

# Estrutura de Dados I

## Capítulo 6: Algoritmos de Backtracking

# Algoritmos de backtracking (retrocesso)

- Muitos problemas exigem que a solução seja encontrada através de uma seqüência de pontos de decisão nos quais cada decisão leva você mais próximo (ou mais distante!) da solução através de um certo “caminho”.
- Se você fizer as escolhas corretas, chegará na solução. Caso contrário você chegará em um ponto “sem saída” ou simplesmente descobrirá que fez uma escolha errada (ficou mais distante da solução), e você terá que **retroceder para um ponto de decisão anterior e tentar outra escolha** (BACKTRACK).
- Algoritmos que usam essa abordagem são chamados de **algoritmos de backtracking** (retrocesso).

# Algoritmos de backtracking (retrocesso)

- Podemos pensar em backtracking como **o processo de repetidamente explorar novos caminhos até que a solução seja encontrada**. A maioria desses problemas são mais facilmente resolvidos de forma recursiva!
- O insight recursivo aqui é o seguinte:  
**Um problema de backtracking tem uma solução se, e somente se, pelo menos um dos problemas menores de backtracking resultantes de cada escolha possível tiver uma solução.**
  - sim, é complicado mesmo...
- Estudaremos alguns exemplos de backtracking (labirinto e jogos de estratégia entre 2 jogadores) para que você comece a entender o assunto. O domínio leva tempo.

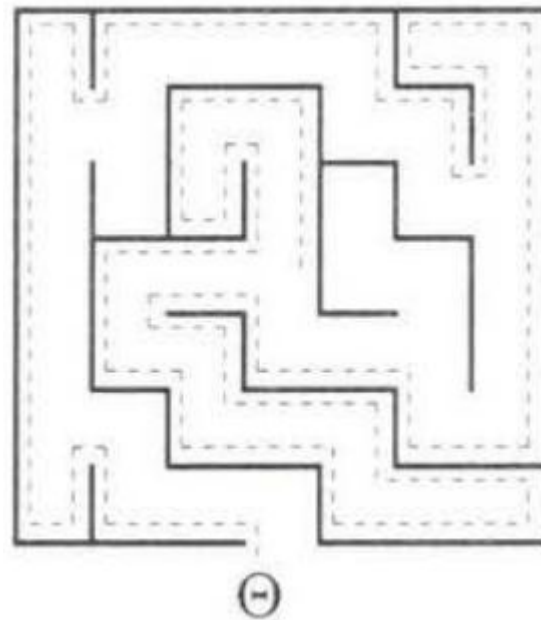
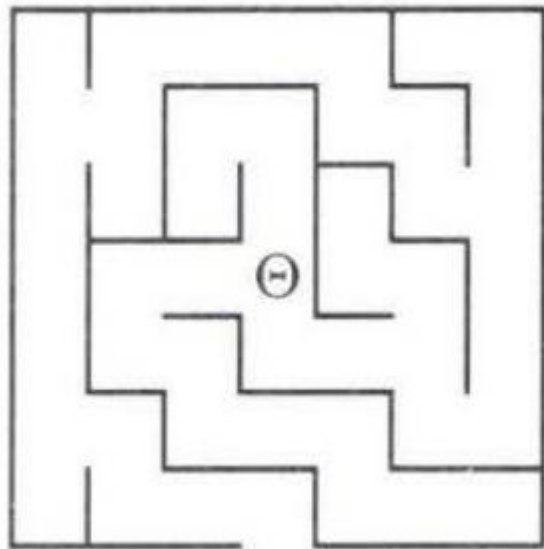
# Saindo de um labirinto pela regra da mão direita

- Uma das estratégias para sair de um labirinto é a chamada **regra da mão direita**, que pode ser expressa em pseudo-código da seguinte forma:

```
while (/* você não sair do labirinto */)
{
    // Mantenha sua mão direita na parede e
    // ande para frente.
}
```

- Você pode acabar voltando por cantos e corredores por onde já andou mas, se seguir essa estratégia, você sempre encontrará a saída do labirinto.

Saindo de um labirinto pela regra da mão direita



# Saindo de um labirinto por backtracking

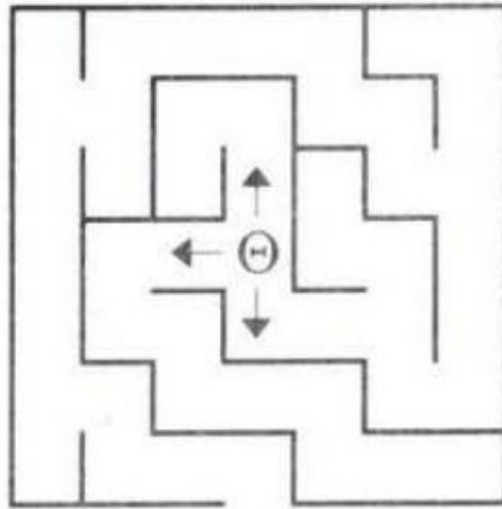
- O “while” na regra da mão direita parece indicar um processo iterativo.

Como encontrar uma solução recursiva?

- Adotar um “mindset” diferente
  - Não pense em termos de encontrar um caminho completo
  - **O seu objetivo é achar um insight recursivo que simplifique o problema, um passo de cada vez**
  - Depois que você fez a simplificação, pode usar o mesmo procedimento para resolver cada um dos subproblemas resultantes
- Vamos fazer uma análise detalhada para encontrar os subproblemas.

# Saindo de um labirinto por backtracking

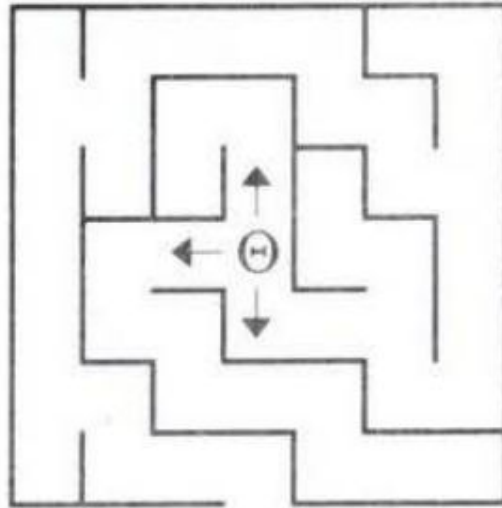
- Da posição inicial do labirinto, você tem 3 escolhas de caminhos, conforme a figura abaixo:



- A saída, se existir, deve ser encontrada seguindo-se um desses caminhos. Mais ainda: **se a escolha for correta, estamos 1 passo mais próximo da saída.**

# Saindo de um labirinto por backtracking

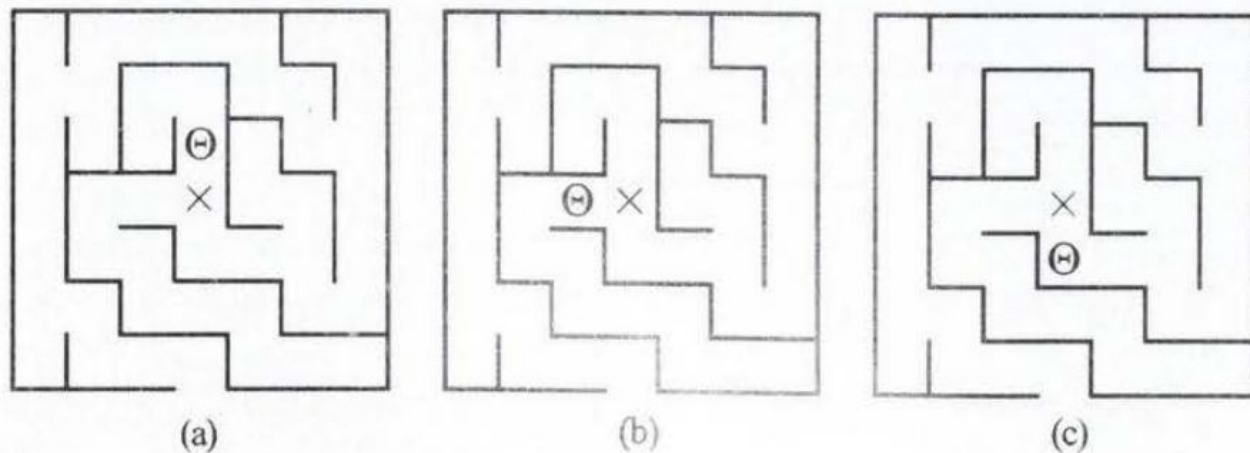
- Se escolhermos o caminho correto, o labirinto tornou-se mais simples (1 passo menor) e isso é a chave para uma solução recursiva.





