Métodos e Técnicas de Programação
.:: Recursão aplicada ::.

Prof. Igor Peretta

2018-2

Contents

1 Recursão **1**
- 1.1 Para saber mais . . . 2
- 1.2 Exemplo: Fatorial . . . 2
- 1.3 Exemplos . . . 3
  - 1.3.1 Sequência de Fibonacci . . . 3
  - 1.3.2 Convertendo strings e arrays . . . 4
  - 1.3.3 Revertendo strings . . . 6

2 Mais detalhes sobre recursão **6**
- 2.1 Em busca da compreensão . . . 6

1 Recursão

Recursividade é um termo usado de maneira mais geral para descrever o processo de repetição de um objeto de maneira similar ao laço que já foi mostrado. Um bom exemplo na prática são as imagens repetidas que aparecem quando dois espelhos são apontados um para o outro.

A recursão na programação é bem exemplificada quando uma função é definida em termos de si mesma. Um exemplo da aplicação da recursão são os parsers (analisadores gramaticais) para linguagens de programação. Uma grande vantagem da recursão é que um conjunto infinito de sentenças possíveis, designs ou outros dados podem ser definidos, analisados ou produzidos por um programa de computador finito.

1.1 Para saber mais

Vídeo do canal Computerphile YouTube: Programming Loops vs Recursion (áudio e legendas em inglês)

1.2 Exemplo: Fatorial

Na matemática, o fatorial de um número natural n, representado por $n!$, é o produto de todos os inteiros positivos menores ou iguais a n. Vamos implementá-lo em C.

No escopo de `main()`:

```
int i, fat = 1, n = 6;
for(i = 2; i <= n; i++) fat *= i;
printf("Fatorial %d = %d", n, fat);
```

```
Fatorial 6 = 720
```

Como função:

```
int fatorial(int n) {
    int i, fat = 1;
    for(i = 2; i <= n; i++)
fat *= i;
    return fat;
}
int main() {
    int n = 6;
    printf("Fatorial %d = %d", n, fatorial(n));
    return 0;
}
```

```
Fatorial 6 = 720
```

Como recursão:

```
int fatorial(int n) {
    if(n == 1) return 1;
    else return n*fatorial(n-1);
}
int main() {
    int n = 13;
```

```
    printf("Fatorial %d = %d", n, fatorial(n));
    return 0;
}
```

```
Fatorial 13 = 1932053504
```

1.3 Exemplos

1.3.1 Sequência de Fibonacci

A Sequência de Fibonacci consiste em uma sucessão de números, tais que, definindo os dois primeiros números da sequência como 0 e 1, os números seguintes serão obtidos por meio da soma dos seus dois antecessores. Portanto, os números são p/ n = 0,1,2...: 0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181...

Dessa sequência, ao se dividir qualquer número pelo anterior, extrai-se a razão que é uma constante transcendental conhecido como número de ouro.

A sequência de Fibonacci é dada pela fórmula:

$$Fib(n) = \begin{cases} 1, & n = 0 \\ 1, & n = 1 \\ Fib(n-1) + Fib(n-2), & \text{caso contrario} \end{cases}$$

```
int fibonacci(int n) {
    if((n==0) || (n == 1)) return 1;
    else return fibonacci(n-1) + fibonacci(n-2);
}
int main() { int i;
    for(i = 0; i < 10; i++)
printf("%d, ", fibonacci(i));
    printf("\n");
    return 0;
}
```

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

```
int fibonacci(int n) {
    int fib;
    switch(n) {
case 0:
case 1: fib = 1; break;
default: fib = fibonacci(n-1) + fibonacci(n-2);
    }
```

```
    return fib;
}
int main() { int i;
    for(i = 0; i < 10; i++)
printf("%d, ", fibonacci(i));
    printf("\n");
    return 0;
}
```

