Aula 06

Recursividade

Introdução ao Conceito

Programação II, 2020-2021

2021-04-14

DETI, Universidade de Aveiro

06.1

Objectivos:
- Funções recursivas.

Conteúdo

06.2

1 Introdução

- Se tivesse de descrever o que é uma boneca *matryoshka*, como o faria?
- Poderia dizer que é uma boneca oca que contém outra boneca oca, que contém outra e assim sucessivamente.
- Mas também pode dar uma definição alternativa, porventura mais simples:
  - Uma boneca *matryoshka* é uma boneca oca que contém outra boneca *matryoshka*.
- 06.3 Este é um exemplo de uma *definição recursiva*.

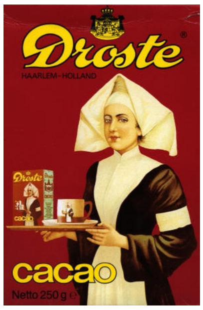
2 Definição

Definição Recursiva: Uma definição de um conceito diz-se recursiva se envolver uma ou mais instâncias do próprio conceito.

Recursividade: Se ainda não entendeu, ver *recursividade*.

Podemos encontrar recursividade um pouco por todo o lado:

- Na descrição de certas formas geométricas.
- Nas imagens refletidas em espelhos paralelos.
- Na definição de certas funções matemáticas.
- Na sintaxe das linguagens de programação.



06.4 (circa 1904)

3 Complexidade

- Como veremos, as definições recursivas podem também aparecer nos dois aspectos essenciais da programação:
  - nos *algoritmos*;
  - nas *estruturas de dados*.
- Usam-se definições recursivas porque frequentemente permitem descrever problemas complexos de forma *simples*.
- Desde Programação 1 temos vindo a apresentar e aplicar tecnologias e métodos para lidar com a complexidade inerente à resolução de problemas.
- Uma característica comum à maioria delas é o facto de *reduzirem a redundância* do código necessário para a solução.
- 06.5 A estratégia consiste em tirar proveito das *semelhanças formais* entre as várias partes do código.

Gestão da Complexidade

Vejamos alguns casos:

- **Variáveis**: permitem que o mesmo código produza resultados diferentes para dados diferentes.
- **Instrução iterativa**: permite substituir uma sequência de comandos estruturalmente semelhantes pela repetição de um único comando (geralmente parametrizado com variáveis auxiliares).
- **Funções**: permitem modularizar certas sequências de operações, transformando-as numa nova operação abstrata, que pode ser reutilizada múltiplas vezes. Promovem uma separação clara entre a *utilização* da função e a respectiva *implementação*. Quem a utiliza, delega a responsabilidade da resolução na função. Quem a implementa, pode livremente escolher o melhor algoritmo.

06.6

4 Relação de Recorrência

- O caso das funções é particularmente interessante. Se quem as *implementa* é livre para escolher o melhor algoritmo, porque não escolher um que *utiliza* a própria função?
- Se o problema se presta a ser descrito recursivamente, então porque não implementá-lo da mesma forma?
- Para se poder fazer isso mesmo torna-se necessário ter uma descrição recursiva formal do problema: esse é o papel das *Relações de Recorrência*.
- Uma relação de recorrência é uma formulação recursiva formal de um problema.
- As relações de recorrência podem ser sempre implementadas de uma forma *iterativa* ou de uma forma *recursiva*.
- A implementação recursiva é estruturalmente muito próxima da própria relação de recorrência (donde resulta a sua simplicidade).

06.7

5 Exemplo 1: A Função Factorial

- Fórmula iterativa:

$$n! = \begin{cases} \prod_{k=1}^{n} k & , n \in \mathbb{N} \\ 1 & , n = 0 \end{cases}$$

- Fórmula recursiva (relação de recorrência):

$$n! = \begin{cases} n \times (n-1)! & , n \in \mathbb{N} \\ 1 & , n = 0 \end{cases}$$

06.8

Exemplo: a função factorial

Implementação Iterativa

```java
static int factorial(int n)
{
   assert n >= 0;

   int result = 1;

   for (int i=2; i <= n; i++)
      result = result * i;

   return result;
}
```

Implementação Recursiva

```java
static int factorial(int n)
{
   assert n >= 0;

   int result = 1;

   if (n > 1)
      result = n * factorial(n - 1);

   return result;
}
```

chamada recursiva

Implementação Iterativa	Implementação Recursiva
$n! = 1 \times 2 \times \cdots \times (n-1) \times n$	$n! = n \times ((n-1) \times \cdots \times (2 \times (1)) \cdots)$
O índice pode variar do caso limite 0 até ao valor n, ou *vice-versa*.	O argumento varia na direcção do caso limite (de n até 0).

