

Procedimentos recursivos

Motivação

Recursão é uma técnica de resolução de problemas muito poderosa. A maioria das linguagens de programação modernas suporta diretamente programação recursiva, como é o caso de C.

Em geral, a técnica recursiva consiste no seguinte. Suponha que o problema a ser resolvido, P, pode ser medido em termos de um parâmetro n . Por exemplo, o problema pode ser calcular a potência 2^n , para $n \geq 0$. Se o problema for ordenar um vetor de números, podemos tomar n como sendo a dimensão do vetor, i.e. como a quantidade de números presentes no vetor, e assim por diante.

Caso base

Então, dada uma instância do problema P, testamos o parâmetro n . Quando n é o menor possível, resolvemos essa instância diretamente. É o *caso base*. Por exemplo, no caso do cálculo da potência base 2, quando n é zero podemos anunciar o resultado diretamente, i.e. $2^0 = 1$. No caso dos vetores, quando $n=1$ (por exemplo) também não precisamos fazer nada, pois um vetor com 1 elemento está automaticamente ordenado.

Caso Indutivo

Resta o *caso indutivo*, i.e., quando n não é o menor possível. Nesse caso, *assumimos* que sabemos resolver *qualquer instância menor* do problema, i.e, instâncias onde o parâmetro é menor que n . Em seguida, consideramos uma instância P1 do problema dado, *onde o parâmetro vale $n-1$* . Por exemplo, para calcular a potência 2^n , a instância menor seria calcular a potência $2^{(n-1)}$. Em seguida, *aplicamos o mesmo método, recursivamente*, para resolver P1. Como assumimos que o método funciona para toda instância de tamanho menor que n , deverá funcionar também para P1. Portanto, a rotina deve retornar com a solução S1 de P1. Então, transformamos esta solução para obter uma solução S para o problema original, P. Por exemplo, no caso do cálculo da potência base 2, a solução S1 será dada pelo valor $z=2^{(n-1)}$, com a rotina retornando z . Agora, bastaria calcular $2*z$ para obtermos o resultado desejado para o problema original que era calcular 2^n .

No caso da ordenação de vetores, poderíamos, por exemplo, considerar todos os elementos do vetor, exceto o último. Este novo vetor teria dimensão $n-1$, menor que n . *Recursivamente*, i.e., aplicando o mesmo método, ordenamos esse novo vetor, obtendo assim uma solução para a instância menor do problema original. Agora, precisamos usar esta solução intermediária para obter uma solução para o problema original, com n elementos no vetor. Para isso, basta inserir o último elemento na posição correta entre os elementos dessa solução parcial. Obteríamos, ao final, um vetor que conteria todos os elementos do vetor original e que estaria ordenado.

É claro que o método recursivo, quando aplicado ao problema menor, P1, também vai separá-lo em um problema ainda menor, P2, sobre o qual reaplicará os mesmos passos, recursivamente. Esse processo prossegue até que atinjamos um problema de ordem suficientemente pequena para estarmos num caso base. Aí o problema é resolvido diretamente. Em seguida, os passos recursivos começam a retornar, construindo, paulatinamente, a solução desejada para o problema original.

Exemplos

Exemplo 1:

Queremos calcular a função fatorial, dada por

$$\text{fat}(n) = 1 \times 2 \times 3 \times \dots \times (n-1) \times n$$

onde n é um inteiro positivo. Ou seja,

$$\text{fat}(3) = 1 \times 2 \times 3 = 6,$$

$$\text{fat}(5) = 1 \times 2 \times 3 \times 4 \times 5 = 120,$$

e assim por diante. Mais ainda, queremos um *algoritmo recursivo* para calcular fat.

Qual é o *caso base*?

Podemos tomar $n=1$ como o caso base. Diretamente, temos facilmente que $\text{fat}(1) = 1$.

Como proceder no *passo indutivo*?

