Notas de aula – AED 1 – recursão
Prof. Jefferson T. Oliva

Na matemática, vários objetos são definidos através de um processo que os produz:
- Exemplo: função fatorial (n!)

Um outro exemplo seria a soma de todos os elementos de um vetor, onde, para que possamos incluir v[i] no somatório, devemos ter somas os elementos de v[0] a v[i-1] e assim, sucessivamente.

Exemplo de um algoritmo para o cálculo do fatorial de n:

```
int factorial(int n){
        int i, f = 1;
        for (i = n; i > 1; i--)
                f *= i;
        return f;
}
```

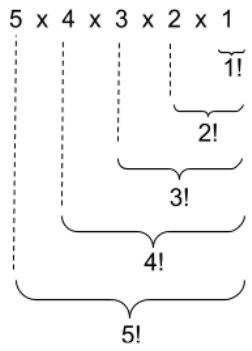
Esse algoritmo é chamado de iterativo pois requer a repetição explícita de um processo até que determinada condição seja satisfeita.

O algoritmo anterior pode ser traduzido para uma função que retorne n! quando recebe n como parâmetro.
- n! = 1, se n = 0
- n! = n * (n − 1)!, se n > 0

A função fatorial anterior está definida em termos de si mesma, ou seja, recursivamente.

Implementação recursiva da função fatorial.

```
int factorial(int n){
        if (n <= 1)
                return 1;
        else
                return n * factorial(n - 1);
}
```



Assim, o problema de calcular o fatorial de 5 pode ser quebrado em partes cada vez menores até encontrar a instância mais simples do problema, que é 1!
Resolvendo 1! é possível resolver 2!
Resolvendo 2! é possível resolver 3!
Resolvendo 3! é possível resolver 4!
Resolvendo 4! é possível resolver 5!

Recursão

Uma função recursiva é uma função chama a si própria.

Recursão é comumente aplicada em problemas de divisão e conquista, onde um problema grande é dividido em vários problemas menores. Quando o problema menor é resolvido, um outro pouco maior é resolvido em seguida por meio da combinação de soluções menores. Isso é feito até que o problema seja resolvido completamente. Não iremos falar muito de divisão e conquista, pois o veremos detalhadamente em AED2.

Exemplos de aplicação de recursão: árvores, ordenação (alguns algoritmos), números naturais, compiladores, etc

Recursão significa repetição. Para a implementação de recursão, há uma coisa que devemos nos preocupar: critério de parada (assim como em um comando repetitivo). Da mesma forma que num comando repetitivo, a parada é uma preocupação, o mesmo ocorre com a recursão; Permite que o procedimento pare de se executar.

Todo algoritmo recursivo deve ter:
- Caso base (critério de parada)
- E caso indutivo, onde ocorre a redução do problema (passa a ficar mais próximo do caso vbase)

Resolução de problemas recursivos
- Instância pequena: resolvida diretamente
- Instância grande: reduzida a uma instância menor do mesmo problema. O método é aplicado à instância menor. Após chegar à instância menor e resolvê-la, deve-se voltar até a instância original.

Entretanto, é importante ressaltar que se não definirmos corretamente o critério de parada, a recursão poderá ser infinita.

Três regras para a recursão
1 - Saber quando parar (critério de parada/caso base). Qualquer função recursiva deve verificar se a jornada já terminou (caso base) antes da nova chamada recursiva.

2 - Decidir como fazer o primeiro passo. Pensar em como quebrar um problemas em subproblemas que possam ser resolvidos instantaneamente. Em outras palavras, decidir como modificar o seu estado (ou entrada(s)) em direção ao caso base.

3 - Analisar o problema de forma que possa ser dividido em subproblemas (Passo indutivo), ou seja, encontrar uma maneira da função chamar a si mesma (recursivamente), passando por parâmetro um problema menor resultante da segunda regra.

Quando uma recursão é realizada, várias ações ocorrem:
- Em qualquer chamada recursiva é criado um registro de ativação na pilha de execução. Registro de ativação: contém todos os dados relativos a um procedimento: endereço de retorno, parâmetros, dados locais e temporários, etc.
- Assim, novas e distintas cópias dos parâmetros passados por valor e variáveis locais são criadas e colocadas na pilha
- A posição que chama a função é colocada em estado de espera, enquanto que o nível gerado recursivamente esteja executando
- No final da execução, o registro é desempilhado e a execução volta ao subprograma que chamou

ver slides de 16 a 32

O desenvolvimento de uma solução recursiva nem sempre é fácil e em outros casos, o problema é naturalmente recursivo (veremos um exemplo hoje).

