# INF 112: Programação II

Aula 03

➔ **Continuação de recursividade: Estratégia *Backtracking***

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

# Backtracking

➔ Certos problemas, para serem resolvidos de forma exata, demandam algoritmos que, para uma data entrada, verifiquem todas as possibilidades de saída para descobrir aquela que é a resposta procurada para a entrada fornecida. Ex.: problema do caixeiro viajante.

➔ Nestes casos, pode-se fazer uma busca exaustiva gerando sempre todas as possibilidades de resposta para escolher aquela que se procura ou então utilizar alguma forma sistemática mais "esperta" de gerar as possíveis respostas, de modo que, na maioria das vezes, não se precise enumerar todas as possibilidades.

➔ ***Backtracking*** é uma estratégia para gerar respostas a um dado problema, mas sem que, em geral, haja a necessidade de produzir todas as possibilidades para a construção dessas respostas.

# Backtracking

➔ A principal ideia da estratégia *backtracking* é construir soluções de forma incremental, ou seja, um componente da solução por vez, de forma que cada solução parcial obtida é avaliada para definir se a construção até uma solução completa deve ser continuada ou interrompida para se tentar outro caminho.

## Backtracking

➔ Portanto, se uma solução construída parcialmente não estiver violando as restrições do problema e, desta forma, puder ser continuada, isto é feito tomando-se a primeira opção possível restante para o próximo componente da solução parcial.

➔ Caso os componentes candidatos a serem incorporados na solução parcial não sejam válidos, ou seja, violem a restrição do problema, então interrompe-se o desenvolvimento da solução parcial, isto é, não há a necessidade de avaliar qualquer outro componente dali para frente. Neste caso, o algoritmo realiza um *backtracking*, ou seja, recua ao componente anterior da solução parcial para tentar uma construção alternativa, isto é, utilizando outras componentes.

# Backtracking

- É conveniente implementar este tipo de estratégia através da construção de uma árvore das opções que são selecionadas. Esta árvore é chamada de **árvore do espaço de estados**.

- A raiz desta árvore representa o **estado inicial**, ou seja, a situação antes da busca por uma solução ser iniciada.

- Os nodos do primeiro nível da árvore representam as opções selecionadas para o primeiro componente de uma solução, os nodos do segundo nível representam as opções para o segundo componente, etc.

# Backtracking

➔ Um nodo da árvore é considerado **promissor** se corresponder a uma solução parcial que ainda pode se transformar em solução completa; caso contrário, o nodo é considerado **não-promissor**.

➔ Uma folha da árvore pode representar uma das seguintes situações: um nodo não-promissor, representando fim da linha daquela tentativa de solução, ou então uma solução completa encontrada pelo algoritmo.

# Backtracking

➔ Na estratégia *backtracking*, a árvore de estados é normalmente gerada usando **busca em profundidade** (*depth-first search*).

➔ Assim, se o nodo atual é promissor, um nodo filho é gerado representando a adição da primeira opção válida das que restam como possíveis de serem adicionadas como mais um componente da solução parcial no momento. O processo é então continuado a partir deste nodo filho gerado.

➔ Se o nodo corrente for não-promissor, o algoritmo recua (*backtracks*) para o nodo pai (então a componente adicionada anteriormente representada pelo novo nodo é descartada – nodo vira folha fim-da-linha) para considerar a próxima opção possível para um novo componente (geração de um nodo irmão do que virou folha).

## Backtracking

➔ Veja que se não houver mais opções de adição de componente para a solução parcial corrente, então há um recuo de mais um nível acima.

➔ Por fim, se o algoritmo alcançar uma solução completa para o problema, então a execução é interrompida, caso o objetivo seja encontrar somente uma solução, ou é mantida no intuito de gerar outras soluções possíveis.

## Backtracking aplicado às *n*-rainhas

➔ Vejamos um exemplo de aplicação desta estratégia para o problema das ***n*-rainhas**.

