Aula 07

Recursividade

Implementação. Procura com retrocesso

Programação II, 2014-2015

v1.4, 13-06-2015

DETI, Universidade de Aveiro

07.1

Conteúdo

1 Recursão: implementação **1**

2 Conversão entre recursão e iteração **3**
2.1 Iteração para recursão 3
2.2 Recursão para iteração 3

3 *Procura com Retrocesso* **4**
3.1 O Problema das N Rainhas 4
3.2 O Problema dos Labirintos 8

07.2

1 Recursão: implementação

- Não há suporte directo para a recursão de métodos nas linguagens (designadas por *linguagens máquina*) que são directamente executadas pelos processadores (`CPU`, *cores*) existentes nos computadores;
- Assim, para que este mecanismo funcione é necessária uma adequada implementação pelos compiladores (ou interpretadores) das linguagens de programação de mais alto nível (como o Java);

Problema: Permitir uma separação clara entre o código do cliente (que invoca o método) e o código do método , impedindo a interferência (indesejada) entre diferentes invocações do método (incluindo possíveis invocações recursivas).

07.3

Recursão: implementação

- Este objectivo pode ser atingido fazendo com que os métodos, sempre que são invocados, funcionem com contextos de execução próprios onde são armazenadas as suas variáveis (argumentos, variáveis locais e resultado da função).
- Podemos fazer uma analogia com a instanciação de objectos, com a diferença de o contexto de existência das variáveis do método se circunscrever ao período de execução do método.
  - As variáveis são criadas quando o método inicia a sua execução, e descartadas quando termina.
- A implementação mais eficiente para este fim assenta numa estrutura de dados composta designada por *Pilha* (*stack*), que se caracteriza por uma gestão do tipo *LIFO* (*Last In First Out*);

07.4

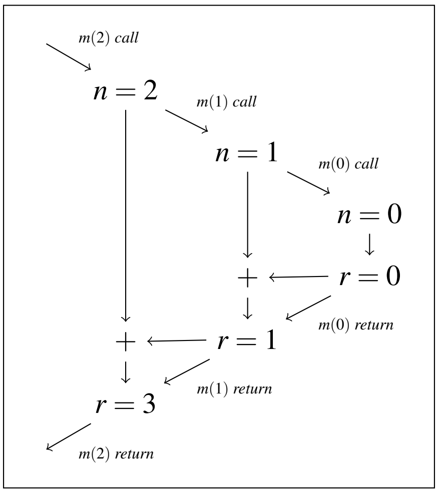
Exemplo

- Vejamos, como exemplo, a seguinte função recursiva $m(n)$, que devolve o somatório dos números de 0 a n:

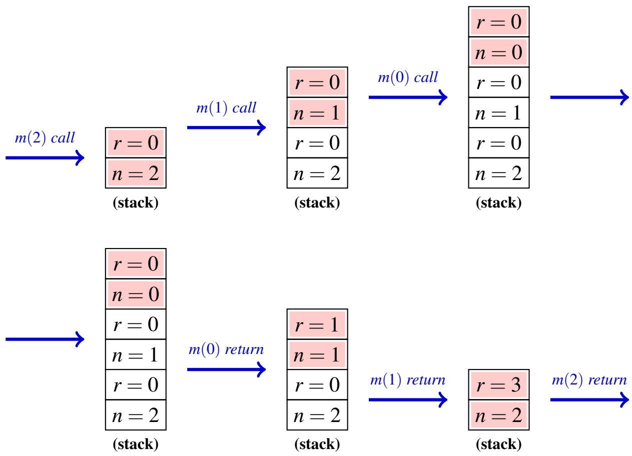
```
static int m(int n)
{
   assert n >= 0;

   int r = 0;
   out.println("n = "+n);
   if (n > 0)
      r = n + m(n-1);
   out.println("r = "+r);
   return r;
}
```

07.5

Exemplo: execução de $m(2)$



07.6



07.7

Note que esta última apresentação da execução de `m(2)` é uma simplificação da implementação real com pilha (na qual, para além da variável local `r`, para cada execução a pilha contém também o resultado da função).