# Saindo de um labirinto por backtracking

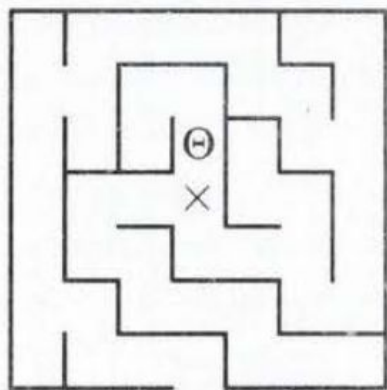
- Agora o mais difícil, o insight recursivo: **o labirinto original terá solução se, e somente se, for possível resolver pelo menos 1 dos novos labirintos (menores) mostrados na figura abaixo** (decomposição recursiva):



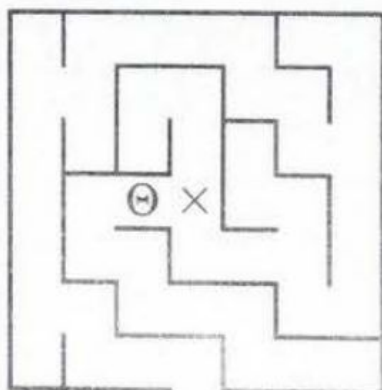
- A letra “X” marca um **ponto já visitado** e é “proibido” nas decomposições recursivas pois a solução ótima nunca terá que voltar para essa posição.

# Saindo de um labirinto por backtracking

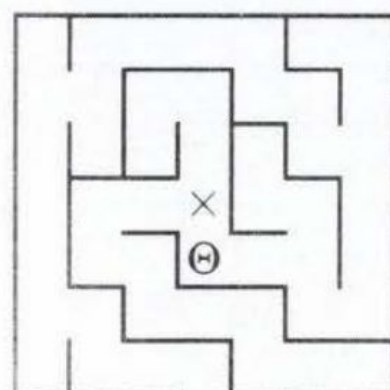
- Visualmente, de nosso ponto de vista privilegiado, é fácil perceber que as decomposições A e B não levarão à solução, e que a C levará à solução. Mas **isso é um erro que você tem que evitar!**
- Se você está pensando recursivamente, **NÃO DEVE PENSAR EM ACHAR O CAMINHO COMPLETO!**



(a)



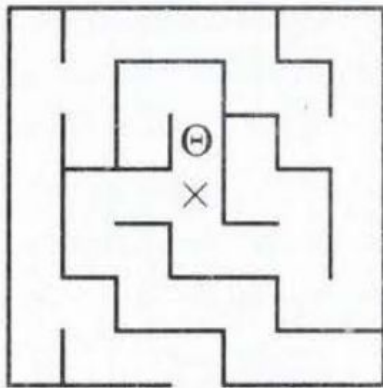
(b)



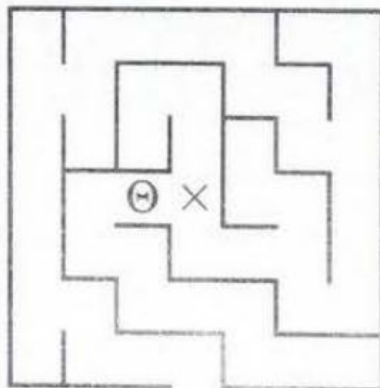
(c)

# Saindo de um labirinto por backtracking

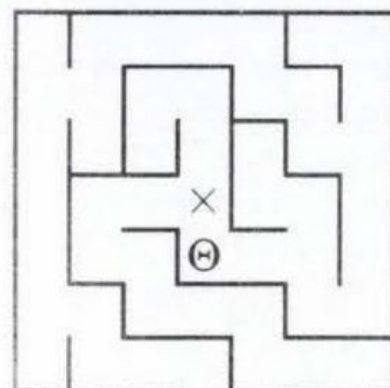
- Você **já achou a decomposição** que torna o problema mais simples, com a mesma forma do original.
- Tudo que você tem a fazer é **confiar no salto de fé recursivo e usar o poder da recursividade para resolver esses subproblemas individuais** e pronto! (ainda temos que achar os casos simples)



(a)



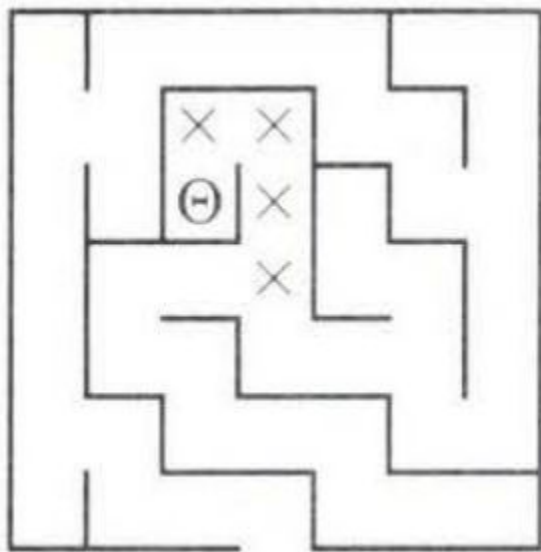
(b)



(c)

# Saindo de um labirinto por backtracking

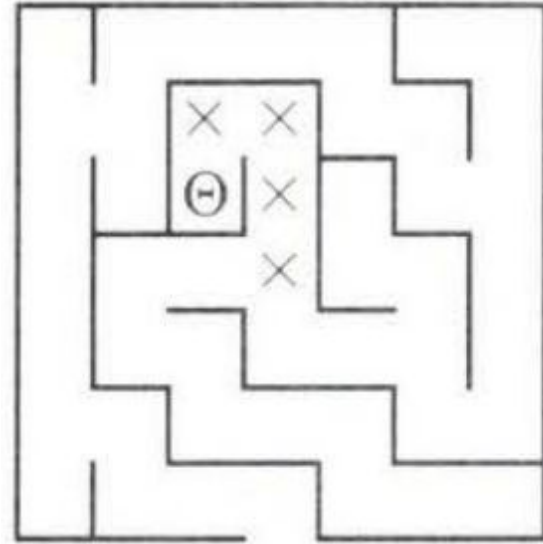
- E quais seriam os casos simples?
  - Se você já estiver fora do labirinto, então já está resolvido
  - Se você chegar em um corredor sem saída, então não tem solução



Todo caminho a partir daqui está marcado com um “X” (que indica que já passamos por aqui) ou bloqueado por uma parede. Isso significa que não há solução a partir deste ponto.

# Saindo de um labirinto por backtracking

- Do ponto de vista do código é mais fácil se, ao invés de checarmos por uma marca X ao considerar as possíveis futuras direções, nós fizermos o movimento e, no início do procedimento, checarmos se estamos atualmente em um X.
- Se estivermos em um X, podemos terminar a recursão nesse caminho pois, se estamos sobre um X, estamos voltando no labirinto o que significa que se a solução, se existir, deve estar em um dos outros caminhos.



# Saindo de um labirinto por backtracking

- Chegamos assim aos nossos casos simples:
  - Se a posição atual for fora do labirinto, achamos a solução;
  - Se a posição atual for um X (estiver marcada), esse caminho não tem solução.

# Saindo de um labirinto por backtracking

- Temos um outro problema: como criar a **representação do labirinto** por si mesmo, de forma a nos permitir:
  - detectar paredes
  - detectar a posição atual
  - detectar se a posição atual está marcada (X)
  - detectar se a posição atual está fora do labirinto
- Para facilitar, já temos uma biblioteca que nos ajudará a resolver esses problemas, a “`labirinto.h`”. Isso nos permitirá focar na solução recursiva!

(faça o download do arquivo no site Disciplinas UVV)

# Saindo de um labirinto por backtracking

```
/* Inicia o boilerplate da interface: */
#ifndef _LABIRINTO_H
#define _LABIRINTO_H

#include "genlib.h"

/**
 * Tipo: pontoT
 * -----
 * O tipo pontoT é usado para encapsular um par de inteiros que formam as
 * coordenadas x e y de um único ponto que indica uma posição no labirinto. A
 * separação da struct de seu typedef é feita por motivos didáticos.
 */
struct st_pontoT
{
    int x, y;
};

typedef struct st_pontoT pontoT;
```



# Saindo de um labirinto por backtracking

```
/**
 * Tipo: direcaoT
 * -----
 * O tipo direcaoT é usado para representar as quatro direções principais
 * da bússola, que são as direções que podemos nos mover a partir de um ponto
 * no labirinto. Implementado como uma enumeração. A separação da enum de seu
 * typedef é feita por motivos didáticos.
 */

enum en_direcaoT
{
    Norte, Leste, Sul, Oeste
};

typedef enum en_direcaoT direcaoT;
```

# Saindo de um labirinto por backtracking

```
/**
 * Procedimento: ler_mapa_labirinto
 * Uso: ler_mapa_labirinto(arquivo);
 * -----
 * Este procedimento lê o mapa de um labirinto a partir de um arquivo
 * específico passado como argumento, e armazena esse mapa em uma estrutura
 * de dados privada mantida por esta biblioteca. No arquivo com o mapa do
 * labirinto, os caracteres '+', '-' e '|' representam esquinas, paredes
 * verticais e paredes horizontais, respectivamente; espaços representam
 * passagens abertas. A posição inicial é indicada pelo caractere 'I'. Por
 * exemplo, o seguinte arquivo de dados define um mapa simples:
 *
 *      +-----+----+      A solução deve ser: andar 2 casas para cima, 4
 *      |         |         casas para direita, 2 casas para baixo, 2 casas
 *      + +-+ + +-+      para direita, 2 casas para cima, e 4 casas para
 *      |I  |         |         a direita (até sair do labirinto).
 *      +---+-----+
 *
 * As coordenadas são numeradas começando em (0, 0), no canto inferior esquerdo.
 * No exemplo acima, os 4 cantos do labirinto e o início estariam em:
 *
 *      inf. esq. = ( 0, 0)
 *      inf. dir. = (10, 0)
 *      sup. esq. = ( 0, 4)
 *      sup. dir. = (10, 4)
 *      Início   = ( 1, 1)
 */
void ler_mapa_labirinto (string arquivo);
```

