# Estrutura de Dados

## Aula 11 : Recursividade

**Prof. MSc. Fausto Sampaio**

fausto.sampaio.unifanor.edu.br

Centro Universitário UniFanor - Wyden

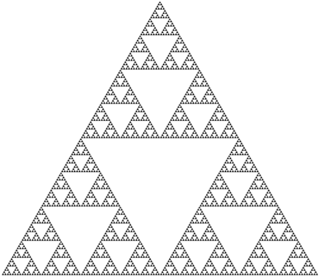
4 de dezembro de 2019

# Sumário

# Recursividade

# Introdução

- **Recursividade** no geral é um termo usado de maneira mais geral para descrever o processo de repetição de um objeto de um jeito similar ao que já fora mostrado.
- Um bom exemplo disso são as imagens repetidas que aparecem quando dois espelhos são apontados um para o outro;



# Introdução

- Na linguagem C, uma função pode chamar outra função;
- A função main() pode chamar qualquer função, seja ela da biblioteca da linguagem (como a função printf()) ou definida pelo programador (função imprime());

```
#include <stdio.h>
#include <stdlib.h>
void imprime(int n){
  int i;
  for (i=1; i<=n; i++)
      printf(“Linha %d \n”,i);
}

int main(){
  imprime(5);
  printf(“Fim do programa!\n”);

  system(“pause”);
  return 0;
}
```

# Recursividade

- Um abordagem onde um procedimento ou função chama a si próprio;
- Processo de repetir alguma coisa de maneira similar;
- Uma função assim é chamada de **função recursiva**;

A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.

# Esvaziar vaso de flores

- Problema: Como esvaziar um vaso contendo três flores?

Para esvaziar um vaso contendo três flores, primeiro verificamos se o vaso está vazio. Se o vaso não está vazio, tiramos uma flor. Temos agora que esvaziar o vaso contendo duas flores.

# Esvaziar vaso de flores

- Problema: Como esvaziar um vaso contendo duas flores?

Para esvaziar um vaso contendo duas flores, primeiro verificamos se o vaso está vazio. Se o vaso não está vazio, tiramos uma flor. Temos agora que esvaziar o vaso contendo uma flor.

# Esvaziar vaso de flores

- Problema: Como esvaziar um vaso contendo uma flor?

Para esvaziar um vaso contendo uma flor, primeiro verificamos se o vaso está vazio. Se o vaso não está vazio, tiramos uma flor. Temos agora que esvaziar o vaso contendo zero flores.

# Esvaziar vaso de flores

- Problema: Como esvaziar um vaso contendo zero flores?

Para esvaziar um vaso contendo uma flor, primeiro verificamos se o vaso está vazio. Como ele já está vazio, o processo termina.

# Esvaziar vaso de flores

## Problema

Como esvaziar um vaso contendo $N$ flores?

## Solução

Para esvaziar um vaso contendo $N$ flores, primeiro verificamos se o vaso está vazio. Se o vaso não está vazio, tiramos uma flor. Temos agora que esvaziar um vaso contendo $N - 1$ flores.

```
void esvaziarVaso(int flores){
  if(flores > 0){
      //remove uma flor
      esvaziarVaso(flores - 1);
  }
}
```

# Fatorial

- **Problema:** Como calcular o fatorial de 4 (definido como 4!)?
- **Solução:** Multiplica-se o número 4 pelo fatorial de 3 (definido como 3!).
- Nesse ponto, já é possível perceber que esse problema é muito semelhante ao do vaso de flores.
- Generalizando esse processo, temos que o fatorial de N é igual a N multiplicado pelo fatorial de (N − 1), ou seja, N! = N * (N − 1)!.
- No caso do vaso de flores, o processo termina quando não há mais flores no vaso.
- No caso do fatorial, o processa termina quando atingimos o número zero. Nesse caso, o valor do fatorial de 0 (0!) é definido como igual a 1.
- **Definição Matemática**:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n > 0 \end{cases}$$

# Fatorial

**Com recursão**

```c
int fatorial (int n){
 if(n == 0)
  return 1;
 else
  return n*fatorial(n-1);
}
```

**Sem recursão**

```c
int fatorial (int n){
  if(n == 0)
   return 1;
  else {
   int i, f = 1;
   for (i=2; i <= n;i++)
      f = f * i;
   return f;
  }
}
```

# Como funciona a recursividade

- **Dividir e Conquistar**:
  - Divide-se um problema maior em um conjunto de problemas menores.
  - Esses problemas menores são então resolvidos de forma independente.
  - As soluções dos problemas menores são combinadas para gerar a solução final.

# Como funciona a recursividade

- **Exemplo**: cálculo do fatorial.
- O fatorial de um número N é o produto de todos os números inteiros entre N até 1.
- Por exemplo, o fatorial de 4 é 4 * 3 * 2 * 1.
- Aplicando a ideia da recursão, temos que:
  - O fatorial de 4 é definido em função do fatorial de 3.
  - O fatorial de 3 é definido em função do fatorial de 2.
  - O fatorial de 2 é definido em função do fatorial de 1.
  - O fatorial de 1 é definido em função do fatorial de 0.
  - O fatorial de 0 é definido como igual a 1 (**caso-base** ou **condição de parada**.