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

1.3.2 Convertendo strings e arrays

Trabalhando com *strings*, podemos nos aproveitar do fato que terminam com '\0'.

```
#include <stdio.h>
#include <ctype.h>
void strToUpper(char * str) {
    if(*str) {
       *str = toupper(*str);
       strToUpper(str+1);
    }
}
int main() {
     char str[] = "Esta e a frase? Quero maiusculas??";
    strToUpper(str);
    printf("%s\n", str);
    return 0;
}
```

```
ESTA E A FRASE? QUERO MAIUSCULAS??
```

```
#include <stdio.h>
void strToUpper(char * str) {
    if(*str) { // (*str != '\0')
       if(*str >= 'a' && *str <='z')
  *str -= 'a' - 'A';
       strToUpper(str+1);
    }
}
```

```
int main() {
     char str[] = "Esta frase eu quero em maiusculas??";
    strToUpper(str);
    printf("%s\n", str);
    return 0;
}
```

```
ESTA FRASE EU QUERO EM MAIUSCULAS??
```

Já com *arrays*, precisamos controlar o tamanho.

```
#include <stdio.h>
#include <math.h>
void map(double * vec, int tam, double (*fun)(double)) {
    if(tam > 0) {

 *vec = fun(*vec);
 map(vec+1,tam-1,fun);
    }
}
double reduce(double * vec, int tam) {
    if(tam > 0)
return *vec + reduce(vec+1, tam-1);
    else
return 0.0;
}
double f2x1 (double x) {
    return 2.0*x + 1.0;
}
int main() {
    double vec[] = {0.0,1.0,2.0,3.0,4.0,5.0};
    int i, tam = sizeof(vec)/sizeof(double);
    map(vec,tam,f2x1);
    for(i = 0; i < tam; i++)
printf("%lg%s", vec[i], (i==tam-1)? "\n" : "; ");
    printf("Soma: %lg\n",reduce(vec,tam));
    return 0;
}
```

```
1; 3; 5; 7; 9; 11
Soma: 36
```

1.3.3 Revertendo strings

```
#include <stdio.h>
void reverter(char * str) {
    if(*str) {
reverter(str+1);
printf("%c",*str);
    }
}
int main() {
    char str[] = "Socorram-me, subi no onibus em Marrocos";
    reverter(str);
    return 0;
}
```

```
socorraM me subino on ibus ,em-marrocoS
```

2 Mais detalhes sobre recursão

Recursividade é uma forma de repetição que pode ser aplicada em muitos casos, não só da Matemática ou da Computação. Na Computação, diz-se que uma função é recursiva quando ela chama a si mesma. Ela geralmente possui pelo menos um caso base, que interrompe a recursividade, e um caso recursivo, que define a condição para que haja repetição. Ter um caso base bem definido é importante para impedir a repetição infinita da função (fonte: `https://pt.wikibooks.org/wiki/Haskell/Recursividade`).

2.1 Em busca da compreensão

Considere o seguinte vetor ordenado que contém os ímpares de 1 a 2097151 (ambos inclusos):

```
int i, vetor[1048576]; // 1 Mega inteiros ou 4 Mbytes de memoria
for(i = 0; i < 1048576; i++) {
    vetor[i] = 2*i + 1; // informação
    if(i < 100 || i > 1048576 - 100)
printf("%u ", vetor[i]);
if(i == 100) printf(" ... ");
if(i == 1048576 - 1) printf("\n");
}
```

```
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 6
```

Consideremos o seguinte problema: é necessário encontrar nesse vetor ordenado o índice de um valor específico. Vamos começar a solução com uma possível implementação *naïve*:

```
#include <stdio.h>
#define K 1048576
int retorna_indice(int * vet, int Tam, int chave) {
    int id, *fim = vet + Tam;
    for(id = 0; vet[id] != chave && vet+id < fim; id++);
    return (id == Tam)? -1 : id;
}
int main() {
    int elemento = 2097121;
    int i, id, vetor[K];
    for(i = 0; i < K; i++)
vetor[i] = 2*i + 1;
    id = retorna_indice(vetor, K, elemento);
    printf("\nIndice de %d: %d\n", elemento, id);
    return 0;
}
```

```
Indice de 2097121: 1048560
```

O algoritmo apresentado possui complexidade n, o que significa que quanto maior o n, mais demorada é a busca. Por exemplo, para $n = 1048576$ elementos, serão 1048576 iterações. Existe alguma coisa mais efetiva?