2 Conversão entre recursão e iteração

2.1 Iteração para recursão

- Como já foi referido, um algoritmo recursivo tem sempre uma versão iterativa e vice-versa;
- Um algoritmo genérico que permite converter um ciclo (estruturado!) para uma função recursiva é o seguinte:

Implementação Iterativa

```
for( INIT; COND; INC) {
   BODY
}
```

Implementação Recursiva

```
INIT
loopEquiv(args)

...

static void loopEquiv(args decl) {
   if (COND) {
      BODY
      INC
      loopEquiv(args);
   }
}
```

- Os argumentos a definir para a função recursiva dependem somente das variáveis utilizadas no ciclo;
- Argumentos ou variáveis locais necessitam de ser passados para a função. 07.8

Note que esta conversão pressupõe que o ciclo é estruturado. Ou seja, nele não existem instruções do tipo “salto” (`break`, `continue` ou `return`).

Iteração para recursão: exemplo

Implementação Iterativa

```
// int[] arr
for(int i=0; i<arr.length; i++)
   out.println(arr[i]);
```

Implementação Recursiva

```
int i = 0;
loopEquiv(arr, i);

...

static void loopEquiv(int[] arr,int i) {
   if (i < arr.length) {
      out.println(arr[i]);
      i++;
      loopEquiv(arr, i);
   }
}
```

- Podemos melhorar esta implementação substituindo o incremento de `i` pela passagem de `i+1` para a função. 07.9

2.2 Recursão para iteração

- A conversão de algoritmos recursivos para ciclos (estruturados) é, em geral, bem mais complexa do que a transformação inversa;
- Um algoritmo geral para fazer essa conversão faz uso de uma *pilha* para armazenar os contextos de execução da função recursiva (composto pelos argumentos, variáveis locais e resultado da função) e implementar as chamadas das funções por instruções (não estruturadas) do tipo *salto* (*goto*);
- No entanto o preço a pagar pode ser bem elevado em termos de legibilidade e até mesmo de correcção do algoritmo;

- Alguns tipos em particular de recursividade, como é o caso da recursão do tipo *cauda* (*tail recursion*) prestam-se a optimizações interessantes (já que podemos prescindir do armazenamento de algum contexto);
- Esta matéria, no entanto, sai fora do âmbito desta disciplina pelo que não a vamos abordar;

07.10

Recursão para iteração: exemplo

- Certas funções recursivas (como o cálculo dos números de Fibonacci ou o factorial) são, no entanto, facilmente convertidas em ciclos:
  - Basta para tal, fazer a iteração desde o(s) caso(s) limite até ao valor desejado, e ir armazenando os valores calculados num array;
  - As invocações recursivas são assim imediatamente convertíveis em acessos ao array.

Implementação Recursiva

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

   int res = 1;

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

   return res;
}
```

Implementação Iterativa (com array)

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

   int[] arr = new int[n+1];
   for(int i = 0; i <= n; i++) {
      if (i < 2) // casos limite
         arr[i] = 1;
      else
         arr[i] = i * arr[i-1];
   }

   return arr[n];
}
```

07.11

Por vezes, poderá verificar-se não ser necessário armazenar todos os valores anteriores e, nesses casos, poderá ser possível optimizar o algoritmo iterativo para usar menos memória. (Pode fazer isso no exemplo acima.)

