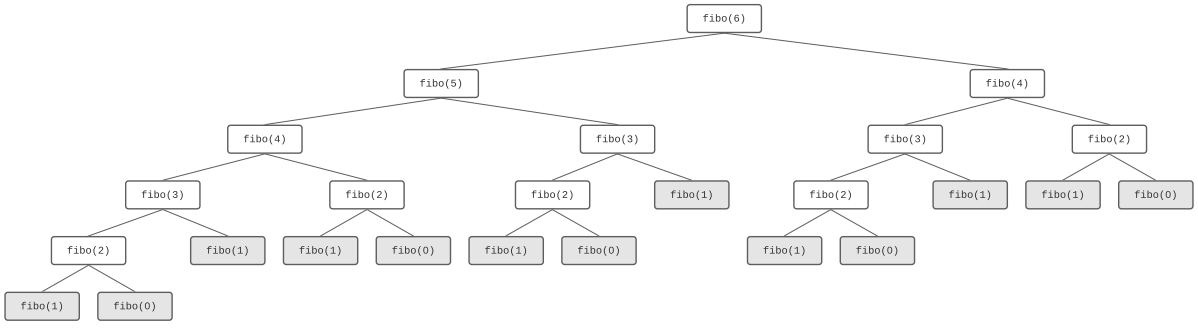
Uma função recursiva para calcular $Fibo(n)$ é simples:

```c
int fibo(int n){
    if (n < 2)
        return n;
    else
        return fibo(n - 1) + fibo(n - 2)

}
```

a) Execute o código abaixo fornecendo valores pequenos para n, como 5 ou 6. Note quantas vezes a função `fibo` é chamada.

```c
#include <stdio.h>

int fibo(int n){
    printf("Iniciando a funcao fibo(%d)\n", n);
    if (n < 2){
        printf("Finalizando a funcao fibo(%d) com valor %d\n", n, n);
        return n;
    }

    int f = fibo(n - 1) + fibo(n - 2);
    printf("Finalizando a funcao fibo(%d) com valor %d\n", n, f);
    return f;

}

int main() {
    int num, fibonacci;
    scanf("%d", &num);
    fibonacci = fibo(num);
    printf("fibo(%d) = %d\n", num, fibonacci);
    return 0;
}
```

A figura abaixo ilustra as chamadas recursivas da função para `n = 6`. Tente seguir os passos do programa por ela. Na chamada da função, o programa "desce" um nível. E no retorno da função, ele "sobe" de volta.



Na Ciência da Computação, esse formato é chamado árvore. Nesse caso, árvore de recursão, sendo a primeira chamada (ou seja, `fibo(6)`) a raiz da árvore, e as chamadas não recursivas (`fibo(1)` e `fibo(0)`) as folhas da árvore.

b) Observe que algumas chamadas recursivas (por exemplo, `fibo(3)`) são calculadas várias vezes. Por isso, para esse cálculo é claramente mais vantajoso fazer uma função iterativa, em vez da função recursiva, a não ser que os valores já calculados sejam de alguma forma armazenados para não serem calculados novamente. Faça uma função iterativa para calcular os termos da série. Note que ela não é tão simples quanto a recursiva, porém é mais eficiente. Meça o tempo de execução[1] da versão recursiva e da versão iterativa para diferentes valores de n.

4. (`ex4.c`) Analise as funções recursivas abaixo.

```c
int M(int a, int b) {
    if (b == 0)
        return 0;
    else if (b == 1)
        return a;
    else
        return a + M(a, b - 1);
}
int DI(int a, int b) {
    if (a < b)
        return 0;
    else
        return 1 + DI(a - b, b);
}
double RD(double a, double b) {
    if (a < b)
        return a;
    else
        return RD(a - b, b);
}
```

a) Qual o objetivo de cada função?

b) Qual será a saída quando for executado, dentro da função `main`, os comandos abaixo?

```c
printf("%d\n", M(2, 3));
printf("%d\n", M(4, 2));
printf("%d\n", DI(2, 3));
printf("%d\n", DI(3, 2));
printf("%d\n", DI(4, 2));
printf("%d\n", DI(10, 6));
printf("%lf\n", RD(2, 3));
printf("%lf\n", RD(3, 2));
printf("%lf\n", RD(25.5, 10.5));
```

[1] Para medir o tempo, você pode usar a função clock() da biblioteca `time.h` ou, se estiver no Linux, você pode usar o comando 'time ./executavel'

5. (ex5.c) Analise o código abaixo.

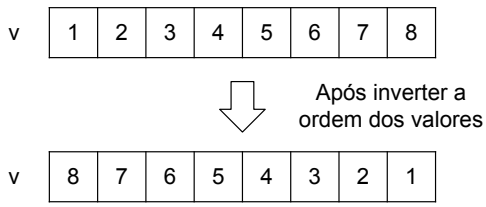
```c
#include <stdio.h>

void imprime(char s[]){
    if (s[0] == '\0')
        return;
    printf("%c", s[0]);
    imprime(&s[1]);
}

int main() {
    char msg[257]; //Ultima posição para o '\0'
    //Lê 256 caracteres ou até encontrar um fim de linha (ou seja, um \n)
    fgets(msg, 256, stdin);
    imprime(msg);
    return 0;
}
```

   a) O que você espera da execução do seguinte programa? Execute e confira o resultado.

   b) Coloque o comando `printf("%c", s[0]);` da função após a chamada recursiva, ou seja, coloque-o como último comando da função. O que você espera do resultado? Execute e veja a diferença.