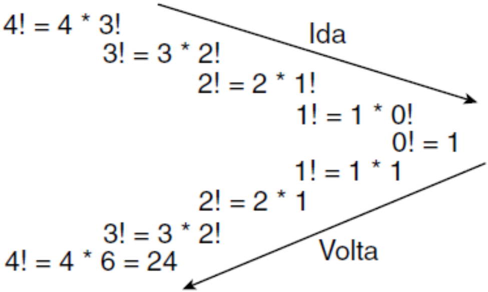
Quando a função recursiva chega ao seu **caso-base**, ela para.

# Como funciona a recursividade

- **Caminho de ida da recursão**: Etapa do cálculo onde as chamadas da função são executadas até chegar ao **caso-base**.
- **Caso-base**: Condição de parada para não chamar mais a função.
- **Caminho de volta da recursão**: Consiste em fazer o caminho inverso devolvendo o valor obtido para quem fez aquela chamada da função, e assim por diante, até chegarmos à primeira chamada da função.

Saber identificar o caso-base, o caminho de ida da recursão e o caminho de volta da recursão torna a construção de uma função recursiva bastante simples.

# Como funciona a recursividade

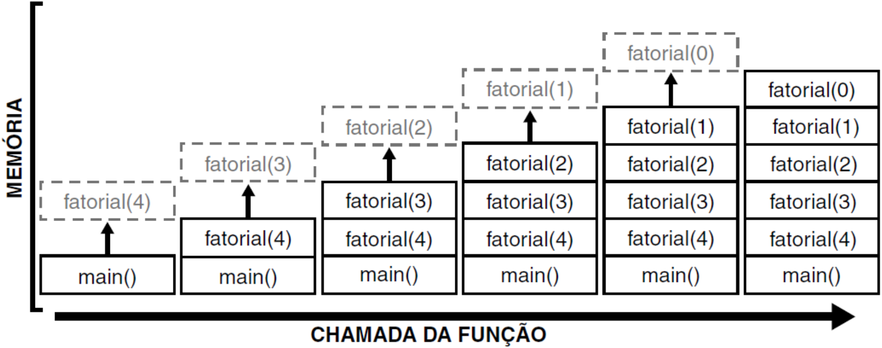



| | |
|---|---|
| `4! = 4 * 3!` | |
| `3! = 3 * 2!` | O contexto de cada chamada é empilhado |
| `2! = 2 * 1!` | |
| `1! = 1 * 0!` | |
| `0! = 1` | Condição de parada |
| `1! = 1` | |
| `2! = 2` | O contexto de cada chamada é desempilhado |
| `3! = 6` | |
| `4! = 24` | |

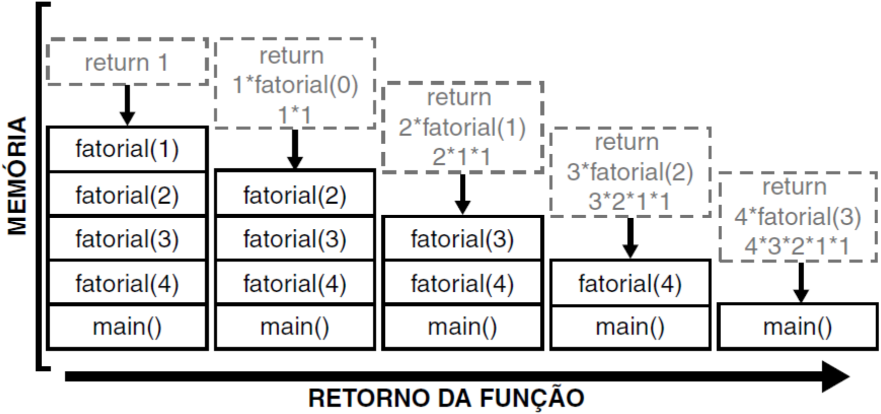
# Como funciona a recursividade

```c
#include <stdio.h>
#include <stdlib.h>
int fatorial (int n){
  if(n == 0)
      return 1;
  else
      return n*fatorial(n-1);
}
int main(){
  int x;
  x = fatorial(4);
  printf("4! = %d\n",x);
  system("pause");
  return 0;
}
```

# Como funciona a recursividade



# Como funciona a recursividade



# Cuidados na implementação da recursividade

- Em geral, as formas recursivas dos algoritmos são consideradas "mais enxutas" e "mais elegantes" do que suas formas iterativas.Isso facilita a interpretação do código.
- Porém, esses algoritmos apresentam maior dificuldade na detecção de erros e podem ser ineficientes.

**Com recursão**

```c
int fatorial (int n){
 if(n == 0)
  return 1;
 else
  return n*fatorial(n-1);
}
```

**Sem recursão**

```c
int fatorial (int n){
   if(n == 0)
    return 1;
   else {
    int i, f = 1;
    for (i=2; i <= n;i++)
       f = f * i;
    return f;
   }
}
```

# Cuidados na implementação da recursividade

## Cuidado

Em funções recursivas, duas coisas devem ficar bem estabelecidas: o **critério de parada** e o **parâmetro da chamada recursiva**.