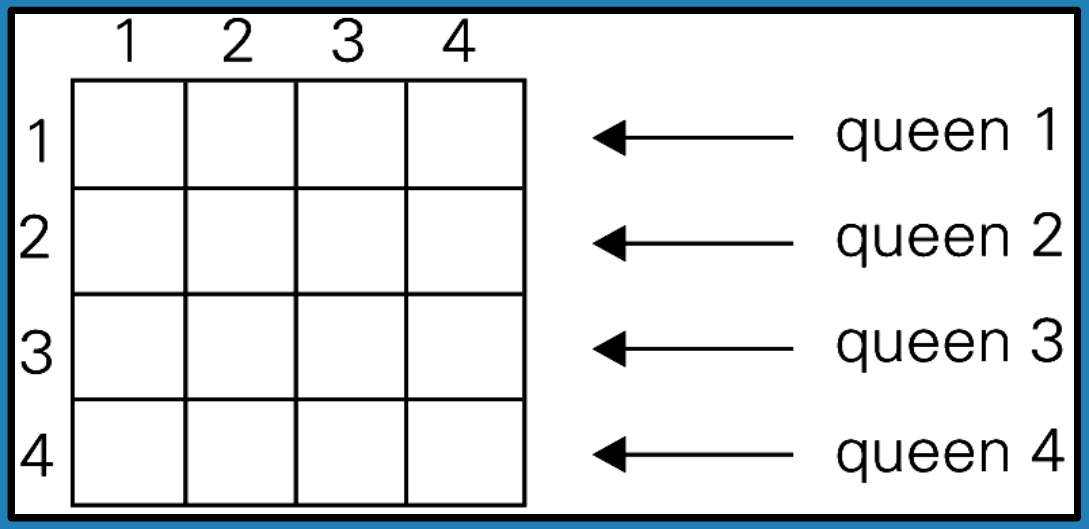
➔ O problema consiste em colocar $n$ rainhas em um tabuleiro de xadrez $n \times n$ de modo que nenhuma rainha possa atacar qualquer outra, ou seja, nenhum par de rainhas pode estar na mesma linha ou na mesma coluna ou ainda na mesma diagonal.

➔ Veja que para $n=1$, o problema tem solução trivial, enquanto que para $n=2$ e $n=3$ não há qualquer solução possível.

➔ Vamos então considerar o problema para $n=4$ e resolvê-lo pela estratégia *backtracking*.

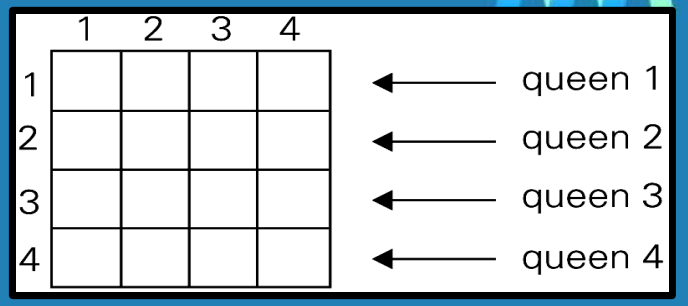
# Backtracking aplicado às *n*-rainhas

➔ Já que cada uma das quatro rainhas precisa ser colocada em sua própria linha, tudo que precisamos fazer é associar uma coluna para cada rainha mostrada no tabuleiro abaixo.



# Backtracking aplicado às *n*-rainhas

- Começamos com o tabuleiro vazio e então colocamos a rainha 1 na primeira posição possível de sua linha, ou seja, na coluna 1 da linha 1.



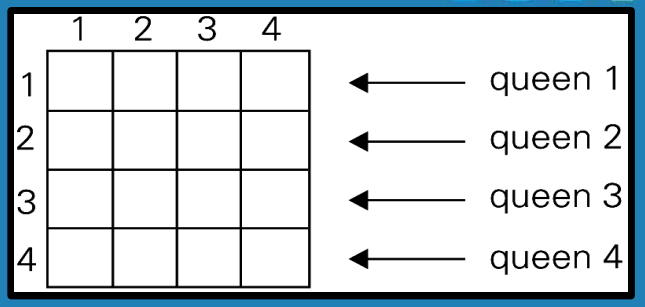

- Então, após tentar sem sucesso as colunas 1 e 2 da linha 2 para a rainha 2, nós a colocamos na primeira posição aceitável que é o quadrado (2,3).

- Quando tentamos colocar a rainha 3, vemos que não há possibilidades para tal, ou seja, a solução parcial para as rainhas 1 e 2 torna-se não-promissora.

- Assim, recuamos ao passo antes da inclusão da rainha 2 e tentamos colocá-la em outro espaço, no caso, o quadrado (2,4).

# Backtracking aplicado às *n*-rainhas

- Agora, a rainha 3 pode ser colocada na posição (3,2). Mas, ao se tentar adicionar a rainha 4, vê-se que isto não é possível, o que mostra que a solução parcial para as rainhas 1, 2 e 3, até o momento, é não promissor.