3 *Procura com Retrocesso*

Algoritmos de *Procura com Retrocesso (Backtracking)*

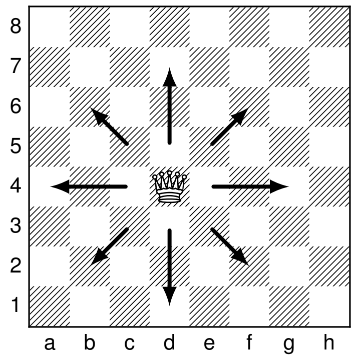
Definição: Método de procura exaustiva de soluções para certos problemas por exploração sucessiva (e sistemática) de diferentes caminhos. Sempre que é atingido um beco sem saída, retrocede (daí o nome) no caminho percorrido até que exista pelo menos um caminho por pesquisar (ou até ao ponto de partida, se não existir nenhuma solução).

- É um método de pesquisa cuja implementação recursiva é bastante simples (e intuitiva);
- Para além dessas vantagens este tipo de algoritmos, quando aplicáveis, ou encontram garantidamente uma solução, ou então garantem a sua não existência.

07.12

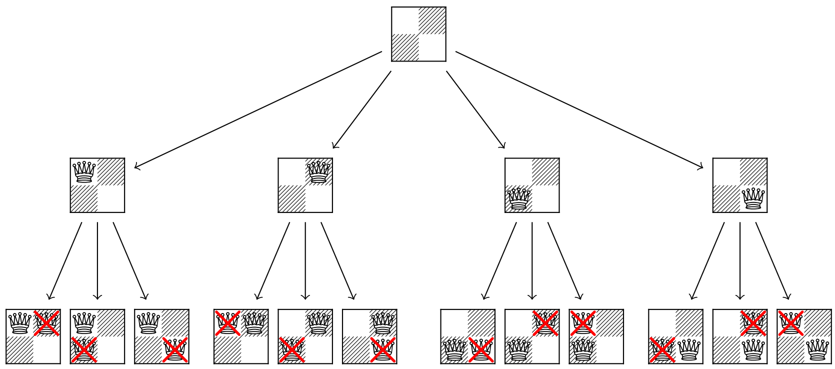
3.1 O Problema das N Rainhas

Problema: Colocar N rainhas num tabuleiro de xadrez (N por N) sem que se ataquem mutuamente.



07.13

Rainhas 2×2: Árvore de Decisão



Para um tabuleiro 2×2, não há solução!

07.14

Rainhas $N\times N$: Estratégia de Resolução

- Para resolver o problema genérico de N rainhas em tabuleiros $N\times N$, podemos combinar uma estratégia pesquisa com retrocesso *backtracking* com uma das características das rainhas:

  A haver solução, só pode existir uma rainha em cada coluna (ou em cada linha).

1. Assim, podemos associar uma rainha a cada coluna, tentando-as colocar coluna a coluna, em posições não atacadas por nenhuma das rainhas já colocadas;
2. Se, numa determinada coluna, chegarmos a um impasse, vamos retrocedendo até uma coluna onde possamos (ainda) re-colocar a respectiva rainha;
3. Em cada coluna vamos percorrer sistematicamente todas as linhas à procura de posições não atacadas.

07.15

Rainhas: Algoritmo Recursivo

- Na forma recursiva, o algoritmo pode colocar-se assim:
- Da esquerda para a direita, colocar rainhas a partir da coluna C (de forma que não sejam atacadas pelas C rainhas já colocadas à sua esquerda):
  1. Começando na primeira linha ($L = 0$) da coluna C:
  2. Se a posição (L,C) não é atacada pelas C rainhas à esquerda, então
     (a) Colocar (tentativamente) uma rainha em (L,C); e
     (b) Colocar rainhas a partir da coluna $C+1$;
     (c) Se conseguimos, terminar indicando sucesso, senão,

(d) Retirar a rainha que tínhamos colocado em (L,C).

3. Avançar para linha seguinte e repetir a partir do passo 2, mas
4. Se já não há mais linhas, terminar indicando insucesso.