- **Critério de parada**: determina quando a função deve parar de chamar a si mesma. Se ele não existir, a função continuará executando até esgotar a memória do computador.
  No cálculo de fatorial, o critério de parada ocorre quando tentamos calcular o fatorial de zero: $0! = 1$.
- **Parâmetro da chamada recursiva**: quando chamamos a função dentro dela mesma, devemos sempre mudar o valor do parâmetro passado, de forma que a recursão chegue a um término. Se o valor do parâmetro for sempre o mesmo, a função continuará executando até esgotar a memória do computador.
  No cálculo de fatorial, a mudança no parâmetro da chamada recursiva ocorre quando definimos o fatorial de N em termos no fatorial de $(N - 1)$:
  $N! = N * (N - 1)!$.

# Cuidados na implementação da recursividade

## Atenção

Algoritmos recursivos tendem a necessitar de mais tempo e/ou espaço do que algoritmos iterativos.

## Nota

Sempre que chamamos uma função, é necessário espaço de memória para armazenar os parâmetros, variáveis locais e endereço de retorno da função.
Em uma função recursiva, essas informações são armazenadas para cada chamada da recursão, sendo, portanto, a memória necessária para armazená-las proporcional ao número de chamadas da recursão. Por exemplo, para calcular o fatorial do número 4 são necessárias cinco chamadas da função fatorial.
Além disso, todas essas tarefas de alocar e liberar memória, copiar informações etc. envolvem tempo computacional, de modo que uma função recursiva gasta mais tempo que sua versão iterativa (sem recursão).

# Como funciona a recursividade

```
int fatorial (int n){
  if(n == 0) //critério de parada
      return 1;
  else //parâmetro do fatorial sempre muda
      return n*fatorial(n - 1);
}
```

# Algoritmos Recursivos

# Sequência de Fibonacci

- A seqüência $[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 89, ...]$ é conhecida como seqüência ou série de Fibonacci e tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem seguila.

## Definição Matemática

$$F(n) = \begin{cases} 0, & \text{se n = 0} \\ 1, & \text{se n = 1} \\ F(n-1) + F(n-2), & \text{outros casos} \end{cases}$$

# Sequência de Fibonacci - Implementação

**Com recursão**

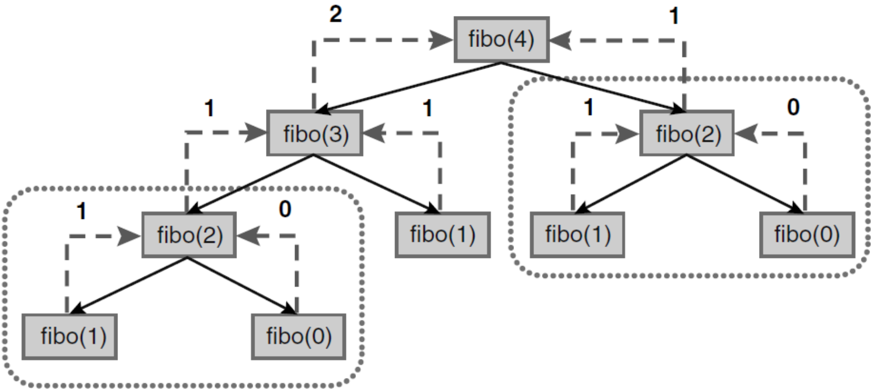
```c
int fibo(int n){
  if(n == 0 || n == 1)
     return n;
  else
     return fibo(n-1) + fibo(n-2);
}
```

**Sem recursão**

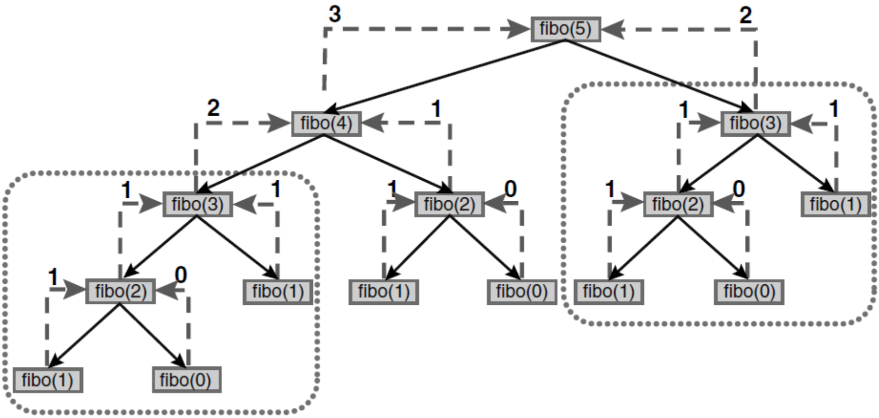
```c
int fibo(int n){
  int i,t,c,a=0, b=1;
  for(i=0;i<n;i++){
      c = a + b;
      a = b;
      b = c;
  }
  return a;
}
```

# Sequência de Fibonacci - Desempenho

- Duas chamadas a si mesma, ou seja, duas chamadas recursivas.
- Desperdício de tempo e espaço.
- No cálculo de fibo(4) teremos duas chamadas para fibo(2).