# Saindo de um labirinto por backtracking

```
/**
 * Função pegar_posicao_inicial
 * Uso: pt = pegar_posicao_inicial();
 * -----
 * Esta função retorna um pontoT indicando as coordenados do ponto de início.
 */

pontoT pegar_posicao_inicial (void);

/**
 * Predicado: saiu_do_labirinto
 * Uso: if (saiu_do_labirinto(pt)) ...
 * -----
 * Este predicado retorna TRUE se o ponto especificado está fora do labirinto.
 */

bool saiu_do_labirinto (pontoT pt);
```

# Saindo de um labirinto por backtracking

```
/**
 * Predicado: e_parede
 * Uso: if (e_parede(pt, dir)) ...
 * -----
 * Esta predicado retorna TRUE se houver uma parede na direção (dir) indicada,
 * a partir do pontoT (pt) fornecido.
 */

bool e_parede (pontoT pt, direcaoT dir);

/**
 * Predicado: esta_marcado
 * Uso: if (esta_marcado(pt)) ...
 * -----
 * Este predicado retorna TRUE se o pontoT indicado por pt estiver marcado, ou
 * seja, já tiver sido visitado ao percorrer o labirinto.
 */

bool esta_marcado (pontoT pt);

/**
 * Procedimentos: marcar_ponto, desmarcar_ponto
 * Uso: marcar_ponto(pt);
 *      desmarcar_ponto(pt);
 * -----
 * Esses procedimentos marcar ou desmarcam o status de um pontoT (pt).
 */

void marcar_ponto (pontoT pt);
void desmarcar_ponto (pontoT pt);

#endif
```

# Saindo de um labirinto por backtracking

- Para criar a solução recursiva, vamos criar o predicado `resolver_labirinto`, que utilizará backtracking recursivo e receberá como argumento a posição inicial (que muda a cada subproblema recursivo).
  - Retorna TRUE se achar alguma solução
  - Retorna FALSE se não achar nenhuma solução

```
bool resolver_labirinto (pontoT pt);
```

# Saindo de um labirinto por backtracking

- Para criar o programa cliente, não há nenhum segredo:

```
#include "genlib.h"
#include "labirinto.h"
#include "simpio.h"

/* Declarações de subprogramas: */

bool resolver_labirinto (pontoT pt);

/* Função Main: */

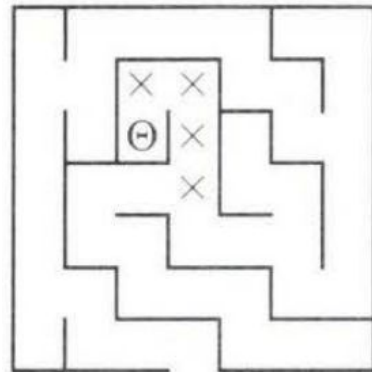
int main (void)
{
    ler_mapa_labirinto("arquivo.txt");

    if (resolver_labirinto(pegar_posicao_inicial()))
        printf("As posições com X marcam o caminho para a saída.\n");
    else
        printf("Não existe saída do labirinto.\n");
}
```

# Saindo de um labirinto por backtracking

- O predicado `resolver_labirinto`, que é nossa solução recursiva, precisa de um pouco mais de atenção. Em pseudocódigo, aqui estão os casos simples:

```
bool resolver_labirinto (pontoT pt)
{
    // Se a posição atual estiver fora do labirinto, retornar TRUE
    // Se a posição atual estiver marcada, retornar FALSE
}
```



# Saindo de um labirinto por backtracking

- Agora a decomposição recursiva:

```
bool resolver_labirinto (pontoT pt)
{
    // Se a posição atual estiver fora do labirinto, retornar TRUE
    // Se a posição atual estiver marcada, retornar FALSE

    // Marcar a posição atual
    for (/* cada uma das 4 direções possíveis */)
    {
        if (/* esta posição não estiver bloqueada por uma parede */)
        {
            // Ande 1 passo na direção indicada a partir do ponto atual
            // Tente resolver o labirinto fazendo uma chamada recursiva
            // Se a chamada recursiva retornar TRUE, retorne TRUE
        }
    }
    // Desmarque a posição atual
    // Retorne FALSE para indicar que nenhuma das 4 direções achou solução
}
```



# Saindo de um labirinto por backtracking

```
bool resolver_labirinto (pontoT pt)
{
    direcaoT dir;

    if (saiu_do_labirinto(pt)) return TRUE;
    if (esta_marcado(pt)) return FALSE;

    marcar_ponto(pt);
    for (dir = Norte; dir <= Oeste; dir++)
    {
        if (!e_parede(pt, dir))
        {
            if (resolver_labirinto(novo_ponto(pt, dir)))
                return TRUE;
        }
    }
    desmarcar_ponto(pt);
    return FALSE;
}
```

# Saindo de um labirinto por backtracking

- Note que nossa solução utiliza uma função adicional: `novo_ponto`

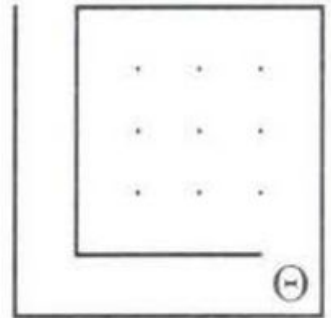
```
pontoT novo_ponto (pontoT pt, direcaoT dir)
{
    pontoT novo_ponto = pt;
    switch (dir)
    {
        case Norte: novo_ponto.y++; break;
        case Leste: novo_ponto.x++; break;
        case Sul:    novo_ponto.y--; break;
        case Oeste: novo_ponto.x--; break;
    }
    return novo_ponto;
}
```

# Saindo de um labirinto por backtracking

- Na solução do labirinto nós desmarcamos os pontos para os quais uma solução não foi encontrada:

```
    }  
  }  
  desmarcar_ponto(pt);  
  return FALSE;
```

- Isso não é estritamente necessário, só fizemos isso para que o caminho que leva até a solução apareça destacado no final. Mas isso pode levar a uma queda imensa de performance se existem loops no algoritmo:



# Saindo de um labirinto por backtracking

- Em algum momento você será capaz de olhar para o código do predicado `resolver_labirinto` e dizer para você mesmo: **“Eu realmente SEI como isso funciona: o problema está ficando menor porque mais pontos serão marcados a cada passo; os casos simples estão claros e corretos. Esse predicado está correto e fará o trabalho de achar a saída do labirinto.”**

- Problema: essa confiança não vem fácil. Seu ceticismo natural fará com que você queira ver os passos da solução. NÃO FAÇA ISSO! São 66 chamadas em 27 níveis, para o labirinto mostrado aqui.

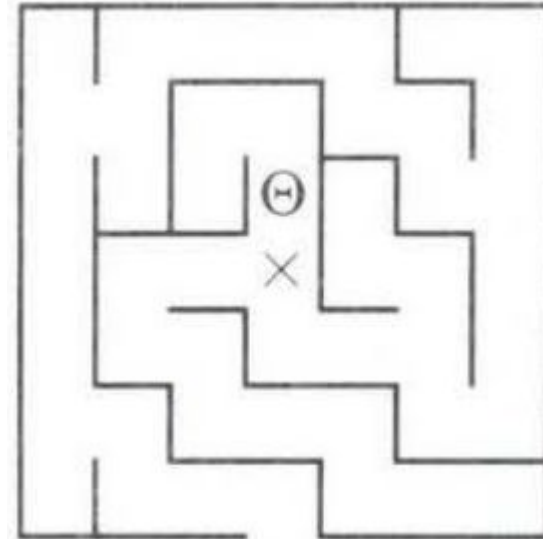
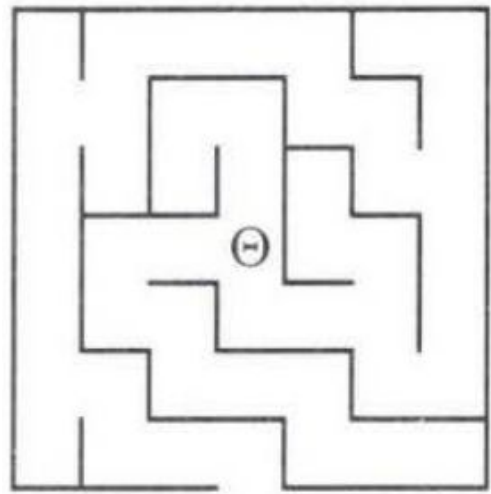
```
bool resolver_labirinto (pontoT pt)
{
    direcaoT dir;

    if (saiu_do_labirinto(pt)) return TRUE;
    if (esta_marcado(pt)) return FALSE;

    marcar_ponto(pt);
    for (dir = Norte; dir <= Oeste; dir++)
    {
        if (!e_parede(pt, dir))
        {
            if (resolver_labirinto(novo_ponto(pt, dir)))
                return TRUE;
        }
    }
    desmarcar_ponto(pt);
    return FALSE;
}
```