Assumimos que já sabemos calcular, *recursivamente*, a função fat até o valor $(n-1)$. Baseado nesses valores, como fazer para calcular $\text{fat}(n)$? Da definição da função fat, é fácil ver que

$$\text{fat}(n) = \text{fat}(n-1) \times n.$$

Essa é a relação que procurávamos, pois podemos expressar $\text{fat}(n)$ em função dos valores anteriores $\text{fat}(n-1)$, $\text{fat}(n-2)$, ..., e do parâmetro n .

Agora resta programar esse algoritmo em C. Considere o programa abaixo:

```
// Programa recursivo para calcular o fatorial
#include <stdio.h>
#include <stdlib.h>
// declaração da função
int fatorial(int n);
// programa para teste
void main(void) {
    int n;
    do {
        printf("Entre com o parâmetro (valor < 1 termina execucao): ");
        scanf("%d", &n);
        if (n<=0) { break; }
        printf("Fatorial de %ld = %ld\n", n, fatorial(n));
    } while (1); // termina lendo valor < 1
    return;
}
// função fatorial
int fatorial(int n) {
    if (n==1) {
        return 1;
    }
    return (n*fatorial(n-1));
}
```

O programa principal é um laço simples que pede novos parâmetros e aciona a função `fatorial` para calcular o fatorial, até que o usuário encerre a execução entrando um parâmetro menor do que 1.

A função `fatorial` é uma implementação direta do algoritmo recursivo. Repare no teste inicial para determinar se estamos no caso base. Se positivo, a função retorna 1 imediatamente. Se negativo, a função *chama a si mesma*, com o valor do parâmetro subtraído de uma unidade, para calcular `fat(n-1)`. Em seguida, multiplica esse valor por `n` e retorna o resultado, conforme o passo indutivo exige.

Exemplo 2:

Queremos um *algoritmo recursivo* para calcular potências na base 2:

$$\text{potencia2}(n) = 2^n,$$

para valores de `n` maiores ou iguais a zero.

A idéia é muito semelhante àquela apresentada no algoritmo anterior e está baseada no fato de que

$$\text{potencia2}(0) = 1,$$

para o *caso base*, e

$$\text{potencia2}(n) = 2 \times \text{potencia2}(n-1),$$

para o *passo indutivo*, quando `n > 1`.

No programa a seguir, implementamos um código ligeiramente diferente.

```
// Programa recursivo para calcular potencias (nao negativas) base 2
#include <stdio.h>
#include <stdlib.h>
// declaração da função
int potencia2(int n);
// programa para teste
void main(void) {
    int n;
    do {
        printf("Entre com o parametro (valor < 0 termina execucao): ");
        scanf("%d", &n);
        if (n<0) { break; }
        printf("Potencia: 2^%1d = %1d\n", n, potencia2(n));
    } while (1); // termina lendo valor < 0
    return;
}
// função potencia
int potencia2(int n) {
    int p;
    if (n==0) {
        p = 1;
    }
    else (p = 2*potencia2(n-1));
    return p;
}
```

De novo, a implementação segue de perto o algoritmo recursivo que encontramos. A diferença agora é que a função acumula o resultado em uma *variável local*, `p`, retornando `p` no final da execução.

Suponha que o usuário passe o valor 2 para `n` e que o programa principal chame `potencia2(n)`. Vamos ver o que ocorre com as várias instâncias da variável `p`, à medida que as chamadas recursivas se sucedem.

Na *primeira chamada* da função, uma posição de memória é associada à variável `p`. Digamos que a posição 1002 foi escolhida. Em seguida o código da função começa a ser executado. Como o teste `(n==0)` é negativo, pois `n` tem o valor 2, a cláusula `else` é executada. Para calcular o valor a ser atribuído à variável `p`, precisamos calcular `potencia2(n-1)`, i.e., `potencia2(1)`. Nesse instante, a *segunda chamada* da função ocorre.

Ao iniciar a segunda chamada, o que se passará com a variável local `p`? A mesma posição de memória anterior, 1002, será designada para `p`? NÃO! Uma *nova posição de memória* é associada à uma *nova instância* da mesma variável `p`! Ou seja, nessa segunda chamada, uma *nova cópia* de `p` é criada, digamos na posição 1001 de memória. Só então o código da função começa a ser executado nessa segunda chamada recursiva.