Vamos analisar o algoritmo de busca binária (fonte: `https://pt.wikipedia.org/wiki/Pesquisa_binária`), que utiliza a estratégia de "dividir e conquistar". Essa é uma possível implementação:

```
#include <stdio.h>
#define K 1048576
int retorna_indice(int * vet, int Tam, int chave) {
    int inf = 0;     // limite inferior
    int sup = Tam-1; // limite superior
    int meio;
    while (inf <= sup)
    {
printf("* ");
```

```
meio = (inf + sup)/2;
if (chave == vet[meio])
    return meio;
if (chave < vet[meio])
    sup = meio-1;
else
    inf = meio+1;
     }
     return -1;   // não encontrado
}
int main() {
    int elemento = 2097151/2;
    int i, id, vetor[K];
    for(i = 0; i < K; i++)
vetor[i] = 2*i + 1;
    id = retorna_indice(vetor, K, elemento);
    printf("Indice de %d: %d\n", elemento, id);
    return 0;
}

* Indice de 1048575: 524287
```

Esse algoritmo de busca possui complexidade $log_2 n$, o que significa que é mais rápido do que a implementação *naïve*. No exemplo, para $n = 1048576$ elementos, serão necessárias $\log_2 1048576 = 20$ iterações.

Comparando as duas complexidades (n e $log_2 n$) para uma quantidade de dados máxima de $n = 1048576$, temos um *speedup* máximo relativo de $\frac{1048576}{20} \approx 52500$ vezes mais rápido.

Teste aqui o algoritmo:

índice	0	1	2	3	4	5	6	7	8	9	10
elemento	5	12	25	32	46	57	66	71	80	93	104

Como funciona? Veja o pseudo-código do algoritmo de busca binária:

```
BUSCA-BINÁRIA (V[], início, fim, e)
    i recebe o índice do meio entre início e fim
    se (v[i] = e) entao
        devolva o índice i   # elemento e encontrado
    fimse
```

```
    se (inicio = fim) entao
        não encontrou o elemento procurado
    senão
       se (V[i] vem antes de e) então
          faça a BUSCA-BINÁRIA(V, i+1, fim, e)
       senão
          faça a BUSCA-BINÁRIA(V, inicio, i-1, e)
       fimse
    fimse
```

Note que se trata de uma definição recursiva. Sua implementação com recursão fica assim:

```
#include <stdio.h>
#define K 1048576
int retorna_indice(int * vet, int ini, int fim, int chave) {
    int i = (ini+fim)/2;
    if(vet[i] == chave) return i;
    if(ini == fim) return -1;
    else (vet[i] < chave)? retorna_indice(vet, i+1, fim, chave) :
retorna_indice(vet, ini, i-1, chave);
}
int main() {
    int elemento = 2097121;
    int i, id, vetor[K];
    for(i = 0; i < K; i++)
vetor[i] = 2*i + 1;
    id = retorna_indice(vetor, 0, K-1, elemento);
    printf("\nIndice de %d: %d\n", elemento, id);
    return 0;
}

Indice de 2097121: 1048560
```

Variante usando apenas referências:

```
#include <stdio.h>
#define K 1048576
int * retorna_ref(int * pini, int * pfim, int * pchave) {
    printf("* ");
    int * veti = pini + ((pfim - pini)/2);
```

```
    if(*veti == *pchave) return veti;
    if(pini == pfim) return NULL;
    else (*veti < *pchave)? retorna_ref(veti+1, pfim, pchave) :
retorna_ref(pini, veti-1, pchave);
}
int main() {
    int elemento = 2097121;
    int i, vetor[K];
    int * pid;
    for(i = 0; i < K; i++)
vetor[i] = 2*i + 1;
    pid = retorna_ref(vetor, vetor+K, &elemento);
    printf("\nIndice de %d: %ld\n", elemento, pid - vetor);
    return 0;
}

* * * * * * * * * * * * * * * * *
Indice de 2097121: 1048560
```

Note que a subtração entre ponteiros retorna a quantidade de elementos do tipo especificado que podem ser armazenados entre dois ponteiros (cada elemento ocupando `sizeof()` do tipo de dado respectivo em bytes), ou seja, o *offset* em elementos entre os dois endereços.