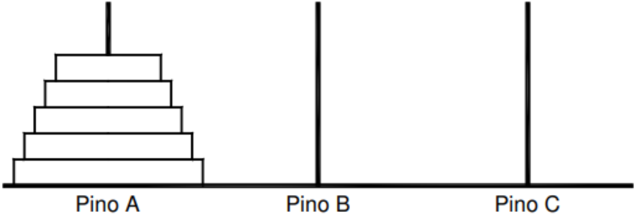


- No cálculo de fibo(5):



# Problema da Torre de Hanói

- Publicado em 1883 pelo matemático francês Edouard Lucas;
- Consiste em transferir, com o menor número de movimentos, a torre composta por $N$ discos do pino **A** (origem) para o pino **C** (destino), utilizando o pino **B** como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.



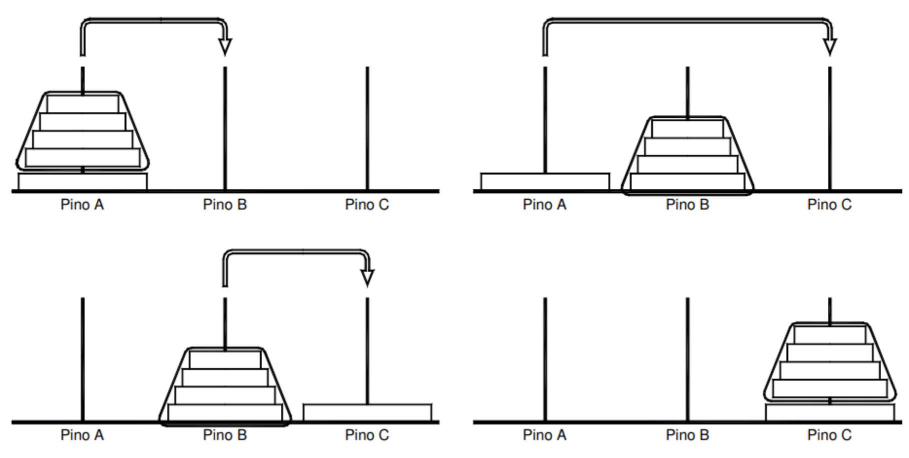
# Problema da Torre de Hanói

**Solução**

Transferir a torre com $N-1$ discos de **A** para **B**, mover o maior disco de **A** para **C** e transferir a torre com $N-1$ de **B** para **C**.
Embora não seja possível transferir a torre com $N-1$ de uma só vez, o problema torna-se mais simples: mover um disco e transferir duas torres com $N-2$ discos. Assim, cada transferência de torre implica em mover um disco e transferir de duas torres com um disco a menos e isso deve ser feito até que torre consista de um único disco.

# Problema da Torre de Hanói

## Solução



# Problema da Torre de Hanói

## Algoritmo

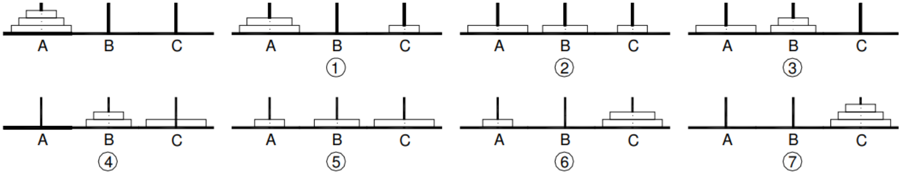
```
procedimento MoveTorre(N : natural; Orig, Dest, Aux : caracter)
início
   se N = 1 então
     MoveDisco(Orig, Dest)  senão
   início
     MoveTorre(N - 1, Orig, Aux, Dest)
     MoveDisco(Orig, Dest)
     MoveTorre(N - 1, Aux, Dest, Orig)
   fim
fim


procedimento MoveDisco(Orig, Dest : caracter)
início
   Escreva("Movimento: ", Orig, " -> ", Dest)
fim
```

# Problema da Torre de Hanói

- Uma chamada a MoveTorre(3, 'A', 'C', 'B') produziria seguinte saída:

```
Movimento: A -> C
Movimento: A -> B

Movimento: C -> B

Movimento: A -> C

Movimento: B -> A

Movimento: B -> C

Movimento: A -> C
```

# Problema da Torre de Hanói




```
MT(3, A, C, B)
 ├MT(2, A, B, C)
 │ ├MT(1, A, C, B)
 │ │ └MD(A, C) .................. A -> C
 │ ├MD(A, B) .................... A -> B
 │ └MT(1, C, B, A)
 │   └MD(C, B) .................. C -> B
 ├MD(A, C) ...................... A -> C
 └MT(2, B, C, A)
   ├MT(1, B, A, C)
   │ └MD(B, A) .................. B -> A
   ├MD(B, C) .................... B -> C
   └MT(1, A, C, B)
     └MD(A, C) .................. A -> C
```

# Problema da Torre de Hanói

- O número mínimo de movimentos para conseguir transferir todos os discos do pino de origem para o pino de destino é $2^n - 1$, sendo $n$ o número de discos.

| **n (número de discos)** | **número de movimentos** |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 7 |
| 4 | 15 |
| 5 | 31 |
| 6 | 63 |
| 8 | 256 |
| 10 | 1.023 |
| 15 | 32.767 |
| 20 | 1.048.575 |
| 30 | 1.073.741.823 |
| 64 | 18.446.744.073.709.551.615 |

# Resumo - Recursividade

# Vantagens

- Os algoritmos recursivos normalmente são mais compactos, mais legíveis e mais fáceis de serem compreendidos.
- Algoritmos para resolver problemas de natureza recursiva são fáceis de serem implementados em linguagens de programação de alto nível.