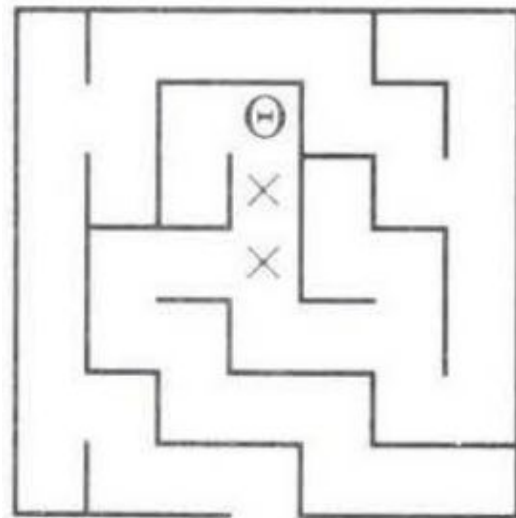
# Saindo de um labirinto por backtracking

- Se você ainda não está preparado para aceitar o SALTO DE FÉ recursivo, o melhor que você pode fazer é rastrear o andamento do código em termos mais gerais.
- Você sabe que o código primeiro tenta resolver o labirinto indo para o NORTE, devido ao `for` que utiliza a enumeração `en_direcaoT`. Assim, o primeiro passo é:



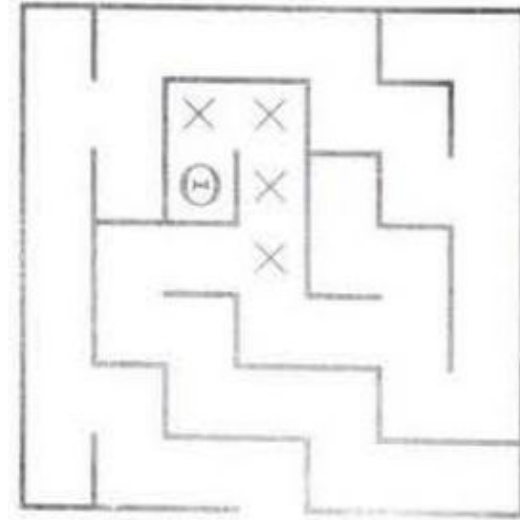
# Saindo de um labirinto por backtracking

- Depois do primeiro passo, o processo ocorre de novo e vamos novamente para o norte:
- Depois de chegar na posição ilustrada, o código faz outra chamada recursiva. Dessa vez:
  - Não é possível ir para o NORTE (parede)
  - Agora tenta o LESTE, e também não pode (parede)
  - Então ele vai para o SUL, mas como já está marcado, ele **RETROCEDE** para a posição anterior
  - Finalmente ele vai para o OESTE, que é a única direção possível a partir deste ponto.



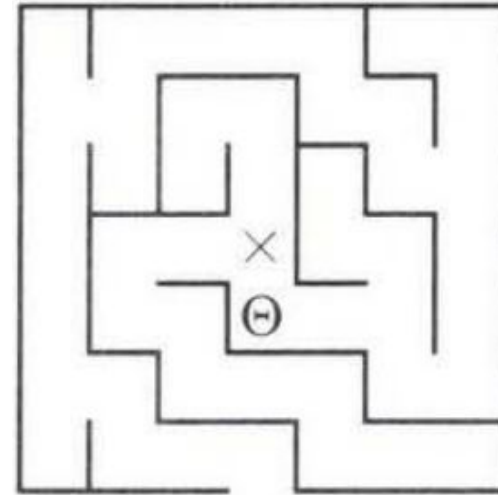
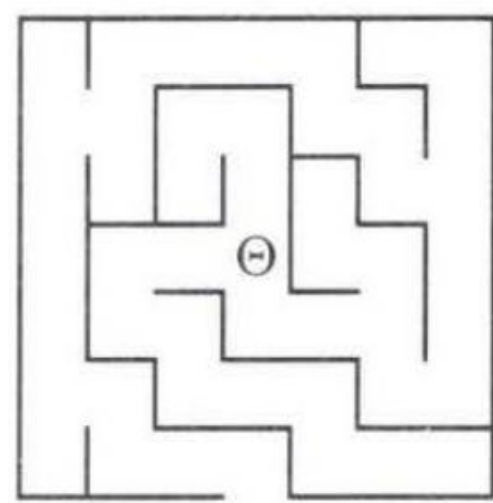
# Saindo de um labirinto por backtracking

- Eventualmente chegamos na posição ilustrada:
  - Não podemos ir para NORTE (já está marcado), nem para LESTE, SUL ou OESTE (paredes).
  - O programa desmarca a posição atual e retorna FALSE para o nível anterior (BACKTRACKING)
  - No nível anterior também já exploramos outros caminhos então o programa desmarca a posição atual e retorna FALSE para o nível anterior (BACKTRACKING)
  - Isso continua até que voltamos (BACKTRACKING) na situação inicial do labirinto, tendo percorrido todas as possibilidades cujo movimento inicial era ir para o NORTE.



# Saindo de um labirinto por backtracking

- Depois de tentar todas as possibilidades pelo NORTE, voltamos na posição inicial.
- Como já tentamos o NORTE, o algoritmo tenta o LESTE, mas não pode pois é uma parede. Então a próxima tentativa é ir para o SUL, e aí temos a seguinte situação:

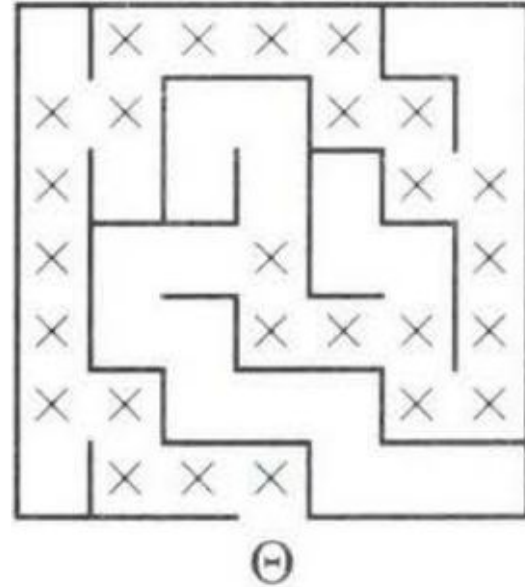






# Saindo de um labirinto por backtracking

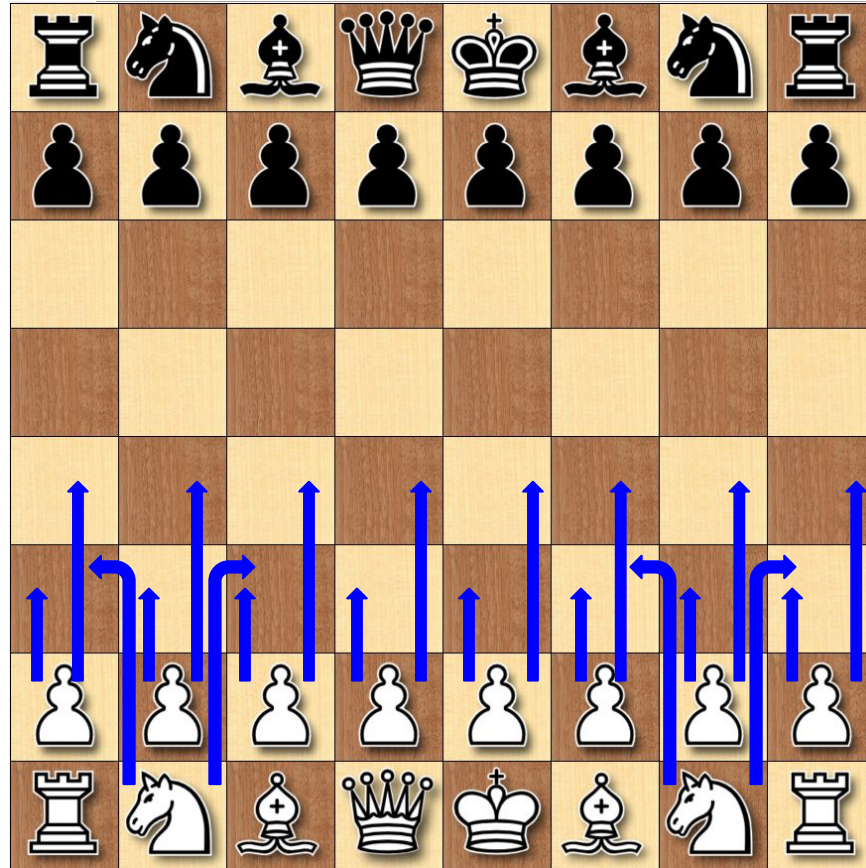
- No momento que a posição atual for a ilustrada, o caso simples é acionado e retorna TRUE para quem o chamou.
- Esse valor TRUE é então propagado de volta através das 66 chamadas em 27 níveis de recursão até chegar na função `main` novamente, indicando que o labirinto tem solução.