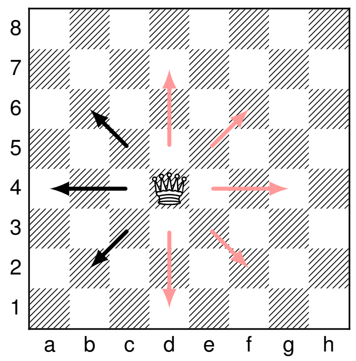
07.16

Repare que:

- No passo 3, sabemos que não há nenhuma rainha colocada em (L,C), ou porque essa era uma posição atacada ou porque retirámos a rainha que lá tínhamos colocado.
- Saber se uma posição (L,C) é atacada (passo 2) é um subproblema que resolveremos já a seguir.
- O passo 2b. é a chamada recursiva.
  É fácil ver que garante a condição de variabilidadei ($C+1 \neq C$).
- Falta o caso limite!
  Para um tabuleiro de N colunas, o caso limite será quando o número de rainhas já colocadas for igual a N ($C = N$). Nesse caso, basta terminar indicando sucesso.
- A convergência é fácil de demonstrar porque a sucessão $C, C+1, \ldots$ acaba por atingir N, desde que $C \leq N$ no início.

O Problema da Posição Não Atacada

- Uma vez que há apenas uma rainha por coluna,das 8 possíveis direcções de ataque de uma rainha, só é necessário verificar 3:



- É necessário escolher uma representação para registar a posição das rainhas.
- A representação mais directa será uma matriz booleana de N por N (o valor `true` indica a presença da rainha).

07.17

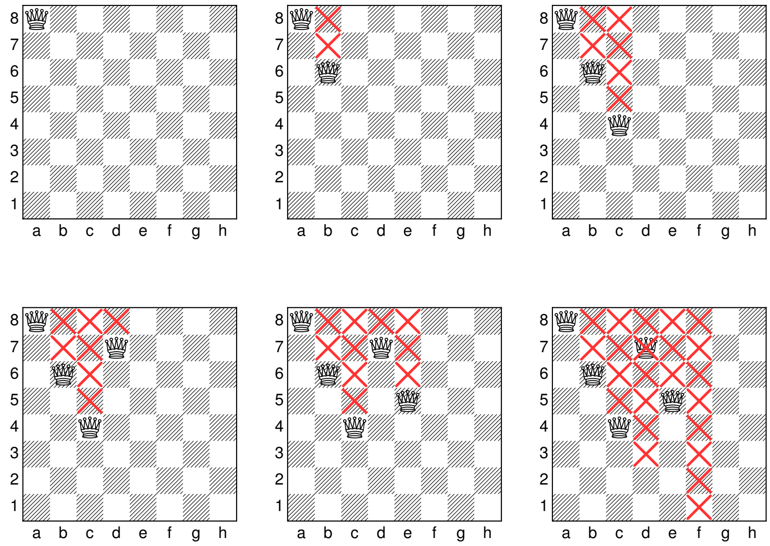
O Problema da Posição Não Atacada: Implementação Possível

A posiçao `(lin,col)` é atacada por rainhas das colunas à esquerda?

```
static boolean attackedPosition(boolean[][] queensPosition, int lin, int col)
{
   assert queensPosition != null;
   assert lin >= 0 && lin < queensPosition.length;
   assert col >= 0 && col < queensPosition.length;

   boolean result = false;
   int n = queensPosition.length;

   for(int c = 0;!result && c < col;c++)
      result = queensPosition[lin][c] ||
               (lin-col+c >= 0 && queensPosition[lin-col+c][c]) ||
               (lin+col-c < n && queensPosition[lin+col-c][c]);

   return result;
}
```