# Desvantagens

- Por usarem intensivamente a pilha, o que requer alocações e desalocações de memória, os algoritmos recursivos tendem a ser mais lentos que os equivalentes iterativos, porém pode valer a pena sacrificar a eficiência em benefício da clareza.
- Algoritmos recursivos são mais difíceis de serem depurados durante a fase de desenvolvimento.

# Aplicações

- Nem sempre a natureza recursiva do problema garante que um algoritmo recursivo seja a melhor opção para resolvê-lo. O algoritmo recursivo para obter a seqüência de Fibonacci é um ótimo exemplo disso.
- Algoritmos recursivos são aplicados em diversas situações como em:
  - problemas envolvendo manipulações de árvores;
  - analisadores léxicos recursivos de compiladores;
  - problemas que envolvem tentativa e erro ("Backtracking").

# Tentativa e Erro (Backtracking)

# Introdução

- Suponha que você tem que tomar uma série de decisões dentre várias possibilidades, onde:
  - Você não tem informação suficiente para saber o que escolher;
  - Cada decisão leva a um novo conjunto de escolhas;
  - Alguma seqüência de escolhas (possivelmente mais que uma) pode ser a solução para o problema

# Definição

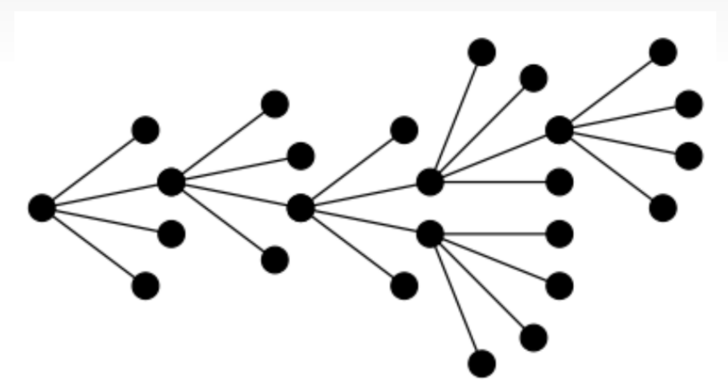
- Tentativa e erro é um modo metódico de tentar várias seqüências de decisões, até encontrar uma que funcione.
- Usada quando se quer achar soluções para problemas para os que não se conhece uma regra fixa de computação;
- Passos:
  - Escolher uma operação plausível;
  - Executar a operação com os dados;
  - Se a meta não foi alcançada, repita o processo até que se atinja a meta ou se evidencie a insolubilidade do problema.

# Definição

- Tentativa e erro é uma técnica que utiliza recursividade.
  - A recursividade pode ser usada para resolver problemas cuja solução é do tipo tentar todas as alternativas possíveis.
- Idéia para algoritmos tentativa e erro é decompor o processo em um número finito de subtarefas parciais (expressas de forma recursiva).
  - Explorá-las exaustivamente;
  - A construção de uma solução é obtida através de tentativas (ou pesquisas) da árvore de subtarefas.

# Definição

- O processo de tentativa gradualmente constrói e percorre uma árvore de subtarefas.



# Funcionamento

- Passos em direção à solução final são tentados e registrados em uma estrutura de dados;
- Caso esses passos tomados não levem à solução final, eles podem ser retirados e apagados do registro.
- A busca na árvore de soluções pode crescer rapidamente (exponencialmente)
  - Necessário usar algoritmos aproximados ou heurísticas que não garantem a solução ótima mas são rápidas.

# Funcionamento

- Exploramos cada possibilidade como segue:
  - Se a possibilidade é a resposta, retorne “sucesso”.
  - Se a possibilidade não for resposta, e não houver outra a ser testada a partir dela, retorne “falha”.
  - Para cada possibilidade, a partir da atual:
    - Explore a nova possibilidade (recursivo).
    - Se encontrou a resposta, retorne “sucesso”.
  - Retorne “falha”.

# Labirinto

- Dado um labirinto, encontre um caminho da entrada à saída.
    - Em cada interseção, você tem que decidir se:
        - Segue direto.
        - Vai à esquerda
        - Vai à direita
    - Você não tem informação suficiente para escolher corretamente.
    - Cada escolha leva a outro conjunto de escolhas.
    - Uma ou mais seqüência de escolhas pode ser a solução.

# O Problema das N-Rainhas

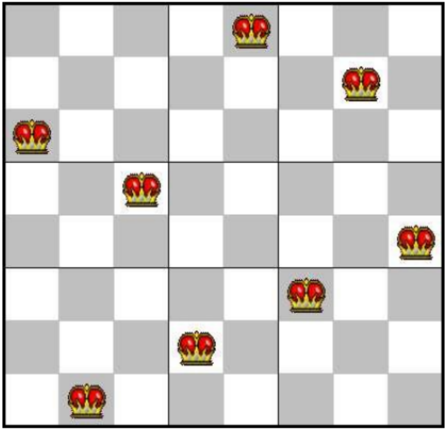
- O problema das $N$-rainhas é muito conhecido da forma como ele foi resolvido originalmente em 1850 por Gauss: com 8 rainhas;
- Posteriormente foi estendida para $N$-rainhas por Hoffman em 1969.