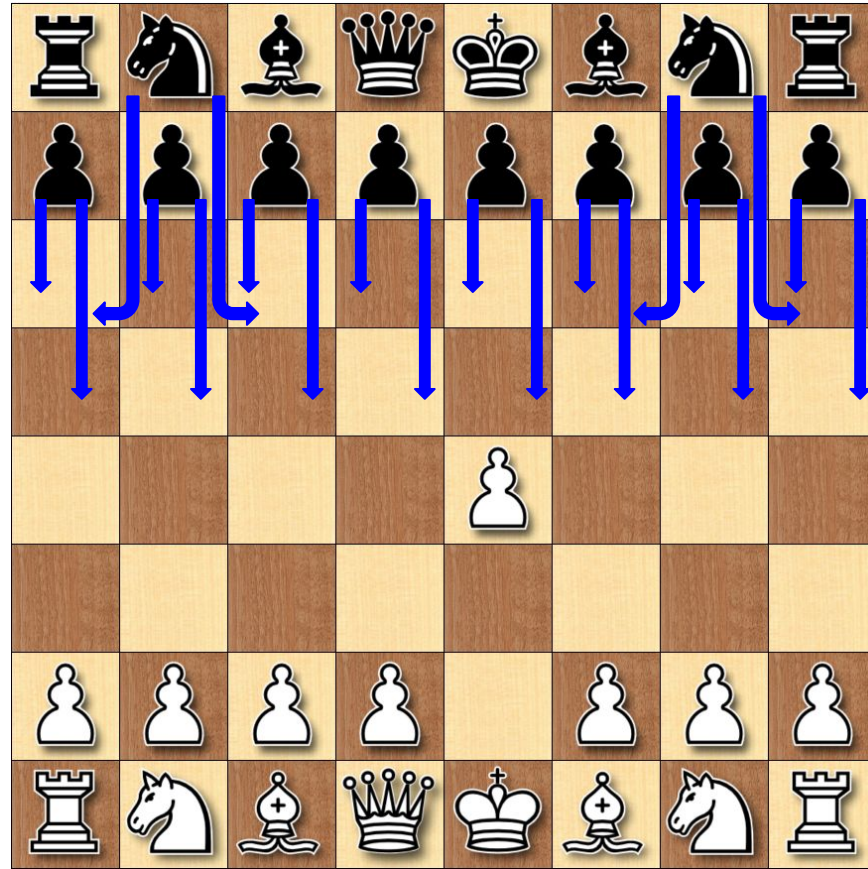
# Backtracking e jogos

- Sair de um labirinto ilustra perfeitamente o uso de backtracking mas, na verdade, a estratégia de backtracking é muito mais geral e pode ser aplicada, por exemplo, à diversos jogos com 2 jogadores:
  - O 1º jogador tem diversas escolhas para o primeiro movimento, e escolhe uma;
  - Depois que o 1º jogador faz seu movimento, o 2º jogador tem outro conjunto de movimentos que podem ser feitos em resposta.
  - Depois que o 2º jogador faz seu movimento, o 1º jogador agora tem um outro conjunto de movimentos possíveis, e faz sua escolha.
  - O processo continua até o final do jogo.
- As diferentes posições possíveis a cada jogada formam uma estrutura ramificada onde cada opção abre mais e mais possibilidades.

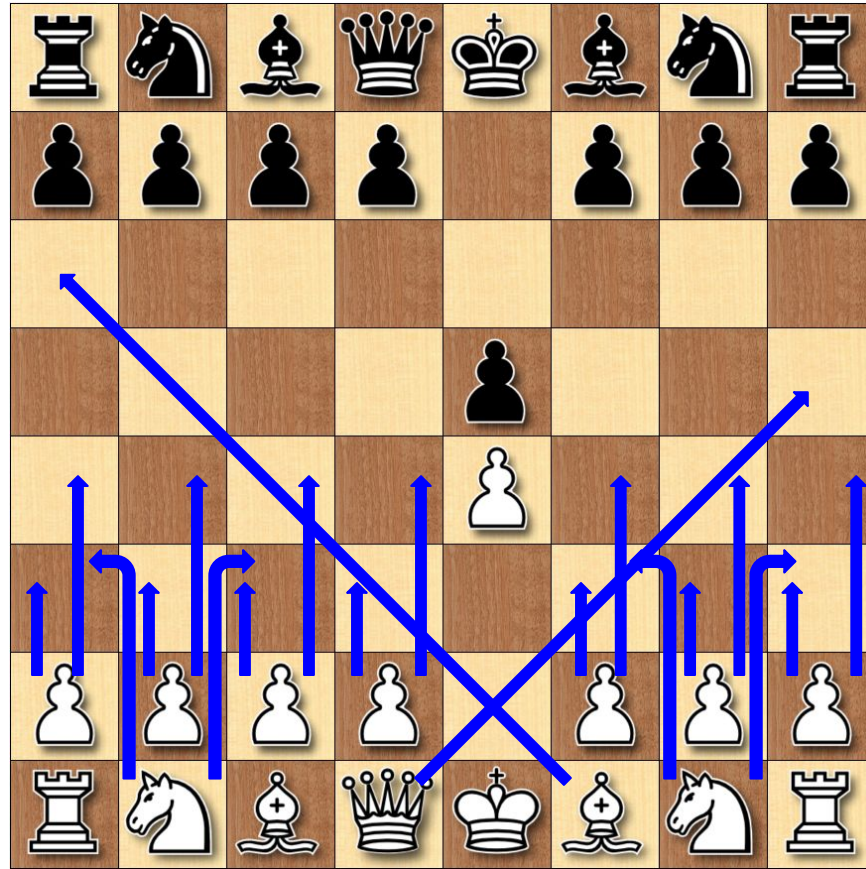
# Backtracking e jogos



# Backtracking e jogos



# Backtracking e jogos





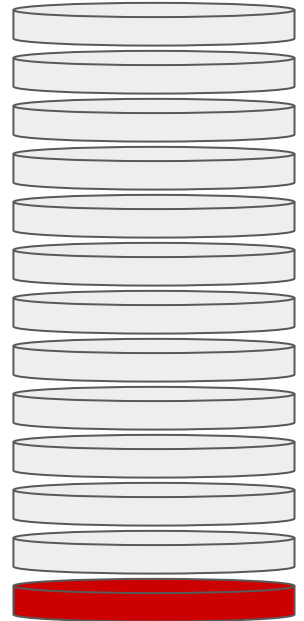
# Backtracking e jogos

- Como poderíamos fazer o computador jogar no lugar de uma pessoa?
  - Seguir todas as ramificações na lista de possibilidades e escolher uma;
  - Mas, ANTES de fazer o 1º movimento, o computador tentaria determinar qual seria a resposta possível do oponente, para cada uma das possibilidades;
  - E aí, ANTES de fazer o 1º movimento, ele tentaria determinar quais seriam suas respostas para cada uma das possíveis jogadas do oponente
  - E assim por diante
- Se o computador for capaz de “olhar” longe o suficiente para encontrar algum movimento que colocaria seu oponente em uma má posição, aí sim ele escolheria esse movimento e faria sua 1ª jogada
- Problema: poder computacional (mas existem jogos possíveis).



# Backtracking para “nim games”

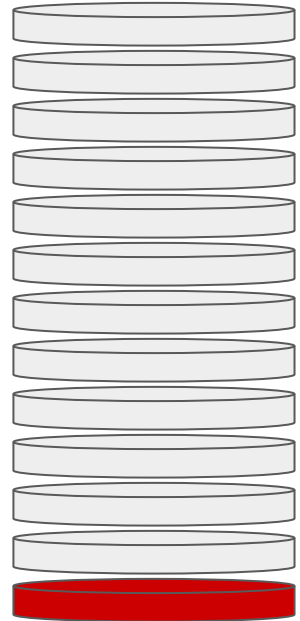
- “Nim” é uma palavra que descreve uma grande categoria de jogos nos quais 2 jogadores se revezam removendo objetos de alguma configuração inicial:
  - <https://en.wikipedia.org/wiki/Nim>
- Programaremos uma versão de Nim que consiste em uma pilha de 13 moedas. A cada jogada um jogador pode retirar 1, 2 ou 3 moedas da pilha. Perde o jogo quem for obrigado a retirar a última moeda.
- Como encontrar uma estratégia para esse jogo?