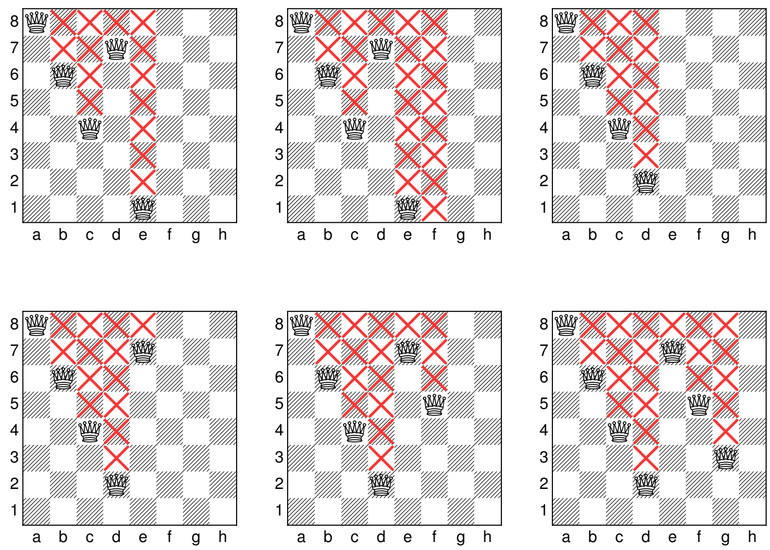
07.18

Rainhas 8×8: Iterações (1)



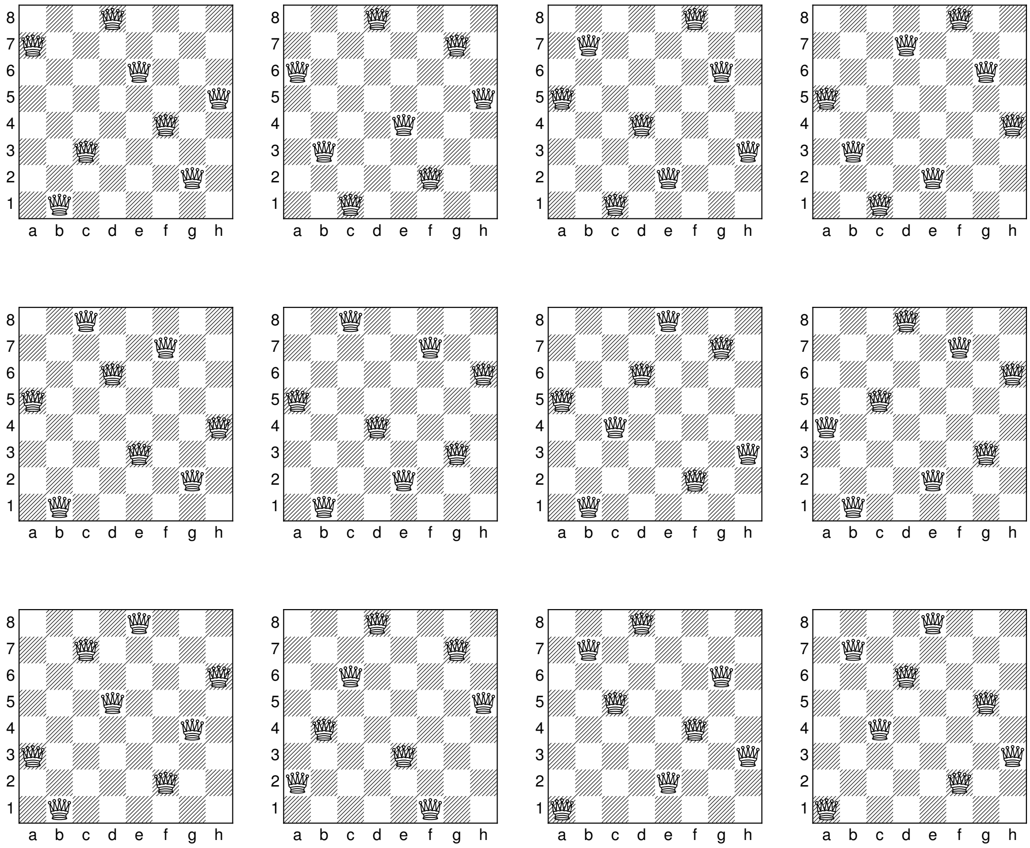
07.19

Rainhas 8×8: Iterações (2)