IMPORTANTE: cada chamada recursiva de uma função em C aloca novos espaços de memória para todas as variáveis locais definidas pela função, inclusive para aquelas definidas como parâmetros da função.

Retornando à segunda execução da função `potencia2`, o primeiro comando a ser executado é o comando `if`. O teste resulta negativo, pois agora temos o valor 1 na variável `n`. Assim, o braço `else` será acionado de novo e ocorre a *terceira chamada recursiva* da função `potencia2`, agora com o parâmetro `n` valendo 0.

A terceira chamada recursiva entra em cena. De novo, novas posições de memória são alocadas para as variáveis locais. Digamos que a variável `p` agora está alocada à posição 1000 de memória. O código começa a ser executado e o comando `if` é acionado. Agora, o teste resulta positivo, pois `n` vale 0. Então o comando `p = 1` é executado, atribuindo 1 à posição de memória 1000. Continuando, o próximo comando a ser executado é `return p`. Como `p` está associado à posição 1000, o valor retornado será 1. Nesse ponto a terceira chamada recursiva termina. A posição de memória 1000, associada a esta instância da variável `p` é desativada, e a execução volta para o comando imediatamente seguinte àquele que causou essa terceira chamada.

Retornamos, então, para o cálculo da expressão `2*potencia2(0)`, que foi onde se deu a terceira chamada recursiva. Como o valor retornado para `potencia2(0)` foi 1, calculamos `2*1` e obtemos o valor 2, que é atribuído à variável local `p` local, que está na posição de memória 1001, a qual foi associada à variável `p` antes de iniciarmos a execução do código da segunda chamada recursiva. Esse valor 2 é então atribuído à posição de memória 1001. Atribuído o valor à variável `p`, o braço `else` termina e o próximo comando a ser executado é `return p`. De novo, o valor de `p` é 2, armazenado na posição de memória 1001. Esse valor é retornado e a execução da segunda chamada recursiva termina, desalocando a posição de memória 1001.

Finalmente, voltamos à execução do código da primeira chamada recursiva, no ponto imediatamente após o comando que disparou a segunda chamada. Voltamos, então, ao cálculo da expressão `2*potencia2(1)`. Como o valor retornado foi 2, calculamos

$2*2$ e obtemos 4. Esse valor deve ser atribuído à variável local `p`, da primeira chamada recursiva. Lembramos que antes de iniciar a execução do código da primeira chamada, essa variável foi associada à posição de memória 1002. Logo, o valor 4 é atribuído à essa posição de memória. Como isso, o braço `else` termina e devemos executar o próximo comando, que é `return p`. Obtemos o valor corrente de `p` da posição de memória 1002, ficando com 4. Esse valor é retornado e a primeira chamada recursiva finalmente termina. O valor 4 será repassado ao programa principal, que vai imprimi-lo.

Exemplo 3:

Queremos generalizar o programa anterior, de forma que lide com qualquer base (número fracionário) e com qualquer expoente (inteiro positivo ou negativo).

Considere as funções abaixo:

```
// função potencia com expoente positivo
double potPositivo(double base, int exp) {
    if (exp==0) {
        return 1.0;
    }
    else {
        return (base*potPositivo(base, exp-1));
    }
}
// função potencia com qualquer expoente
double potReal(double base, int exp) {
    if (exp >=0) {
        return potPositivo(base, exp);
    }
    else {
        return (1.0/potPositivo(base, -exp));
    }
}
```

A primeira função é uma repetição da função `potencia2`, já discutida. Apenas que a base agora é um número fracionário qualquer. A segunda função meramente chama a primeira se o expoente for positivo, ou chama a primeira com o sinal do expoente trocado, caso seja negativo. Nesse último caso, o valor retornado deve ser o inverso do valor calculado, uma vez que o expoente original é negativo.

