Programação I

Recursividade

Estrela Ferreira Cruz

1

Objetivos aula

- Recursividade:
  - Apresentação do conceito: definição de recursividade
  - Critério de paragem e passo recursivo
  - Tipo de recursividade: direta e indireta
  - Método iterativo versus método recursivo
  - Vantagens/desvantagens do uso de recursividade
  - Apresentação de exemplos práticos

2

2

Recursividade

Existem dois métodos essenciais para a conceção e desenvolvimento de algoritmos: método iterativo e o método recursivo.

- Método iterativo – Este método de desenvolvimento de algoritmos tem por base a divisão do problema principal em sub-problemas. Determinados sub-problemas e respetivas tarefas têm um padrão de comportamento repetitivo o que leva à utilização de estruturas cíclicas de controlo do problema.
- Método recursivo – A recursividade é uma característica fundamental para a conceção e construção de algoritmos. Esta característica é interpretada como a capacidade de uma função (ou método) se invocar a si própria.

3

3

Recursividade

Método recursivo

Para se aplicar a recursividade a um determinado problema, é necessário considerar:

- O caso base (de paragem) – existência de um, ou mais casos que não necessitam de operações significativas de computação.
- O passo recursivo – representação dos diferentes subprogramas que recursivamente chegarão ao caso base.

4

4

Recursividade

A Recursividade pode ser descrita como uma abordagem algorítmica em que uma função se invoca a ela própria.

A recursividade pode ser de dois tipos:

- Direta - Quando uma função se invoca a ela mesma no seu corpo da função.
- Indireta -Quando uma função f invoca uma outra função g que por sua vez volta a invoca a função f.

5

5

Recursividade

Existem algumas regras que devem ser seguidas para ser conseguida uma boa programação, usando recursividade:

- A primeira instrução de uma função recursiva deve ser o critério de paragem, isto é, a ou as condições que se devem verificar para a função parar de se invocar a ela própria;
- Só depois deve ser escrita a chamada recursiva, sempre relativa a um subconjunto.

6

6

Recursividade

Exemplo 1: Fatorial

O fatorial de um número n pode ser definido como

n! = 1 x 2 x 3 x ... x n

ou

n! = n x … x 3 x 2 x 1

E também pode ser definido por recorrência, ou seja, **recursivamente** através das seguintes duas regras:

se n = 0 então n! = 1

Senão n! = n x (n-1)!

Neste caso, definimos a função fatorial em termos da própria função fatorial.

7

7

Recursividade

Exemplo: O próximo programa calcula o fatorial de um numero, de uma forma iterativa:

```
long fatorial( int n ) {
        int i,res=1;
        for( i=n; i>1; i-- ) {
                res = res * i;
        }
        return res;
}
int main() {
        int n=0;
        long result=0;
        printf("introduza um numero para calcular o fatorial\n");
        scanf("%d",&n);
        result=fatorial(n);
        printf("o fatorial de %d é %ld\n",n, result);
        return 0;
}
```

8

8

Recursividade

Exemplo: A função fatorial também poderia ser implementada de uma forma recursiva, como se pode ver no exemplo seguinte:

```
long factorial( int n ) {
        if (n<=0) return 1;
        return n* factorial(n-1);
}


int main() {
        int n=0;
        long result=0;
        printf("introduza um numero para calcular o fatorial\n");
        scanf("%d",&n);
        result=fatorial(n);
        printf("o fatorial de %d é %ld\n",n, result);
        return 0;
}
```

9

9

Recursividade

Exemplo 2: O próximo programa calcula o somatório dos números inteiros positivos até ao número n:

```
int soma( int n ) {
        int i,tot=0;
        for( i=1; i<=n; i++ ) {
                tot = tot + i;
          }
        return tot;
}

int main(){
        int n, result=0;
        printf("introduza um numero inteiro\n");
        scanf("%d",&n);
        result=soma(n);
        printf("A soma dos nº até %d é %ld\n",n, result);
        return 0;
}
```

10

10

Recursividade

Exemplo 2: A função soma() também pode ser implementada usando recursividade, da seguinte forma:

```
int soma( int n ) {
        if (n==0) return 0;
        return n + soma(n-1);
}

int main(){
        int n, result=0;
        printf("introduza um numero inteiro\n");
        scanf("%d",&n);
        result=soma(n);
        printf("A soma dos nº até %d é %ld\n",n, result);
        return 0;
}
```

11

11

Recursividade

NOTA: É necessário ter muito cuidado na construção de funções recursivas.

Assim:

- É absolutamente necessário assegurar que existe um critério de paragem.
- Este vai determinar quando a função deve parar de se invocar a ela própria.
- Isto impede que a função se chame infinitas vezes, ou seja, impede que a função seja "infinita".

12

12

Recursividade

Exemplo 3: A sucessão de *Fibonacci*

Os números da **sucessão de Fibonacci** são definidos da seguinte forma:

- O **primeiro número é 1**.
- O **segundo também é 1**.
- O **enésimo número é definido como sendo a soma dos dois números anteriores**.

Ou seja:

se n=1 então fib(n) = 1

Senão

se n=2 então fib(n) = 1

Senão

fib(n) = fib(n-1) + fib(n-2)

13

13

Recursividade

Exemplo 3: A implementação recursiva é uma tradução direta da definição da sucessão de Fibonacci, como se pode ver no exemplo seguinte:.