07.20

Nos tabuleiros normais de xadrez (8×8) existem 92 soluções para este problema. Eliminando variantes resultantes de rotações simples do tabuleiro, ficamos com as seguintes 12 soluções:



N Rainhas: Implementação Possível

Coloca rainhas a partir da coluna `col` em posições não atacadas. Resultado indica se foi encontrada solução e, nesse caso, o tabuleiro `queensPosition` terá a solução encontrada.

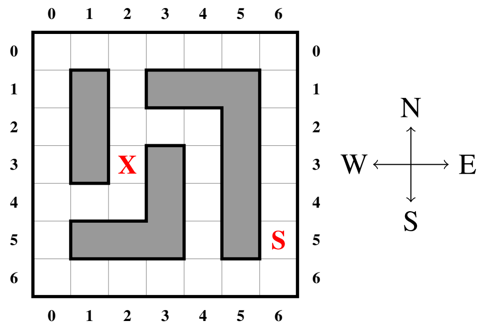
```
static boolean placeQueens(boolean[][] queensPosition, int col)
{
   assert queensPosition != null;
   assert col >= 0 && col <= queensPosition.length;

   int n = queensPosition.length;

   boolean result = (col == n);

   for (int lin = 0;!result && lin < n;lin++)
   {
      if (!attackedPosition(queensPosition, lin, col))
      {
         queensPosition[lin][col] = true;
         result = placeQueens(queensPosition, col+1);
         if (!result)
            queensPosition[lin][col] = false;
      }
   }

   return result;
}
```

07.21

3.2 O Problema dos Labirintos

Labirintos

- Considere o seguinte problema: dado um labirinto onde nos podemos movimentar na horizontal e na vertical (Norte, Sul, Este e Oeste), como descobrir um caminho entre dois pontos (de **S** para **X**)?



07.22

Teste de algumas soluções iterativas

Para tentar resolver este problema podemos experimentar as seguintes possibilidades:

1. Percorrer o labirinto com uma ordem pré-determinada das direcções (por exemplo: Norte, Oeste, Sul e Este):
   - A aplicar directamente este algoritmo, no exemplo da figura, teríamos o ciclo interminável:

   `(5,6)->...->(0,6)->...->(0,0)->...->`
   `(6,0)->...->(6,6)->...->(0,6)->...`

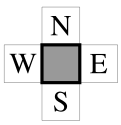
2. Para evitar o ciclo infinito, podemos ir marcando as células já visitadas (não as repetindo):

   `(5,6)->...->(0,6)->...->(0,0)->...->`
   `(6,0)->...->(6,6)->`*deadlock*

07.23

Labirinto: solução recursiva

- Uma vez que a procura de um caminho num labirinto se presta a ser resolvida de uma forma repetitiva e sistemática, porque não tentar uma solução procura com retrocesso (*backtracking*)?
- Para tal vamos reduzir o problema do labirinto a uma função que procura o ponto de chegada numa dada posição, e, caso lá não esteja, faz o mesmo nas quatro localizações vizinhas:



07.24

Labirinto: solução baseada em *procura com retrocesso*

```java
public static boolean searchPath(int lin, int col)
{
   boolean result = false;
   if (isInside(lin, col) && isRoad(lin, col))
   {
      if (isGoal(lin, col))
         result = true;
      else if (freePosition(lin, col))
      {
         markPosition(lin, col);
         if (searchPath (lin-1, col))       // Search North
            result = true;
         else if (searchPath (lin, col+1))  // Search East
            result = true;
         else if (searchPath (lin, col-1))  // Search West
            result = true;
         else if (searchPath (lin+1, col))  // Search South
            result = true;
         else
            unmarkPosition(lin, col);
      }
   }
   return result;
}
```

07.25