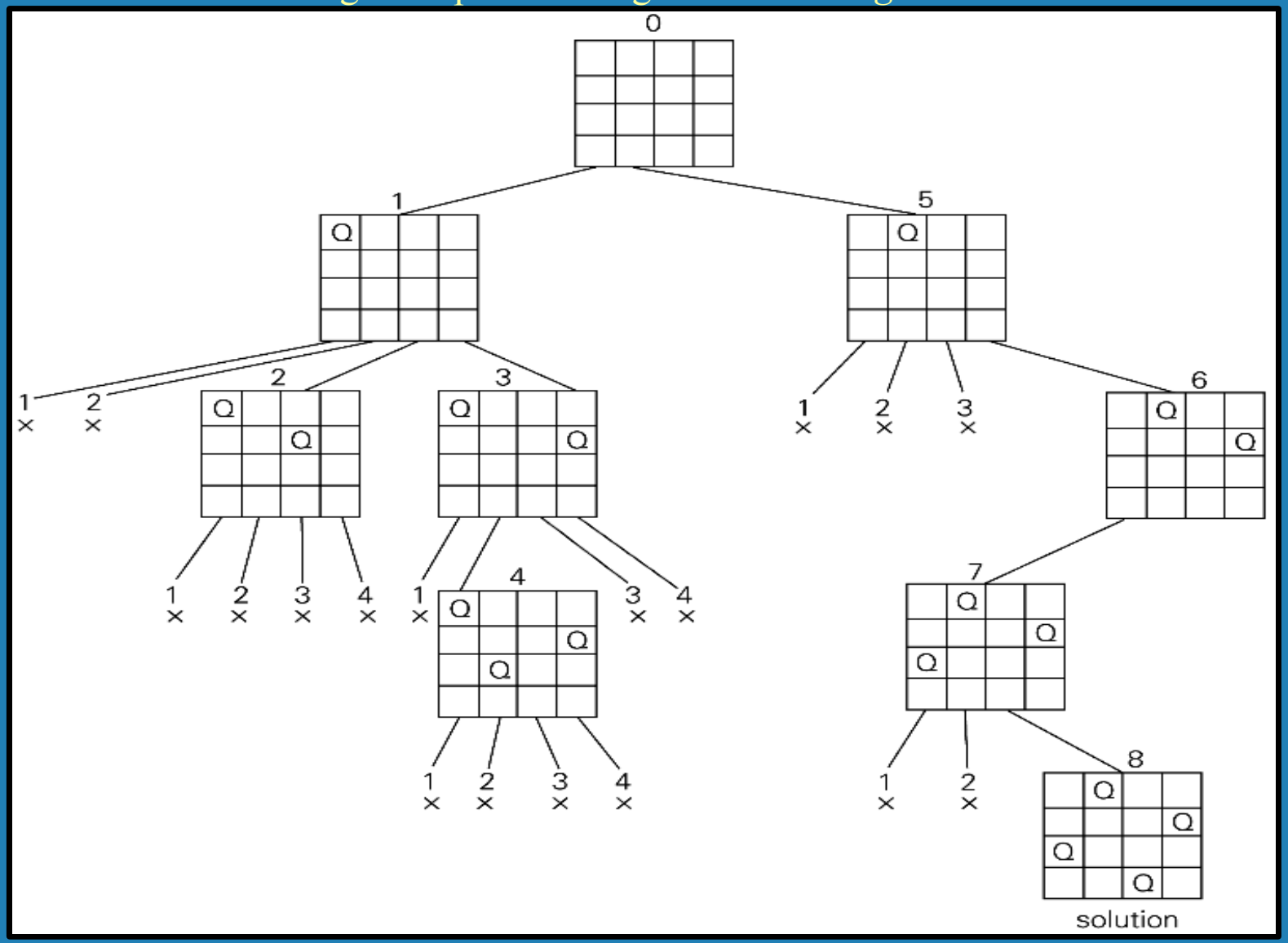

- Deste modo, recuamos vários passos até a situação inicial, pois verifica-se que as soluções parciais construídas em cada passo são não-promissoras. Após voltar para a situação inicial, colocamos a rainha 1 no quadrado (1,2).

- A rainha 2 é então colocada na posição (2,4), a rainha 3 em (3,1) e a rainha 4 em (4,3), chegando-se a uma solução para o problema.

- A árvore do espaço de estados desta busca é mostrada a seguir.