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

14

14

Recursividade

Exemplo 3:

A função iterativa que implementa a sucessão de fibonacci.

```
long fibIt( int n ) {
int f=0, i=0, fAnt=1, f2Ant=1;

for (i=3; i<=n; i++){
        f = fAnt + f2Ant;
        f2Ant = fAnt;
        fAnt = f;
}
return f;
}
```

15

15

Recursividade

Exemplo 3: Compare o tempo de resposta das duas funções anteriores (iterativa e recursiva). Aumento numero até ....

```
int main(){
int n;
long result=0;

printf("introduza um numero inteiro\n");
scanf("%d",&n);
result=fibIt(n);
printf("O nº %d da sucessão de fibonacci iterativo = %ld\n",n, result);
result=fib(n);
printf("O nº %d da sucessão de fibonacci recursivo = %ld\n",n, result);
return 0;
}
```

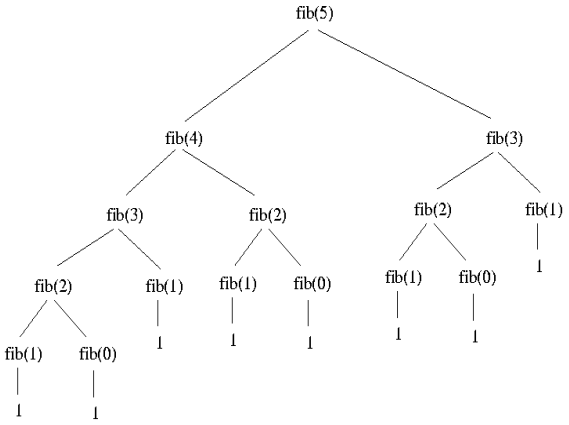
16

16

Recursividade

A solução recursiva é mais simples de programar do que a versão iterativa. Mas é muito ineficiente porque de cada vez que a função fib() é chamada, a dimensão do problema reduz-se apenas de uma unidade (de n para n-1), mas são feitas duas chamadas recursivas.

Isto dá origem a uma explosão combinatória e o computador acaba por efetuar os mesmos cálculos várias vezes.



17

17

Recursividade

Exemplo 3: A sucessão de *Fibonacci*

Para calcular fib(5) temos de calcular fib(4) e fib(3).

Para calcular fib(4) temos de calcular fib(3) e fib(2).

Para calcular fib(3) temos de calcular fib(2) e fib(1).

.....

Este tipo de processamento é ineficiente porque obrigamos o computador a fazer trabalho "desnecessário".

- No exemplo concreto, para calcular fib(5) temos de calcular: fib(4) 1 vez; fib(3) 2 vezes; fib(2) 3 vezes; fib(1) 5 vezes; fib(0) 3 vezes.
- Numa implementação iterativa, apenas era necessário calcular fib(5), fib(4), fib(3), fib(2), fib(1) e fib(0) 1 vez.

18

18

Recursividade

Para todo algoritmo recursivo existe um outro correspondente iterativo, que executa a mesma tarefa.

- Em muitos casos a implementação iterativa é mais eficiente que a respetiva implementação recursiva.
- Contudo, existem implementações recursivas que são igualmente eficientes.
- Muitas vezes, é evidente a natureza recursiva do problema a ser resolvido. Por isso, a implementação recursiva pode ser muito mais simples que a correspondente implementação iterativa.
- Em geral, os algoritmos recursivos possuem código mais claro (legível) e mais compacto do que os correspondentes iterativos.

19

19

Recursividade

Existem, no entanto, algumas desvantagens:

- Os algoritmos recursivos quase sempre consomem mais recursos do computador, como memória, etc.
- Os algoritmos recursivos tendem a apresentar um desempenho inferior aos iterativos;
- Algoritmos recursivos são mais difíceis de serem depurados, especialmente quando a profundidade de recursão é elevada, ou seja, quando o número máximo de chamadas simultâneas é alto.

20

20

Recursividade

Quando devemos usar recursividade?

Devemos usar recursividade sempre que:

- O problema é de natureza recursiva. Neste caso, a solução recursiva é mais natural e fácil de implementar dando origem a um algoritmo mais simples, elegante e fácil de compreender;
- A solução recursiva não implica uma excessiva repetição do cálculo dos mesmos valores.
- A solução iterativa equivalente é muito complexa, aumentando assim a probabilidade de ocorrência de erros.

21

21

Recursividade

Exercícios:

1. Desenvolver um programa que, dando o valor de x e y calcule x^y recorrendo ao uso da recursividade. Desrecursive a função implementada anteriormente.

2. Implementar um programa que leia do teclado um número inteiro positivo e escreva para o ecrã o respetivo **número quadrático**. Desrecursive a função implementada anteriormente. Os números quadráticos são definidos pela seguinte relação de recorrência:

$$\begin{cases} \text{Se } n=1,\ Q(n)=1 \\ \text{Se } n>1,\ Q(n)=Q(n-1)+2n-1 \end{cases}$$

3. Implementar uma função recursiva que calcule o somatório de um array de inteiros. Os números do array deverão ser introduzidos pelo utilizador.

22

22

Bibliografia

- Programação Avançada Usando C, António Manuel Adrego da Rocha, ISBN: 978-978-722-546-0.
- Schildt, Herbert: C the complete Reference, McGraw-Hill, 1998.
- Algoritmia e Estruturas de Dados, José Braga de Vasconcelos, João Vidal de Carvalho, ISBN: 989-615-012-5.
- Linguagem C, Luís Manuel Dias Damas, ISBN: 972-722-156-4.
- Elementos de Programação com C - Pedro João Valente D. Guerreiro, 3ª edição, ISBN: 972-722-510-1.
- Introdução à Programação Usando C, António Manuel Adrego da Rocha, ISBN: 972-722-524-1.