Normalmente, não há porque propor uma solução recursiva, pois muitas vezes pode acarretar no uso de grande quantidade de recursos computacionais.

Entretanto, para alguns problemas a solução recursiva é mais elegante e mais fácil em relação de desenvolver soluções iterativas.

Para cada algoritmo recursivo, pode ser desenvolvido um algoritmo iterativo (muitas vezes bem difícil!).

Exemplos de Problemas Recursivos: torre de Hanoi

Dado três torres (A, B e C) e n discos de diâmetros diferentes, o disco de menor diâmetro deve sempre estar em cima do disco de maior diâmetro.

Inicialmente, todos discos deve estar na torre A

Problema: como colocar todos os discos na torre C, utilizando a torre intermediária B, sem inverter a ordem dos diâmetros em nenhum torre?

Se há solução para n - 1 discos, então há solução para n discos.

No caso trivial de n = 1, a solução é simples, basta transferir para a torre C

A solução para n discos é realizada em termos de n – 1
Procedimento:
- Se n = 1, mover o disco da torre A para C
- Mova os n - 1 discos de A para B, usando C como auxiliar
- Mova o último disco de A para C
- Mova os n - 1 discos de B para C, usando A como auxiliar

```
void hanoi(char de, char para, char via, int n){
   if (n >= 1){ // Caso base: quando n for igual a zero (nada é feito)
      hanoi(de, via, para, n - 1);
      printf("disco %d de %c para %c\n", n, de, para);
      hanoi(via, para, de, n - 1);
   }
}
```

Ver slide 39

Exemplos de Problemas Recursivos: sequência de Fibonacci

É uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores. A sequência recebeu o nome do matemático italiano Leonardo de Pisa, mais conhecido por Fibonacci, que descreveu, no ano de 1202, o crescimento de uma população de coelhos, a partir desta. Esta sequência já era, no entanto, conhecida na antiguidade. A sequência de Fibonacci tem aplicações na análise de mercados financeiros, na ciência da computação e na teoria dos jogos. Também aparece em configurações biológicas, como caracol, desenrolar da samambaia ... [Wikipédia]

Sequência de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Aplicação interessante: conversão de milhas para quilômetros: para converter 5 mil, pode ser utilizado o número seguinte da sequência de Fibonacci (8 km). Não funciona para todos os casos e também, os valores de milhas são fracionários. Em outras palavras, com Fibonacci podemos obter um valor aproximado.

Ver, no slide 41, a equação

```
int fib(int n){
        if (n <= 0)
                return 0;
        else if (n == 1)
                return 1;
        else
                return fib(n - 1) + fib(n - 2);
}
```

Ver slides 43 e 44

Implementar exemplos em listas encadeadas, como imprimir uma lista de trás para frente, inserção, etc.

Simulando a Recursividade

Todo problema com solução recursiva pode ser resolvido iterativamente.

Basta examinar com detalhes os mecanismos usados para implementar a recursividade para que seja possível simulá-los usando técnicas não-recursivas.

A solução recursiva geralmente é mais cara computacionalmente do que a solução não-recursiva.

A possibilidade de gerar uma solução não-recursiva a partir de um algoritmo recursivo é muito interessante em várias situações.

Cada vez que uma função recursiva chama a si mesma, uma área de dados totalmente nova precisa ser alocada.

Essa área de dados precisa ter todos os parâmetros, variáveis locais, temporárias e um endereço de retorno.

Cada chamada acarreta a alocação de uma nova área de dados e toda a referência a um item na área de dados da função destina-se à área de dados da chamada mais recente. Da mesma forma, todo retorno provoca a liberação da atual área de dados e a área de dados alocada anteriormente torna-se a área atual.

Geralmente, uma versão não-recursiva (iterativa) de um programa executará com mais eficiência, em termos de tempo e espaço, do que uma versão recursiva.

Na versão iterativa, o trabalho extra dispendido para entrar e sair de um bloco é evitado. Também, muita atividade de empilhamento e desempilhamento pode ser evitada

Exemplo de quando não usar recursividade: sequência de Fibonaci, pois essa solução é extremamente ineficiente, pois refaz o mesmo cálculo diversas vezes.

Deve-se evitar uso de recursividade quando existe solução óbvia por iteração.

```
int fibonacci(int n){
        int i, fib1, fib2, fibN;
        if (n <= 0)
                return 0;
        else if (n == 1)
                return 1;
        else{
                fib1 = 0;
                fib2 = 1;
                fibN = 0;
                for (i = 2; i <= n; i++) {
                        fibN = fib1 + fib2;
                        fib1 = fib2;
                        fib2 = fibN;
                }
                return fibN;
        }
}
```

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Rosa, J. L. G. Recursão em C. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2008.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007