**Problema**

Colocar um determinado número $N$ maior ou igual a 2, de rainhas em um tabuleiro de xadrez de forma que elas não se ataquem simultaneamente, e ao final possa se dizer quantas formas deste existam.

## Solução

Para a resolução deste problema, utilizam-se algoritmos heurísticos, em particular algoritmos genéticos, de forma a medir a eficiência na resolução.



# O Problema das N-Rainhas

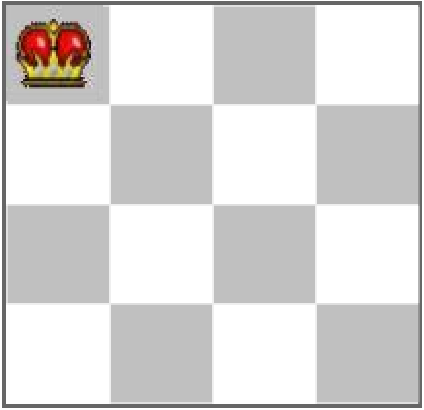
- No caso do problema das $N$-rainhas, o **backtracking** é utilizado quando for incrementar uma nova rainha e não tiver mais posições no tabuleiro no modo em que a mesma esteja em posição de defesa.
- Com isso, é retirada a rainha do topo da pilha e é voltado para a rainha anterior, logo em seguida é mudada a mesma em outra, então depois é inserido a rainha

# Simulação

- Resolução por backtracking.
- Colocar a primeira rainha em uma posição da primeira coluna do tabuleiro,
- a segunda rainha em uma posição da segunda coluna,
- a terceira em uma posição da terceira coluna e assim por diante...
- Cada rainha irá se movimentar na sua coluna para tentarmos achar a solução!
- Ao colocar uma rainha, é preciso verificar se elas se atacam.
- Se elas se atacarem, realiza-se um **backtracking** (volta para algum estado anterior) para tentar novamente de outra forma.

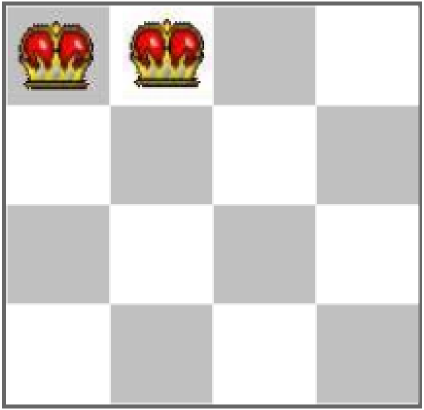
# Simulação

Coloca-se a primeira rainha numa posição da primeira coluna:



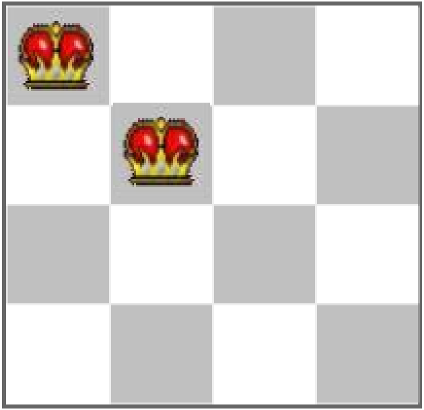
# Simulação

Coloca-se a segunda rainha numa posição da segunda coluna:
Ops, as rainhas se atacam, tenta colocar a segunda rainha em outra posição...



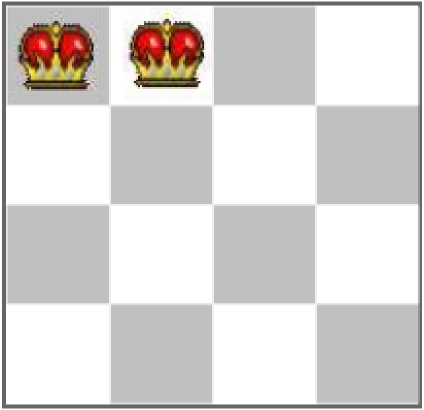
# Simulação

Ops, as rainhas se atacam, tenta colocar a segunda rainha em outra posição...
Backtracking!



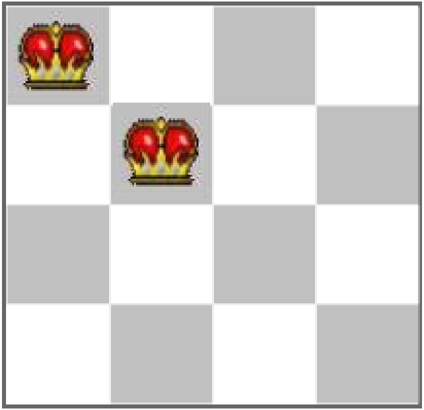
# Simulação

Ops, as rainhas se atacam, tenta colocar a segunda rainha em outra posição...
Backtracking!