Exemplo 4:

Um dos teoremas fundamentais do cálculo diz que toda função contínua deve ter pelo menos um zero entre dois pontos a e b , onde $a < b$, desde que $f(a)$ e $f(b)$ tenham sinais diferentes. Queremos um algoritmo recursivo para obter um desses zeros. Ou seja, queremos um ponto z , onde $a < z < b$, tal que $f(z) = 0$.

O método de Newton-Raphson é uma maneira rápida de se obter um desses zeros. O método calcula o ponto médio entre a e b , $x = (a + b)/2$. Em seguida, testa $f(x) == 0$. Se positivo, achamos o zero desejado. É o caso base.

Suponha negativo. Então $f(x) < 0$, ou $f(x) > 0$. Em qualquer desses casos, sempre teremos $f(x)$ e $f(a)$, ou $f(x)$ e $f(b)$, com sinais trocados. Se tivermos $f(a)$ e $f(x)$ com sinais trocados, então aplicamos o método *recursivamente* entre os pontos a e x , substituindo b por x . É um dos casos do passo indutivo. Note que o comprimento do

intervalo de busca agora diminuiu de $(b - a)$ para $(x - a)$. Se $f(x)$ e $f(b)$ estão com sinais trocados, aplicamos o mesmo método *recursivamente* entre os pontos x e b . É o outro caso do passo indutivo. Agora o comprimento do intervalo de busca também diminuiu de $(b - a)$ para $(b - x)$.

Essas aplicações sucessivas do método vão rapidamente estreitando o intervalo de busca para $1/2$ do intervalo original, depois para $1/4$ do intervalo original, $1/8$ do intervalo,

Uma última observação antes de apresentar o código. Como estamos trabalhando com números fracionários, sempre devemos prever a hipótese de que haja um erro de arredondamento nos cálculos (o computador, na verdade, não trabalha com números reais, mas com uma aproximação deles). Assim, vamos aceitar que encontramos o zero desejado, z , quando o valor absoluto de $f(z)$ estiver dentro de uma tolerância dada, ϵ . I.e., se $|f(z)| < \epsilon$, então aceitamos que $f(z)$ está suficientemente próximo de zero.

Considere o programa abaixo:

```
double func(double x);
// programa para teste
void main(void) {
    double a,b,epsilon;
    double fa,fb;
    double z;
    char denovo[2];
    do {
        printf("Entre com os pontos inicial e final (fracionarios): ");
        scanf("%lf %lf",&a,&b);
        printf("Entre com a tolerancia (fracionario): ");
        scanf("%lf",&epsilon);
        fa = func(a);
        fb = func(b);
        if (fabs(fa)<epsilon) saiSolucao(a,fa);
        else if (fabs(fb)<epsilon) saiSolucao(b,fb);
        else if (((fa>0)&&(fb>0)) || ((fa<0)&&(fb<0))) saiErro(a,b,fa,fb);
        else {
            z = newtonRaphson(a,b,epsilon);
            saiSolucao(z, func(z));
        }
        printf("Quer tentar de novo (S/N)?: ");
        scanf("%s",denovo);
        denovo[0]=toupper(denovo[0]);
    } while (denovo[0]!='S'); // termina se entrar algo diferente de S, ou s
    return;
}
// função continua
double func(double x) {
    return (3.4*potReal(x,5) - 1.2*potReal(x, 2) + 5.89*potReal(x, -2) +
9.0);
}
// procura zero
double newtonRaphson(double a, double b, double epsilon) {
    double x, z, fb;
    fb = func(b);
    x = (a+b)/2.0;
    z = func(x);
    if(fabs(z)<epsilon) { return x; }
    if ( ((z>0)&&(fb<0)) || ((z<0)&&(fb>0)) )
    { return newtonRaphson(x,b,epsilon); }
    else
    { return newtonRaphson(a,x,epsilon); }
}
// imprime solucao
void saiSolucao(double a, double fa) {
```

```

        printf("Zero em %1e. Conferindo:\n\t func(%1e) = %1e\n",a,a,fa);
        return;
    }
    // imprime erro
    void saiErro(double a, double b, double fa, double fb) {
        printf("Valores com sinais invalidos! Conferindo:\n\t func(%1e) =
                %1e\n\t func(%1e) = %1e ... \n",a,fa,b,fb);
        return;
    }
    // função potencia com expoente positivo
    double potPositivo(double base, int exp) {
        if (exp==0) {
            return 1.0;
        }
        else {
            return (base*potPositivo(base, exp-1));
        }
    }
    // função potencia com qquer expoente
    double potReal(double base, int exp) {
        if (exp >=0) {
            return potPositivo(base, exp);
        }
        else {
            return (1.0/potPositivo(base, -exp));
        }
    }
}