06.9

6 Relação de Recorrência: Síntese

- *Método Iterativo* (Repetitivo)
  - O algoritmo assenta num ciclo em que o índice pode variar desde o valor correspondente às situações limite até ao valor pretendido.
- *Método Recursivo*

– Uma solução recursiva para um problema é expressa em função de si própria.
– Para que se atinja uma solução, cada invocação recursiva deve estar mais próxima de uma situação limite.
– Método poderoso e compacto de resolução de problemas mas potencialmente menos eficiente em termos de recursos pois tem de guardar o estado das várias invocações da função.

06.10

7 Exemplo 2: Cálculo das Combinações

- Fórmula:

$$
\begin{aligned}
C_k^n = \frac{A_k^n}{A_k^k} &= \frac{n \times (n-1) \times \cdots \times (n-k+1)}{k!} \\
&= \frac{n!}{(n-k)! \times k!} \text{ , com } n,k \in \mathbb{N}_0 \land n \geq k
\end{aligned}
$$

- A aplicação destas fórmulas pode levantar problemas de cálculo numérico devido ao facto de os registos internos de armazenamento de um valor terem uma capacidade limitada.
- Exemplo:

$$C_{23}^{25} = \frac{15511210043330985984000000}{51704033477769953280000} = 300$$

- Para representar estes números necessitaríamos de pelo menos 84 bits (mesmo o tipo `long` tem apenas 64).
- Solução?

06.11

Exemplo 2: Combinações – Relação de Recorrência

- Demonstração:

$$
\begin{aligned}
C_k^n &= \frac{n!}{(n-k)! \times k!} = \frac{(n-1)! \times (k+n-k)}{(n-k)! \times k!} \\
&= \frac{(n-1)! \times k}{(n-k)! \times k!} + \frac{(n-1)! \times (n-k)}{(n-k)! \times k!} \\
&= \frac{(n-1)!}{(n-k)! \times (k-1)!} + \frac{(n-1)!}{(n-k-1)! \times k!} \\
&= C_{k-1}^{n-1} + C_k^{n-1}
\end{aligned}
$$

- Relação de recorrência:

$$
\begin{aligned}
C_k^n &= C_{k-1}^{n-1} + C_k^{n-1} \quad \text{, com } n,k \in \mathbb{N} \land n > k \\
C_0^n &= 1 \quad \text{, com } n \in \mathbb{N}_0 \qquad \text{(caso limite)} \\
C_n^n &= 1 \quad \text{, com } n \in \mathbb{N}_0 \qquad \text{(caso limite)}
\end{aligned}
$$

06.12

Exemplo Combinações: Implementação Recursiva

```
static int combNKK(int n, int k)
{
   assert 0 <= k && k <= n;

   int result = 1;

   if (k > 0 && k < n)
      result = combNKK(n-1, k-1) + combNKK(n-1, k);

   return result;
}
```

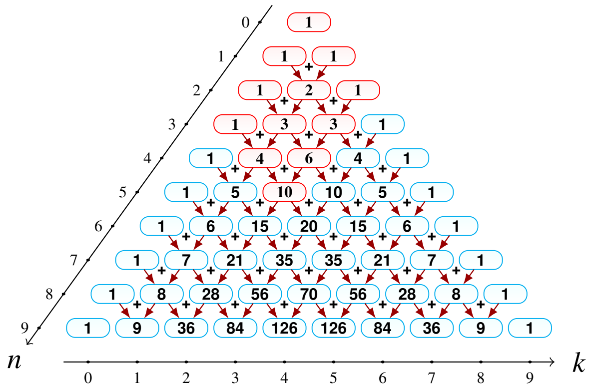
chamadas recursivas

- Método Recursivo:
  - Simples;
  - Compacto;
  - Legível;
  - Fácil detectar erros.
- E se tentarmos implementar uma solução com o método iterativo?

06.13

Exemplo Combinações: Implementação Iterativa 1

- Triângulo de Pascal:



$$C_2^5 = C_1^4 + C_2^4 \begin{cases} C_1^4 = C_0^3 + C_1^3 \{\cdots \\ C_2^4 = C_1^3 + C_2^3 \{\cdots \end{cases}$$

06.14

Exemplo Combinações: Implementação Iterativa 1

- Necessitamos de um *array* de $k+1$ elementos para guardar os valores de uma linha (inicializado a zeros).
- O processo iterativo pode seguir as regras seguintes:
  1. existem $n+1$ iterações (uma por linha);
  2. a primeira linha ($n = 0$) tem apenas o valor 1 (no posição $k = 0$ do *array*), esse valor manter-se-á fixo para todas as linhas;
  3. para as restantes n linhas, os valores do *array* desde o índice 1 até ao índice k são calculados como sendo a soma dos dois valores referidos pela relação de recorrência (se o índice do array for i, então será a soma dos valores com índice $i-1$ e i).
- O resultado é o elemento de índice k da linha n.
- Este algoritmo pode ser optimizado considerando as seguintes factos:
  - Não é necessário calcular um triângulo completo (para C_2^5 bastam os valores assinalados a vermelho na figura).
  - O triângulo de Pascal é simétrico, por isso basta calcular metade.
- O programa mostrado a seguir faz todas essa optimizações.