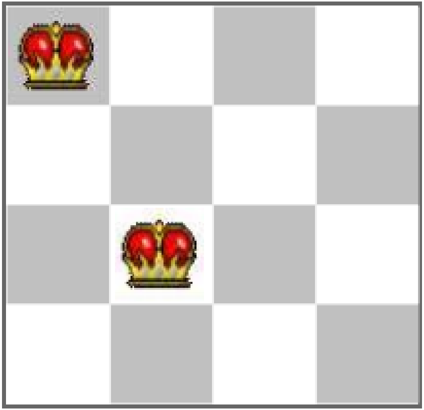
# Simulação

Ops, as rainhas se atacam, tenta colocar a segunda rainha em outra posição...
Backtracking!



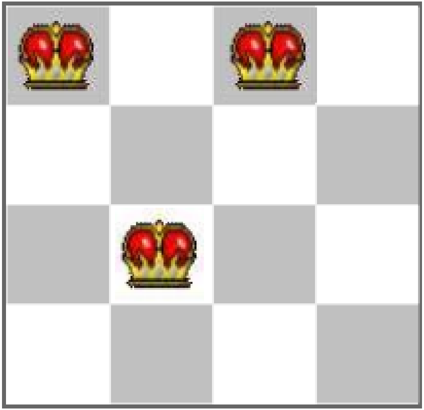
# Simulação

Legal, conseguimos colocar a segunda rainha de forma que as rainhas não se ataquem.
Vamos tentar colocar a terceira rainha...



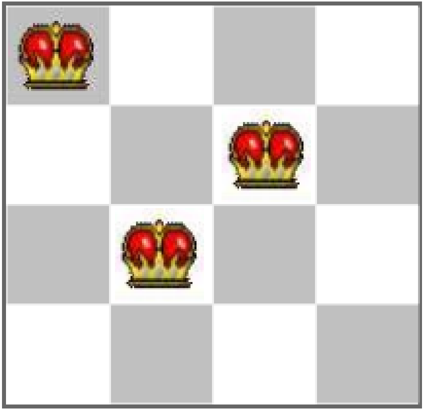
# Simulação

Colocamos a terceira rainha na terceira coluna.
Ops, as rainhas se atacam! Tenta colocar a terceira rainha em outra posição da terceira coluna.



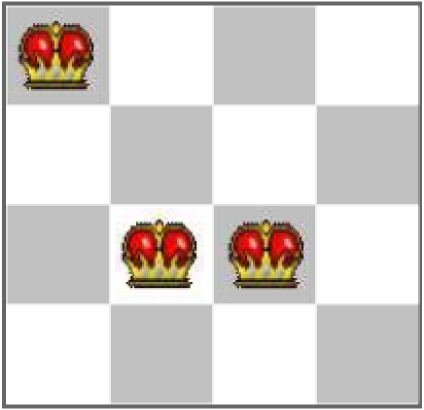
# Simulação

Ops, as rainhas se atacam! Tenta colocar a terceira rainha em outra posição da terceira coluna.



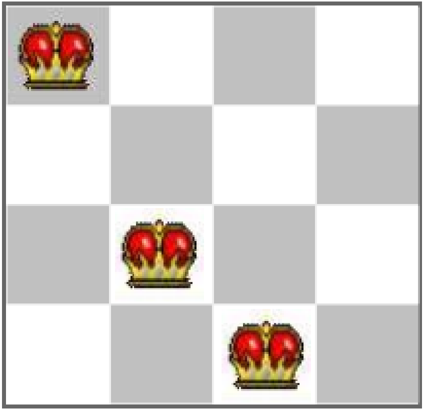
# Simulação

Ops, as rainhas se atacam! Tenta colocar a terceira rainha em outra posição da terceira coluna.



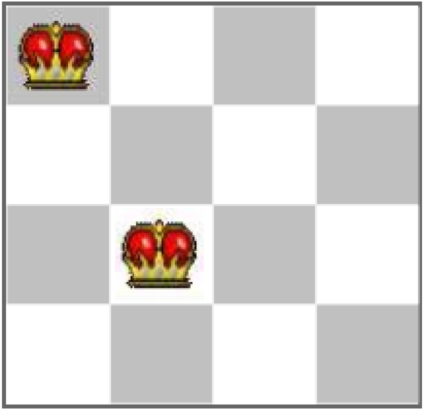
# Simulação

Ops, as rainhas se atacam!
Não conseguimos, realizamos o backtracking...



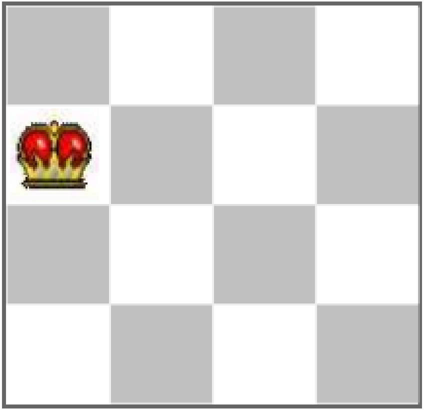
# Simulação

Mais backtracking...

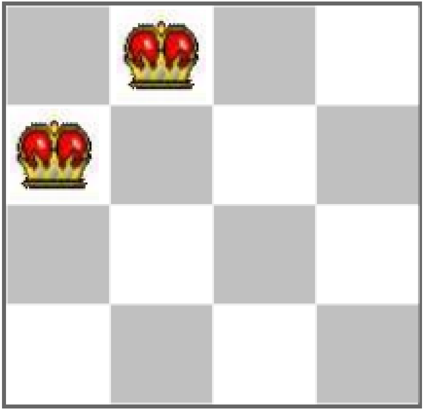


# Simulação

Tentamos uma posição diferente para a primeira rainha....