# Backtracking para “nim games”

```
[abrantesaf@ideapad ~/ed1/cap06]$ ./nim
Olá! Bem-vindo ao jogo de nim.
Neste jogo começaremos com uma pilha de
13 moedas na mesa.
A cada jogada você e eu retiraremos,
de forma alternada, entre 1 e 3 moedas
da mesa. O jogador que for obrigado a
retirar a última moeda perde.
```

```
Existem 13 moedas na pilha.
Quantas moedas você irá retirar? 2
Existem 11 moedas na pilha.
Eu retirarei 2 moedas.
Existem 9 moedas na pilha.
Quantas moedas você irá retirar? 3
Existem 6 moedas na pilha.
Eu retirarei 1 moedas.
Existem 5 moedas na pilha.
Quantas moedas você irá retirar? 1
Existem 4 moedas na pilha.
Eu retirarei 3 moedas.
Só há 1 moeda restante.
Eu ganhei!
```



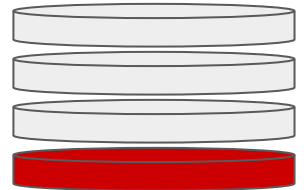
# Backtracking para “nim games”

- Como encontrar uma estratégia para esse jogo? Ir de trás para frente!
- Se você estiver apenas com uma moeda na pilha, você está em uma **má posição** e terá que fazer um **movimento ruim**: terá que pegar essa moeda e perderá o jogo:



# Backtracking para “nim games”

- Se a pilha tiver 2, 3 ou 4 moedas, você está em uma **boa posição**, pois pode fazer um **bom movimento** (tirar 1, 2 ou 3 moedas) e deixar o seu oponente em uma **má posição** (ficar apenas com 1 moeda):

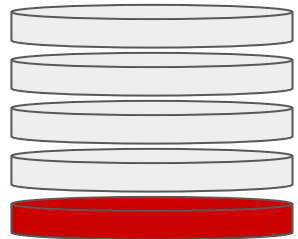


# Backtracking para “nim games”

- E se a pilha tiver 5 moedas?

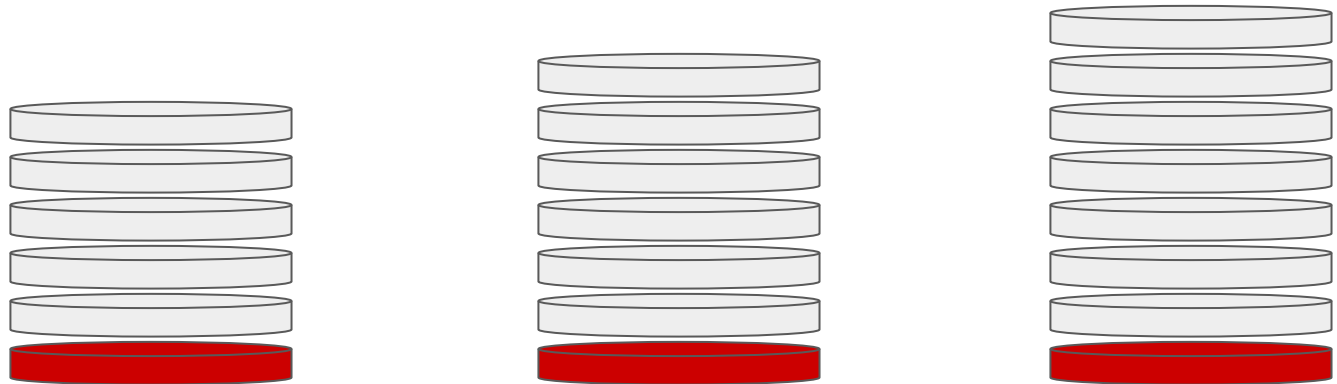
Você está em uma **má posição** pois não importa o que você faça (retirar 1, 2 ou 3 moedas), sempre deixará seu oponente em uma boa posição: ele vai ficar com pilhas de 4, 3 ou 2 moedas (que já vimos que são boas posições).

Você está condenado se tiver 5 moedas pois, a partir daqui, você **não terá nenhum movimento bom** para deixar seu oponente em uma má posição.



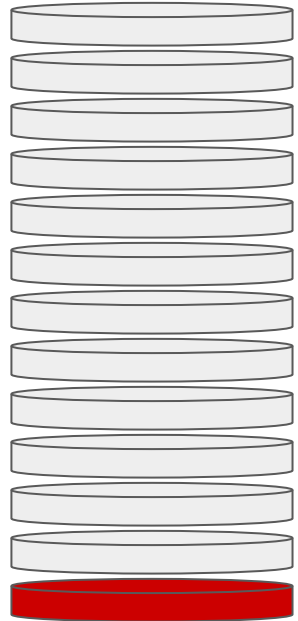
# Backtracking para “nim games”

- Se a pilha tiver 6, 7 ou 8 moedas, você está em uma **boa posição**, pois pode fazer um **bom movimento** (tirar 1, 2 ou 3 moedas) e deixar o seu oponente em uma **má posição** (ficar com 5 moedas):



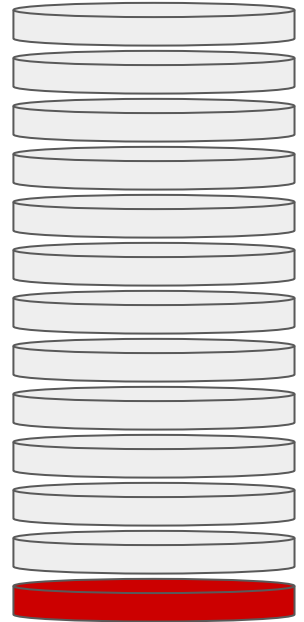
# Backtracking para “nim games”

- Em resumo, a cada vez você está buscando um bom movimento!
- **Um bom movimento é aquele que deixa seu oponente em uma má posição.**
- **Uma má posição é aquela a partir da qual não há nenhum bom movimento possível.**
- Apesar das definições de bom movimento e má posição serem circulares, elas são uma estratégia completa para nosso jogo de nim. **Confie no poder da recursão!**



# Backtracking para “nim games”

- Se tivermos uma função “`achar_bom_movimento`” que receba o número de moedas, tudo que ela tem a fazer é testar todas as possibilidades de movimentos, procurando um que deixe o oponente em uma má posição.
- O trabalho de determinar se uma posição em particular é ruim pode ser deixada para o predicado “`e_posicao_ruim`”.
- Esses dois subprogramas chamam a si mesmos várias vezes (recursão mútua) avaliando todas as possibilidades a medida que o jogo executa.





# Backtracking para “nim games”

- Em pseudocódigo:

```
int achar_bom_movimento (int moedas)
{
    for ( /* cada um dos movimentos possíveis */ )
    {
        // Avalie a posição que resulta em fazer esse movimento.
        // Se a posição for ruim (para o oponente), retorne esse movimento.
    }
    // Retorne um sentinela indicando que não há nenhum bom movimento.
}
```

- Os valores legais que a função `achar_bom_movimento` pode retornar são 1, 2 ou 3 (a quantidade de moedas a retirar que forma um bom movimento). Se não houver nenhum bom movimento, retorna um sentinela (-1).

# Backtracking para “nim games”

- Em pseudocódigo:

```
bool e_posicao_ruim (int moedas)
{
    if ( /* só temos 1 moeda */ )
        return TRUE;

    return ( /* Não há mais movimentos bons? */ )
}
```

- Interpretação:
  - Ter 1 só moeda é uma má posição
  - Uma posição é ruim se não houver mais movimentos bons

# Backtracking para “nim games”

```
211 /**
212  * Função: achar_bom_movimento
213  * Uso: retirar = achar_bom_movimento(moedas);
214  * -----
215  * Esta função busca um movimento vencedor, dado o número especificado de moedas
216  * através do argumento de mesmo nome. Se existir um movimento vencedor nessa
217  * posição, a função retorna esse valor; se não houver um movimento vencedor, a
218  * função retorna a constante SEM_BOM_MOVIMENTO. Esta função depende do insight
219  * recursivo de que:
220  *     - Um bom movimento é aquele que deixe o oponente em uma má posição;
221  *     - Uma má posição é aquela onde não existem bons movimentos.
222  */
223
224 static int achar_bom_movimento (int moedas)
225 {
226     int retirar;
227
228     for (retirar = RETIRADA_MINIMA; retirar <= RETIRADA_MAXIMA; retirar++)
229     {
230         if (e_posicao_ruim(moedas - retirar))
231             return retirar;
232     }
233     return SEM_BOM_MOVIMENTO;
234 }
```

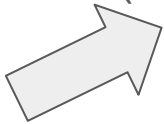
# Backtracking para “nim games”

```
236 /**
237  * Predicado: e_posicao_ruim
238  * Uso: if (e_posicao_ruim(moedas)) . . .
239  * -----
240  * Este predicado retorna TRUE se a quantidade de moedas indicada pelo argumento
241  * for uma má posição. Uma má posição é aquela na qual não há bons movimentos
242  * possíveis. Ser deixado com 1 única moeda também é, claramente, uma má posição
243  * e representa o caso simples da recursão.
244  */
245
246 static bool e_posicao_ruim (int moedas)
247 {
248     if (moedas == 1)
249         return TRUE;
250     return (achar_bom_movimento(moedas) == SEM_BOM_MOVIMENTO);
251 }
```

# Backtracking para “nim games”

- O caso simples está bem claro no predicado “**e\_posicao\_ruim**”.
- **Onde está a decomposição recursiva???**

(estude o código completo que está no site da disciplina)



# Backtracking generalizado para jogos com 2 jogadores

- O código que vimos para o jogo de nim funciona, mas tem um **problema** fundamental: **incorpora detalhes específicos do jogo dentro da estrutura do código**, o conhecimento de que o computador retirará 1, 2 ou 3 moedas. Isso torna o algoritmo menos genérico.
- Muitos jogos com 2 jogadores podem ser solucionados com a mesma estratégia, mesmo que jogos diferentes necessitem de detalhes de código diferentes.
- É possível generalizar essa estratégia? Sim, com **abstração**, que é o processo de separar os aspectos gerais de um problema para que eles não sejam mais obscurecidos pelos detalhes de um domínio específico.