```

O programa principal pede os valores dos parâmetros de trabalho. Em seguida testa se temos um zero já no ponto [a](#) ou no ponto [b](#). Em caso positivo, imprime a solução. Se não for o caso, o programa testa se os sinais da função nos pontos [a](#) e [b](#) são diferentes. Em caso negativo, uma mensagem de erro é impressa. Se os valores realmente são distintos, é chamado o método de Newton-Raphson para obter um zero no intervalo permitido. Feito isso, o programa pergunta ao usuário se deseja executar o método novamente.

A rotina *recursiva* [newtonRaphson](#) é uma implementação direta do método de Newton-Raphson para obter o zero da função, como já discutido. As rotinas para imprimir solução ou para imprimir mensagem de erro são triviais. E as rotinas (também *recursivas*) para cálculo de potência já foram discutidas no exemplo anterior. No caso desse exemplo, usamos para teste a função polinomial

$$f(x) = -3.4x^5 + 1.2x^2 + 5.89x^{-2} + 9.0$$

Se quisermos trocar a função, teríamos que descrevê-la na rotina [func](#) e recompilar o programa. Uma situação não muito confortável. Uma melhoria seria usar apontadores.

Exemplo 5:

O problema da população de coelhos. José comprou um casal de coelhos jovens e gostaria de calcular a população de coelhos que terá à medida que o tempo for passando. As restrições são as seguintes:

1. Inicialmente, José comprou apenas um casal de coelhos jovens.
2. Cada casal de coelhos jovens leva uma unidade de tempo para amadurecer e estar apto para procriar.
3. Quando já está maduro, cada casal de coelhos gera um novo casal de coelhos por unidade de tempo.
4. Coelhos nunca morrem.

Assim, José tem:

1. No instante 1, um casal de coelhos jovens
2. No instante 2, um casal de coelhos adultos
3. No instante 3, um casal de coelhos adultos mais um casal de coelhos jovens gerados pelo casal adulto do item 2. Portanto dois casais.
4. No instante 4, dois casais adultos, mais um casal jovem (gerado pelo único casal adulto). Portanto 3 casais.
5.

Seja $C(n)$ o número de casais que José terá no instante n . Assim, nos *casos base*, com $n=1$ e $n=2$, teremos $C(1) = 1$, e $C(2) = 1$.

No *passo indutivo*, no instante n (com $n > 2$), José terá os casais que tinha no instante anterior (coelhos não morrem), ou seja, $C(n-1)$. Mais os casais gerados pelos casais adultos no instante $(n-1)$, pois só esses podem gerar outros casais. Como o tempo para casais jovens se tornarem adultos é de 1 período, os casais aptos a procriar no instante $(n-1)$ são todos os casais presentes no instante $(n-2)$. Assim, devemos acrescentar mais $C(n-2)$ casais gerados no instante $(n-1)$. A população de casais no instante n será, pois, dada por $C(n) = C(n-1) + C(n-2)$.

Esta é a função de Fibonacci:

Caso base: $C(1) = C(2) = 1$

Passo indutivo: $C(n) = C(n-1) + C(n-2)$, para $n > 2$.