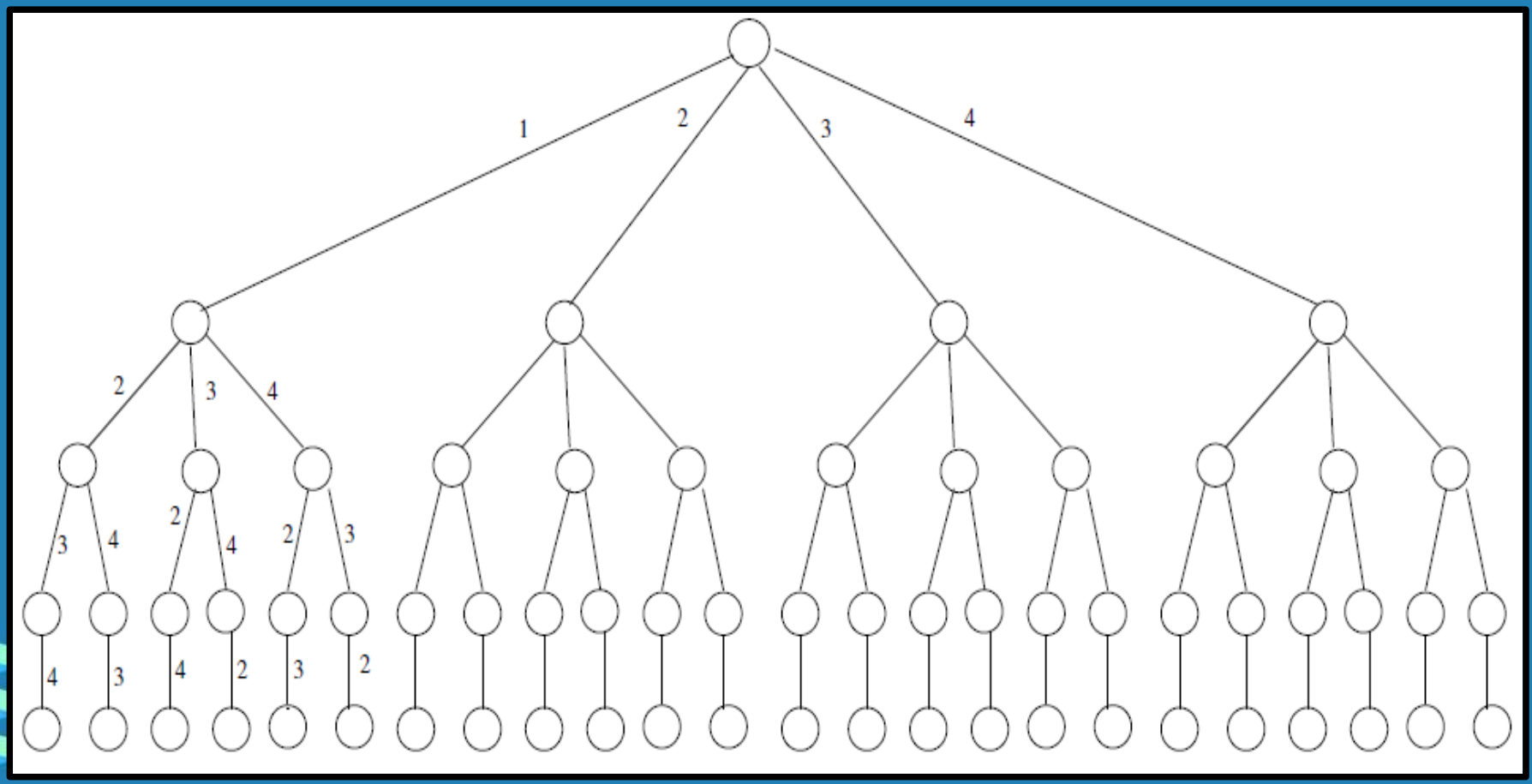
# Backtracking aplicado às *n*-rainhas

- Árvore de estados gerada pela estratégia *backtracking*:



# Backtracking aplicado às *n*-rainhas

➔ Note que se formos gerar todas as possibilidades, portanto sem nenhuma estratégia que permita eliminar opções, a árvore de estados seria completa, o que significa um número muito maior de estados (mais esforço computacional para se chegar à mesma resposta).



# Backtracking – procedimento genérico

➔ Para implementar tal estratégia em computador, podemos pensar na seguinte estrutura:

- Uma solução é modelada como um vetor $(a_1, a_2, ...., a_n)$;
- Cada elemento $a_i$ representa uma parte na solução:
  - ✓ Uma jogada de uma sequência;
  - ✓ Uma parte de um caminho;
  - ✓ Um subconjunto de elementos, etc.
- A cada passo, tem-se uma solução parcial com $k$ elementos:

  $(a_1, a_2, ...., a_k)$.
- Testa-se se a solução desejada foi encontrada;
- Senão, tenta-se estender a solução adicionando-se mais um elemento ao vetor $(a_1, a_2, ...., a_k , a_{k+1})$.