# Backtracking generalizado para jogos com 2 jogadores

- Precisamos de um programa que seja geral o suficiente para ser adaptado facilmente para jogar nim, jogo da velha, ou qualquer outro jogo de estratégia entre 2 jogadores que você quiser.
- Como criar esse programa abstrato geral?
  - **1º passo**: reconhecer os **conceitos comuns** a todos esses jogos de estratégia entre 2 jogadores.
    - **ESTADO**: alguma coleção de dados que definirá exatamente o que está ocorrendo no jogo em qualquer instante no tempo. Qualquer que seja o jogo, sempre será possível coletar todos os dados relevantes do estado em um registro.
    - **MOVIMENTO**: representa o que o jogador fez, e também pode ser representado por um registro com as informações relevantes.

# Backtracking generalizado para jogos com 2 jogadores

- Como criar esse programa abstrato geral?
  - **2º passo**: abstrair esses conceitos comuns (estado e movimento) definindo-os como tipos gerais com nomes como `estadoT` ou `movimentoT`, que transcenderão os detalhes específicos de qualquer jogo.
    - A estrutura interna desses tipos será diferente para jogos diferentes, mas nosso algoritmo será abstrato e poderá se referir a eles de forma genérica!
- Analise o programa a seguir:



# Backtracking generalizado para jogos com 2 jogadores

```
int main (void)
{
    estadoT estado;
    movimentoT movimento;

    dar_instrucoes();
    estado = novo_jogo();
    while (!jogo_terminou(estado))
    {
        mostrar_jogo(estado);
        switch (vez_de_quem(estado))
        {
            case Humano:
                movimento = obter_movimento_do_jogador(estado);
                break;
            case Computador:
                movimento = escolher_movimento_do_computador(estado);
                mostrar_movimento(movimento);
                break;
        }
        fazer_movimento(estado, movimento);
    }
    anunciar_resultado(estado);
}
```

# Backtracking generalizado para jogos com 2 jogadores

```
int main (void)
{
    estadoT estado;
    movimentoT movimento;

    dar_instrucoes();
    estado = novo_jogo();
    while (!jogo_terminou(estado))
    {
        mostrar_jogo(estado);
        switch (vez_de_quem(estado))
        {
            case Humano:
                movimento = obter_movimento_do_jogador(estado);
                break;
            case Computador:
                movimento = escolher_movimento_do_computador(estado);
                mostrar_movimento(movimento);
                break;
        }
        fazer_movimento(estado, movimento);
    }
    anunciar_resultado(estado);
}
```

Que jogo é esse???

# Backtracking generalizado para jogos com 2 jogadores

```
int main (void)
{
    estadoT estado;
    movimentoT movimento;

    dar_instrucoes();
    estado = novo_jogo();
    while (!jogo_terminou(estado))
    {
        mostrar_jogo(estado);
        switch (vez_de_quem(estado))
        {
            case Humano:
                movimento = obter_movimento_do_jogador(estado);
                break;
            case Computador:
                movimento = escolher_movimento_do_computador(estado);
                mostrar_movimento(movimento);
                break;
        }
        fazer_movimento(estado, movimento);
    }
    anunciar_resultado(estado);
}
```

**Qualquer um! Pode ser xadrez,  
nim, jogo da velha...**

Cada jogo terá sua própria definição de estado, movimento, etc., mas a implementação é geral!

# Backtracking generalizado para jogos: minimax

- Nosso programa “main” já está generalizado o suficiente, serve para qualquer jogo de estratégia entre 2 jogadores. Mas isso é o de menos: o importante é achar uma estratégia efetiva para vencer o jogo. A função “escolher\_movimento\_do\_computador” deverá chamar outra, a função que realmente implementa a estratégia, chamada de “encontrar\_melhor\_movimento”. Ela retorna o melhor movimento.
- **O melhor movimento em qualquer posição é simplesmente aquele que deixa seu oponente em uma posição pior.**
- **A pior posição é aquela que oferece o melhor movimento mais fraco.**

# Backtracking generalizado para jogos: minimax

- Essa idéia, **encontrar a posição que deixe seu oponente com o pior “melhor movimento possível”** é chamada de estratégia **minimax**, pois o objetivo dessa estratégia é **minimizar a oportunidade máxima de seu oponente**.
- A melhor maneira de entender a estratégia minimax é **pensar nos movimentos futuros** do jogo e criar uma **árvore de movimentos** onde cada nó da árvore representa um estado do jogo. O estado inicial é apenas esse:



# Backtracking generalizado para jogos: minimax

- Agora considere que você fará o 1º movimento. Se existirem 3 movimentos possíveis, você pode escolher um desses 3 caminhos, que nos levarão a novos estados do jogo:



# Backtracking generalizado para jogos: minimax

- Agora é o seu oponente que fará a jogada. Se ele também tiver 3 opções de caminhos para cada um desses estados, ele pode escolher entre esses movimentos abaixo:



- Sabendo disso, a pergunta agora é: Qual será seu 1º movimento? Será aquele que deixar seu oponente com menor chance de vencer.

# Backtracking generalizado para jogos: minimax

- Para identificarmos qual movimento deixará seu oponente com a menor chance de vencer, temos que adicionar uma pontuação a cada possível movimento. Quanto maior a pontuação, maior a chance de vencer.



- Também podemos calcular a pontuação possível para cada posição (estado) do jogo.



# Backtracking generalizado para jogos: minimax

- A pontuação dos movimentos e do estado é calculada da perspectiva do jogador que fará o próximo movimento.



- Um último detalhe: as pontuações devem ser simétricas ao redor do 0 (zero), ou seja, uma pontuação de +9 para um jogador será de -9 para o outro.

# Backtracking generalizado para jogos: minimax

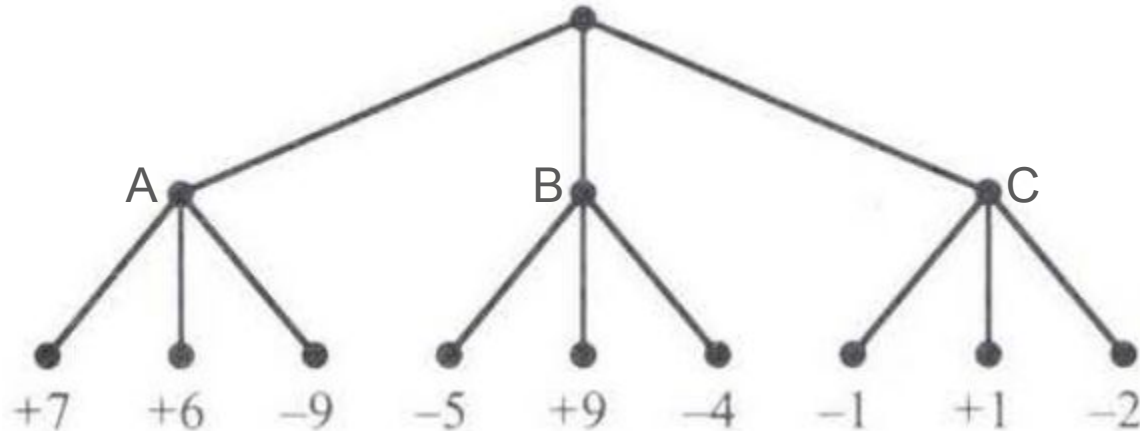
- A interpretação das pontuações captura a idéia de que uma posição que é boa para um jogador é má para o oponente, e nos permite expressar relações entre as pontuações dos movimentos e das posições.



- A pontuação para qualquer movimento é o oposto da pontuação que seria obtida pelo seu oponente. A pontuação de qualquer posição é a pontuação de seu melhor movimento.

# Backtracking generalizado para jogos: minimax

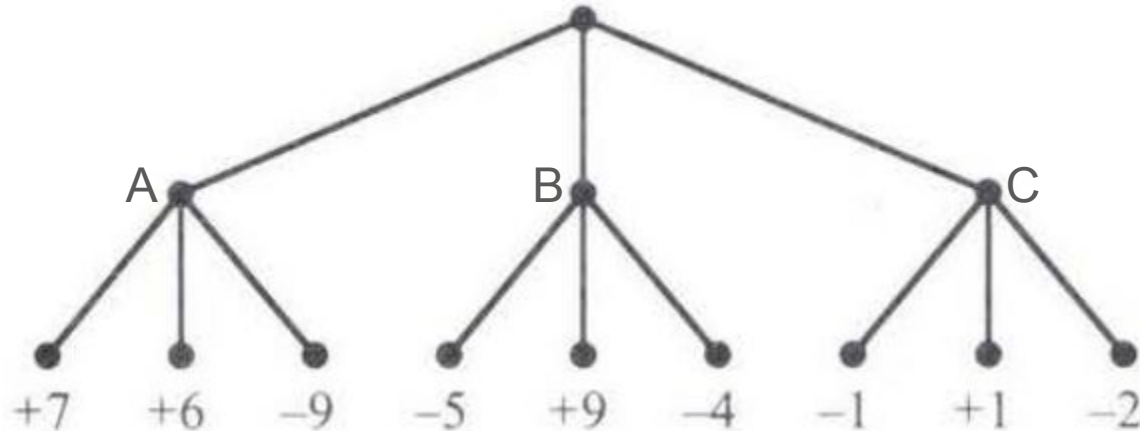
- Suponha que você analisou 2 passos à frente, antes de fazer sua primeira jogada: o seu próprio movimento e as respostas possíveis do oponente:



- E agora? Qual seria o seu primeiro movimento? A, B ou C?