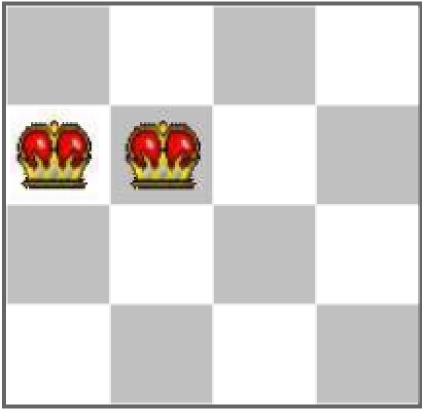


Vamos colocar a segunda rainha... Ops, elas se atacam!



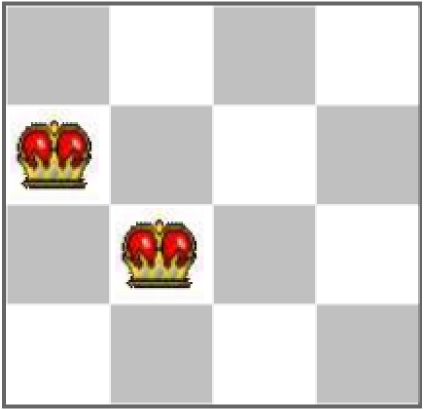
# Simulação

Ops, elas se atacam novamente!



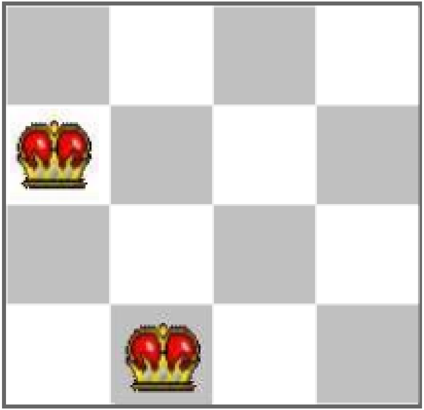
# Simulação

Ops, elas se atacam novamente!



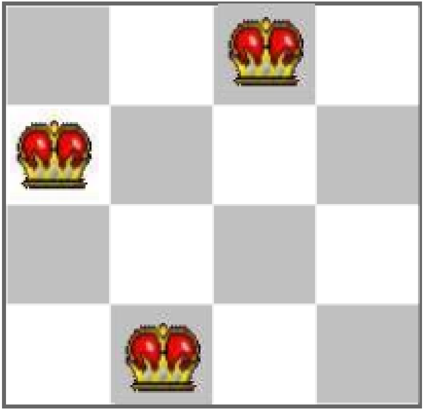
# Simulação

Ok, finalmente conseguimos colocar a segunda rainha!



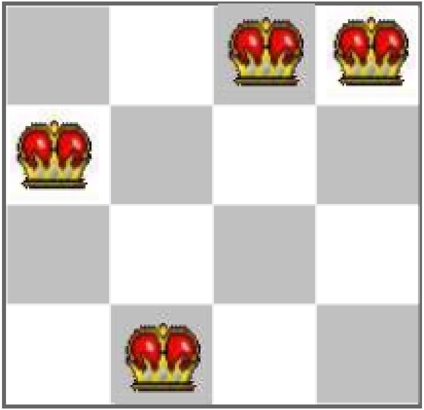
# Simulação

Vamos colocar a terceira... Legal!! Conseguimos colocar a terceira!



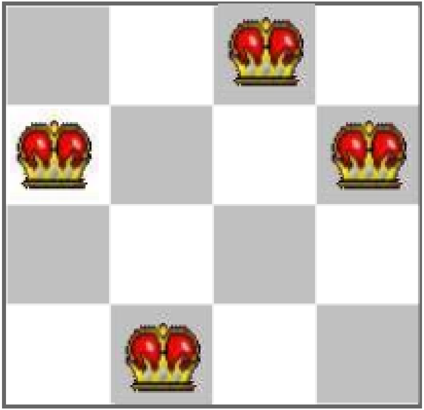
# Simulação

Vamos colocar a quarta... Ops, elas se atacam....



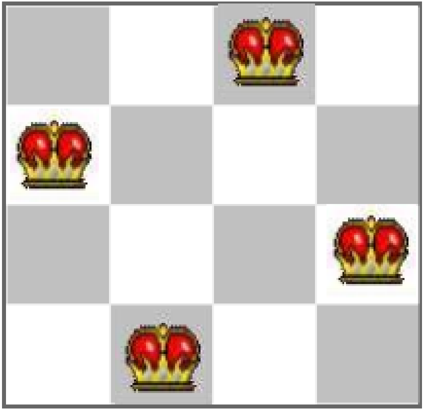
# Simulação

Ops, elas ainda se atacam...



# Simulação

Legal!! Conseguimos colocar as 4 rainhas de forma que elas não se ataquem!



# Estudo complementar

- **Desafio**: Implemente a solução para o problema das $N$-rainhas.
- Pequise mais exemplos de algoritmos que usam Backtracking:
  - Passeio do Cavalo;
  - Jantar dos Filósofos;
  - Problema da mochila;