O seguinte programa calcula essa função *recursivamente*:

```
// Programa recursivo para calcular a funcao de Fibonacci
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
// declaração de funções
int fibonacci(int n);

// programa para teste
void main(void) {
    int n;
    char denovo[1];
    do {
        printf("Entre com o parametro(inteiro): ");
        scanf("%d", &n);
        if (n <= 0) { printf("Valor %ld eh invalido.\n"); }
        else printf("Fibonacci de %ld = %ld\n", n, fibonacci(n));
        printf("Quer tentar de novo (S/N)?: ");
        scanf("%s", denovo);
        denovo[0] = toupper(denovo[0]);
    } while (denovo[0] != 'S'); //termina se entrar algo diferente de S, ou s
    return;
}

// função de Fibonacci
int fibonacci(int n) {
    if ((n==1) || (n==2)) { return 1; }
    else {
        return fibonacci(n-1) + fibonacci(n-2);
    }
}
```

O programa principal repete o que já foi discutido em exemplos anteriores. A rotina `fibonacci` é uma implementação direta da recursão discutida nesse exemplo.

Ocorre que esta implementação é bem ineficiente. Repare que para calcular `fibonacci(50)`, por exemplo, a rotina calcula `fibonacci(49)` e, em seguida, calcula `fibonacci(48)`. Suponha que `fibonacci(48)` acabou de ser calculado e vamos agora calcular `fibonacci(49)`. Mas, $\text{fibonacci}(49) = \text{fibonacci}(48) + \text{fibonacci}(47)$. Portanto, a rotina vai *recalcular novamente* `fibonacci(48)`. E o mesmo se dá para `fibonacci(47)`. E, o que é muito pior, esses recálculos se repetem por toda a recursão, à medida que vamos calculando outros valores de Fibonacci quando o parâmetro decresce. Tente executar o programa (em uma máquina rápida) para valores de `n` a partir de 40 (ou de 30, se sua máquina tiver menos capacidade), aumentando o valor do parâmetro de uma unidade de cada vez. Observe os aumentos no tempo de espera para que o computador produza o resultado.

Melhor seria se fôssemos armazenando os valores calculados à medida que a recursão progride. Considere o programa:

```
// Programa recursivo para calcular a funcao de Fibonacci
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define TAM 1000
// declaração de funções
long long unsigned int fibonacci(long long unsigned int *fibos, int n);
// programa para teste
void main(void) {
    int n, i;
    char denovo[1];
    // variavel global para armazenar os valores calculados
    long long unsigned int fibos[TAM+1];
    do {
        printf("Entre com o parametro(inteiro): ");
        scanf("%d", &n);
        if (n<=0) { printf("Valor %ld eh invalido.\n"); }
        else {
            for (i=1; i<=TAM; i++) { fibos[i]=0; }
            printf("Fibonacci de %ld = %ld\n", n, fibonacci(fibos, n));
        }
        printf("Quer tentar de novo (S/N)?: ");
        scanf("%s", denovo);
        denovo[0]=toupper(denovo[0]);
    } while (denovo[0]!='S'); //termina se entrar algo diferente de S, ou s
    return;
}

// função de Fibonacci
long long unsigned int fibonacci(long long unsigned * fibos, int n) {
    if ( ((n==1) || (n==2)) ) {
        if (!fibos[n]) {fibos[n]=1;}
        return 1;
    }
    else {
        if (!fibos[n]) {
            fibos[n]=fibonacci(fibos, n-1)+fibonacci(fibos, n-2);
        }
        return fibos[n];
    }
}
```

O programa principal é idêntico. Nesse caso, o vetor `fibos` é usado para armazenar os valores já calculados. Note que esse vetor é reinicializado com zeros a cada iteração, de modo que não temos aproveitamento cumulativo entre duas execuções. A

rotina `fibonacci` foi modificada para que só calcule algum valor recursivamente se este valor ainda não está armazenado no vetor `fibos`.

Como antes, execute o programa passando o valor inicial de 40 e aumentando o valor do parâmetro uma unidade de cada vez. Você sentirá uma melhora muito significativa na velocidade de cálculo quando comparada ao programa anterior.