06.15

Exemplo Combinações: Implementação Iterativa 1

```
static int combIter1(int n,int k)
{
   assert 0 <= k && k <= n;

   int result = 1;
   if (k > 0 && k < n) {
      int kMin = k < n-k ? k : n-k; // minimo(k, n-k)
      int[] linha = new int[k + 1];
      int c = 0;
      int cIni = 1;
      linha[0] = 1;
      for(int l = 1;l <= n;l++) {
         if (l > n-kMin+1)
            cIni++;
         for(c = kMin;c >= cIni;c--)
            linha[c] = linha[c]+linha[c-1];
      }
      result = linha[kMin];
   }

   return result;
}
```

06.16

Exemplo Combinações: Implementação Iterativa 2

- Há uma solução iterativa mais simples e mais eficiente.
- Baseia-se em construir os elementos necessários do triângulo diagonal-a-diagonal em vez de linha-a-linha.
- O algoritmo segue os passos:
  1. Começamos com um *array* de $k+1$ elementos iniciados com uns, correspondendo aos elementos a vermelho na diagonal descendente mais à direita do triângulo. Esta é a diagonal zero.
  2. Cada iteração do ciclo externo vai construir a diagonal seguinte;
  3. Para isso, o ciclo interno vai "descendo" ao longo da diagonal, adicionando a cada elemento do array (que ainda traz o valor da diagonal anterior) o elemento anterior do array (que tem o novo valor acabado de calcular).
  4. O ciclo externo é repetido até chegar à diagonal número $n-k$.
  5. No fim, o valor da posição k do array tem o resultado pretendido.
- Na verdade, é mais simples iniciar o array com apenas um 1 na primeira posição e gerar a diagonal zero da mesma forma que as seguintes.

06.17

Exemplo Combinações: Implementação Iterativa 2

- Este algoritmo é preferível ao anterior porque percorre e gera apenas os elementos necessários, de forma regular e sem necessitar de condições extra.

```
static int combIter2(int n,int k)
{
   assert 0 <= k && k <= n;

   int[] diag = new int[k+1];
   diag[0] = 1;
   for (int i = 0; i <= n-k; i++)
      for (int j = 1; j <= k; j++)
         diag[j] += diag[j-1];
   return diag[k];
}
```

- O programa é mostrado a seguir.

06.18

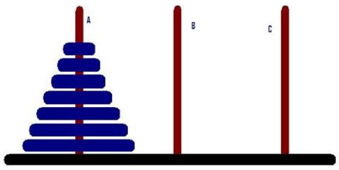
8 Relação de Recorrência: Classificação

Em termos de complexidade do mecanismo de descrição:

- *Simples*: quando há apenas uma chamada recursiva.
  - Exemplo: factorial.
- *Composta*: quando há múltiplas chamadas recursivas.
  - Exemplo: combinações, torres de Hanói.

06.19

9 Exemplo 3: Torres de Hanói



- Este jogo, criado pelo matemático francês Édouard Lucas no Século XIX, é um dos exemplos clássicos que mostram as potencialidades dos algoritmos recursivos.
- Existem três postes onde se podem enfiar discos de diâmetros decrescente.
- O objectivo do jogo é mover todos os discos de um poste para outro, de acordo com as seguintes regras:
  1. Só pode mover um disco de cada vez;
  2. Não pode colocar um disco em cima de outro de menor dimensão.

06.20

Torres de Hanói

Relação de recorrência:

- `moverDiscos(n, tOrigem, tDestino, tAuxiliar)`
  1. `moverDiscos(n−1, tOrigem, tAuxiliar, tDestino)`
  2. `moverUmDisco(tOrigem , tDestino)`
  3. `moverDiscos(n−1, tAuxiliar, tDestino, tOrigem)`

Caso limite:

- `moverDiscos(1, tOrigem, tDestino, tAuxiliar)`
  1. `moverUmDisco(tOrigem, tDestino)`

ou, alternativamente:

- `moverDiscos(0, tOrigem, tDestino, tAuxiliar)`
  1. *(não é preciso fazer nada)*

06.21

Torres de Hanói: Implementação Recursiva

```
static void moverDiscos(int n, String origem, String destino, String auxiliar)
{
   assert n >= 0;

   if (n > 0)
   {
      moverDiscos(n-1, origem, auxiliar, destino);
      out.println("Move disco "+n+" da torre "+origem+" para a torre "+destino);
      moverDiscos(n-1, auxiliar, destino, origem);
   }
}
```

- E se tentarmos implementar uma solução com o método iterativo?
- Existe solução para esse problema (como para qualquer outro algoritmo recursivo) mas a implementação é bastante complexa!