6. (ex6.c) Escreva uma função recursiva para calcular a n-ésima potência de um número, ou seja, x^n. O seu algoritmo deve ter complexidade $O(\log n)$. Como você deve imaginar, você não pode usar a função `pow` ou outra similar. Dica: procure por "exponenciação rápida".

7. (ex7.c) Muitas vezes precisamos inverter a ordem dos elementos de um vetor. Veja o exemplo abaixo.



   Como você resolveria este problema? Qual seria a complexidade de tempo e espaço da sua solução? Só continue a leitura após responder as perguntas anteriores :).

   Uma forma ingênua (não eficiente) de resolver esse problema é usando outro vetor auxiliar:

```c
void inverte(int v[], int n) {
    int *tmp = malloc(n*sizeof(int));
    //O vetor 'tmp' recebe os valores na ordem inversa
    for(int i = 0, j = n - 1; i < n; i++, j--){
        tmp[i] = v[j]; // Note a diferença dos índices
    }
```

```
7       //Agora copiamos os valores de volta ao vetor 'v'
8       for(int i = 0; i < n; i++){
9           v[i] = tmp[i];
10      }
11      free(tmp);
12  }
```

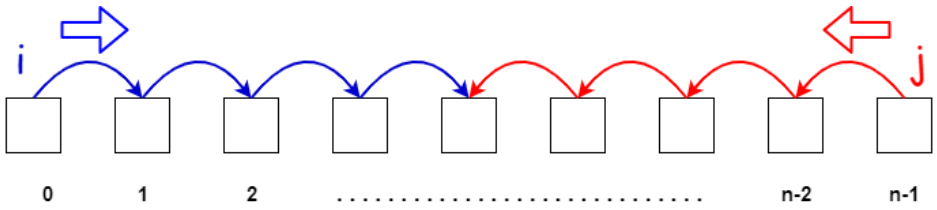
Você pensou em algo parecido com o código anterior?

Na função `inverte`, note que o `for` da linha 4 declara e inicializa duas variáveis, `i` e `j` (observe a utilização do operador ',' para essa finalidade). A variável `i` representa o índice do vetor `tmp` e a `j` representa o índice de `v`. Observe também que `i` vai de `0` a `n - 1` (sendo incrementada em uma unidade em cada iteração) enquanto `j` vai de `n - 1` a `0` (sendo decrementada em uma unidade em cada iteração). Após a execução desse `for`, teremos em `tmp` os valores de `v` na ordem inversa. Agora basta copiarmos os valores de volta para o vetor `v`, o que é feito pelo `for` da linha 8.

Essa estratégia possui um grande problema: ela precisa de um vetor auxiliar (`tmp`) para inverter os elementos. Imagine um vetor que ocupe alguns gigabytes de memória, com essa abordagem esse espaço precisa ser duplicado toda vez que a função `inverte` for chamada. Será que dá para implementar esse algoritmo sem precisar do vetor auxiliar? Como você deve imaginar, a resposta é SIM. Você consegue pensar em como fazer isso? Note como fica difícil pensar em algo diferente após ver o código da função `inverte`.

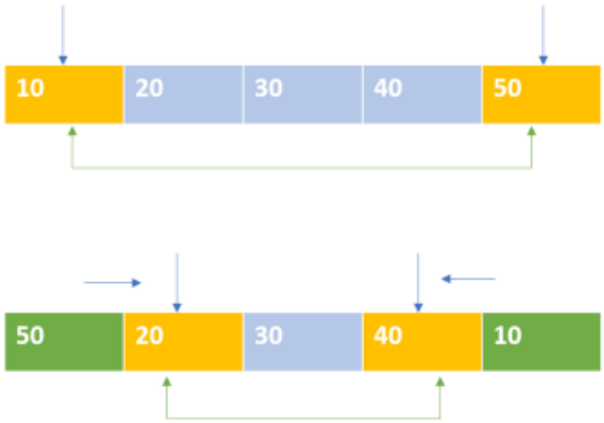
Vamos ver outra estratégia (mais eficiente) que nos ajuda nesse e em vários outros problemas.

Two Pointers Inverter os elementos de um vetor é um problema típico que pode ser resolvido por uma técnica chamada *Two Pointers*. Nela usamos dois "apontadores" que caminham pelo vetor. Normalmente, esses apontadores são "colocados" nas extremidades opostas do vetor e caminham um em direção ao outro, como mostra a figura abaixo.



Você consegue pensar em como usar esse técnica para resolver o problema de inverter os elementos de um vetor sem usar um vetor auxiliar? (Pense um pouco antes de continuar a leitura).

A ideia é simples: coloque cada apontador (digamos `i` e `j`) em uma extremidade do vetor, ou seja, `i = 0` e `j = n - 1`, e, a cada iteração, troque os elementos que estão nas posições `i` e `j` (ou seja, `v[i] <-> v[j]`). Após a troca, incremente o apontador `i` e decremente o apontador `j`. Repita esse processo enquanto `i < j`. A figura abaixo ilustra parte desse processo.



O pseudo-código dessa estratégia é mostrado abaixo (note a simplicidade):

```
i = 0
j = n - 1
enquanto (i < n)
    troque v[i] com v[j]
    i = i + 1
    j = j - 1
```

Tarefa: implemente a estratégia *Two Pointers* para resolver o problema de inverter os elementos de um vetor.

Para saber mais: `https://www.geeksforgeeks.org/two-pointers-technique/`