➔ Com isto pode-se ter a seguinte rotina genérica para a estratégia *backtracking*...

# Backtracking – procedimento genérico

```
void backtrack(int a[], int k, Data info) {
  int c[MAXCANDIDATOS]; // Candidatos a serem adicionados
  int nCandidatos; // Contador de candidatos
  int i;
  if ( ehUmaSolucao(a,k,info) )
    processaSolucao(a,k,info);
  else {
    k = k+1;
    enumeraCandidatos(a,k,info,c,nCandidatos);
    for (i=0; i<nCandidatos; i++) {
      a[k-1] = c[i];
      backtrack(a,k,info);
      if (acabou) return; // Para ter possibilidade de
    }                     // terminar mais cedo.
  }
}
```

## Backtracking – procedimento genérico

- `ehUmaSolucao(a,k,info)`: testa se os primeiros *k* elementos do *array a* formam uma solução desejada (*info* permite passar dados do problema específico para a rotina);

- `enumeraCandidatos(a,k,info,c,nCandidatos)`: preenche o *array c* com o conjunto completo de todos os possíveis candidatos para a posição *k* do *array a*, dado o conteúdo das *k*-1 primeiras posições. A variável `nCandidatos` é passada por referência;

- `processaSolucao(a,k,info)`: esta rotina faz coisas do tipo imprimir, contar ou realizar algum processamento desejado uma vez que a solução tenha sido encontrada.

# Backtracking - implementação para as *n*-rainhas

- Esta é uma rotina genérica para a estratégia *backtracking*. Para cada caso particular deve-se adaptá-la de acordo, incluindo a retirada de partes desnecessárias.
- No caso do problema das *n*-rainhas, poderíamos ter a seguinte implementação, seguindo a rotina mostrada (note que neste caso não estamos mostrando as soluções mas só contando-as):

```
bool acabou = false; // Variaveis
int contaSolucoes = 0; // globais. Mas podem virar locais.
#define NRAINHAS 4 //Se quiser aumentar é só alterar aqui.

void processaSolucao() { // Parâmetros desnecessários
  contaSolucoes++; // Só conta as soluções.
}

bool ehUmaSolucao(int k, int info) { //arranjo desnecessário
  return (k == info);
}
```

# Backtracking - implementação para as *n*-rainhas

```
void enumeraCandidatos(int a[],int k,int n,int c[],int&
nCandidatos) {
  int i,j; bool movLegal; nCandidatos = 0;
  for (i=1; i<=n; i++) { // Para n possíveis colunas.
    movLegal = true;
    for (j=1; j<k; j++) { // Para k-1 rainhas já colocadas.
      if (i == a[j-1]) // Ataque pela coluna
        { movLegal = false; break; }
      if (abs(k-j) == abs(i-a[j-1])) // Ataque diagonal
        { movLegal = false; break; }
    }
    if (movLegal == true) {
      c[nCandidatos] = i;
      nCandidatos++;
    }
  }
}
```

# Backtracking - implementação para as *n*-rainhas

```
void backtrack(int a[], int k, int info) {
  int c[NRAINHAS]; //Posicoes candidatas para a próx. rainha.
  int nCandidatos; // Numero de candidatos gerados.
  int i;
  if ( ehUmaSolucao(k,info) )
    processaSolucao();
  else {
    k++;
    enumeraCandidatos(a,k,info,c,nCandidatos);
    for (i=0; i<nCandidatos; i++) {
      a[k-1] = c[i];
      backtrack(a,k,info);
      if (acabou) return; // Para ter a possibilidade
    }                     // de terminar mais cedo.
  }
}
```

# Backtracking - implementação para as *n*-rainhas

- Um pequeno programinha para utilização da função `backtrack`.

```
int main() {
  int vet[NRAINHAS];
  backtrack( vet, 0, NRAINHAS );
  cout << contaSolucoes << endl;
}
```

- Se quiser imprimir as soluções (não só contá-las), basta alterar a função `processaSolucao` para que a mesma receba o arranjo que representa a solução e imprima esta última na forma de um tabuleiro.

## Exercício

1) Faça o rastreio detalhado do programa do *slide* anterior.

2) Modifique a função `processaSolucao` mostrada para o problema das *n*-rainhas de modo que não só conte a solução encontrada mas também a imprima.

3) Faça a modificação necessária para que o procedimento mostrado para o problema das *n*-rainhas seja interrompido ao se encontrar a primeira solução.