06.22

Torres de Hanói: Implementação Iterativa

```
static void moverDiscosIter(int n, String torreOrigem, String torreDestino, String torreAuxiliar)
{
  assert n >= 1;

  long s = 1; // Stack of bits
  long call;
  int d = n; // disk size
  String src = torreOrigem;
  String dst = torreDestino;
  String aux = torreAuxiliar;
  String tmp;
  boolean finish = false;
  while(!finish)
  {
    while(d > 0)
    {
      tmp = dst; dst = aux; aux = tmp; // swap(dst,aux)
      s = (s << 1) + 1; // push(1)
      d--;
    }
    call = 0;
    while(s != 1 && call != 1)
    {
      call = s % 2;
      s = s >> 1; // pop
      d++;
      if (call == 1)
      {
        tmp = dst; dst = aux; aux = tmp; // swap(dst,aux)
      }
      else
      {
        tmp = src; src = aux; aux = tmp; // swap(src,aux)
      }
    }
    finish = (s == 1) && (call == 0);
    if (!finish)
    {
      out.println("Move disco "+d+" da torre "+src+" para a torre "+dst);
      tmp = src; src = aux; aux = tmp; // swap(src,aux)
      s = s << 1; // push(0)
      d--;
    }
  }
}
```

10 Definição Recursiva: Condições de Terminação

- Para que uma função recursiva termine é preciso que:

1. Exista pelo menos uma alternativa não recursiva (**CASO(S) LIMITE**);
2. Todas as alternativas recursivas ocorram num contexto diferente do original (**VARIABILIDADE**);
3. Em cada alternativa recursiva, o contexto (2) varie de forma a aproximar-se de um caso limite (1) (**CONVERGÊNCIA**).

- As condições (1) e (2) são *necessárias*. As três juntas são *suficientes* para garantir a terminação da recursão.

06.23

Análise dos Exemplos Apresentados

Todos os exemplos de recursividade apresentados até agora verificam estas três condições:

- *Factorial*:
  1. $f(0)$ é um caso limite.
  2. $f(n)$ expresso em função de $f(n-1)$ e $n \neq n-1, \forall n$.
  3. A sucessão $n, n-1, \ldots$ converge para 0.
- *Combinações*:
  1. $C(n,0)$ e $C(n,n)$ são casos limite.
  2. $C(n,k)$ expresso em função de $C(n-1,k)$ e $C(n-1,k-1)$.
  3. n converge para k ou k converge para 0.
- *Torres de Hanói*:
  1. Mover 1 disco (ou 0 discos) é trivial.
  2. $moveTorre(n, \ldots)$ expresso em função de $moveTorre(n-1, \ldots)$.
  3. n converge para 1 (ou 0).

06.24

10.1 Casos Atípicos

Por vezes a evolução dos parâmetros de uma função recursiva pode ser bastante errática e a convergência em direção aos casos limite (condição 3) pode ser menos óbvia, ou mesmo difícil de demonstrar. Vejamos dois casos famosos.

Exemplo de casos atípicos

- Função *McCarthy 91*:

```
static int mc_carthy91(int n) {
   assert n > 0;
   int result;
   if (n > 100)
      result = n - 10;
   else
      result = mc_carthy91(mc_carthy91(n + 11));
   return result;
}
```

  – Sabe-se que termina, mas o tipo complexo de recursão dificulta a demonstração.

- Conjectura de *Collatz* $(3n+1)$:[1]

[1] Ver `http://www.ieeta.pt/~tos/3x+1.html`.

```java
static long collatz(long n) {
   assert n > 0;
   long result = n;
   if (n == 1)
      result = 1;
   else if (n % 2 == 0)
      result = collatz(n / 2);
   else
      result = collatz(3 * n + 1);
   return result;
}
```

– *Acredita-se* que termina sempre, mas ninguém o demonstrou!

06.25

10.2 Casos com Interesse

- Na área da programação, os problemas recursivos considerados são sempre problemas em que as três condições de terminação estão bem identificadas e podem ser implementadas.

06.26