# Backtracking generalizado para jogos: minimax

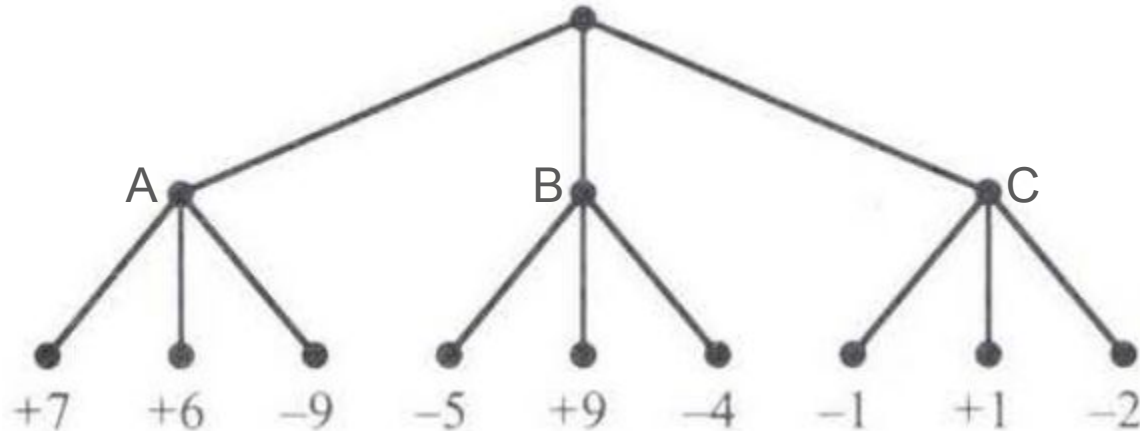
- A jogada “A” te leva para a posição com a pontuação final mais positiva (+4) e a jogada “B” pode te levar a uma posição final individual com +9!



- E agora? Qual seria o seu primeiro movimento? A, B ou C?

# Backtracking generalizado para jogos: minimax

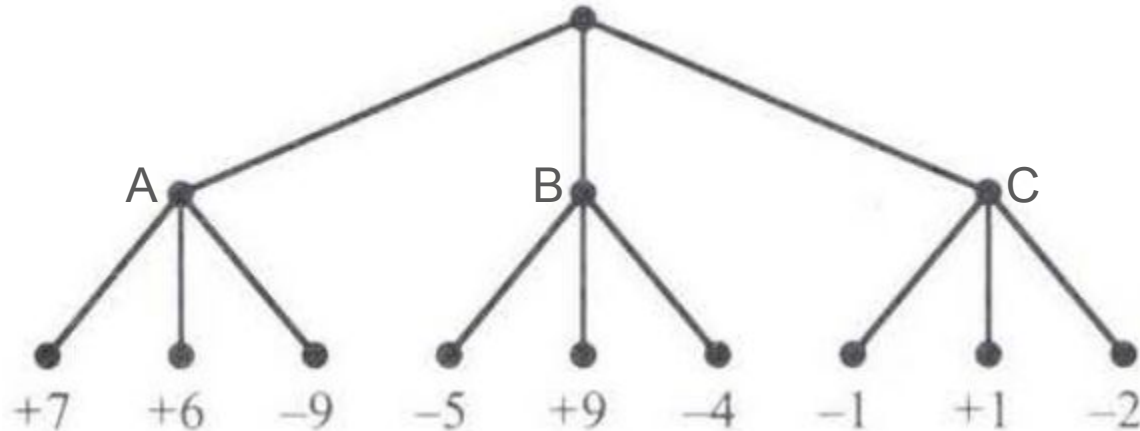
- Na verdade, nenhuma dessas considerações anteriores importam, pois o seu adversário irá escolher o próximo movimento!



- Se você escolher “A”, o oponente te deixará em -9; se você escolher “B”, seu adversário te deixará em -5.

# Backtracking generalizado para jogos: minimax

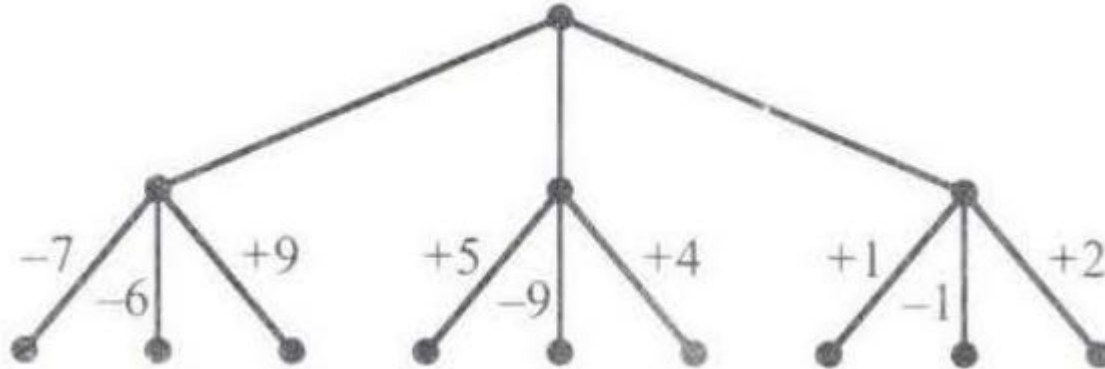
- O melhor que você tem a fazer é escolher a jogada “C”, que permite no máximo que seu oponente te deixe em -2.



- Não é o ideal, mas é a melhor alternativa. Perceba: você **minimizou a oportunidade máxima de seu adversário**, que eram de: -9, -5 ou -2!

# Backtracking generalizado para jogos: minimax

- A pontuação de um movimento, da perspectiva do jogador que está fazendo o movimento, é o negativo da pontuação da posição resultante. Assim, **do ponto de vista do seu adversário**, temos:



- Seu oponente tentará fazer a jogada melhor para ele. Ao escolher a jogada “C”, você **minimiza a pontuação máxima disponível para seu oponente**.

# Backtracking: implementação do algoritmo minimax

- O algoritmo minimax é bem geral e pode ser implementado de uma maneira que não dependa das características específicas de um determinado jogo.
- Em geral um algoritmo minimax consiste de **2 funções mutuamente recursivas**:
  - Uma **encontra o melhor movimento**
  - A outra **avalia a qualidade de uma posição**
- Além disso deve ser possível:
  - **Limitar a profundidade da busca recursiva** (por quê?)
  - **Atribuir pontuações para movimentos e para posições**, por exemplo:
    - Usar números inteiros negativos e positivos, sendo o 0 uma pontuação neutra;
    - A maior pontuação positiva indica uma posição que o jogador invariavelmente vence
    - A pontuação mais negativa indica uma posição que o jogador invariavelmente perde



# Backtracking: implementação do algoritmo minimax

```
static movimentoT encontrar_melhor_movimento (estadoT estado, int profundidade,  
                                              int *ptr_pontuacao);  
static int avaliar_posicao (estadoT estado, int profundidade);
```

# Backtracking: implementação do algoritmo minimax

```
static movimentoT encontrar_melhor_movimento (estadoT estado, int profundidade,
                                              int *ptr_pontuacao)
{
    for ( /* cada movimento possível ou até que você encontre a vitória */ )
    {
        // Faça o movimento
        // Avalie a posição resultado, incrementando o indicador de profundidade
        // Registre a pontuação mínima até então, junto com o movimento
        // Desfaça o movimento para restaurar o estado original
    }
    // Armazene a pontuação do movimento usando o ponteiro fornecido
    // Retorne o melhor movimento encontrado
}
```

# Backtracking: implementação do algoritmo minimax

```
248 static movimentoT encontrar_melhor_movimento (estadoT estado, int profundidade,
249                                               int *ptr_pontuacao)
250 {
251     movimentoT arr_movimentos[MAX_MOVIMENTOS], movimento, melhor_movimento;
252     int i, n_movimentos, pontuacao, pontuacao_minima;
253
254     n_movimentos = gerar_lista_de_movimentos(estado, arr_movimentos);
255     if (n_movimentos == 0) Error("Nenhum movimento disponível");
256     pontuacao_minima = POSICAO_VENCEDORA + 1;
257     for (i = 0; i < n_movimentos && pontuacao_minima != POSICAO_PERDEDORA; i++)
258     {
259         movimento = arr_movimentos[i];
260         fazer_movimento(estado, movimento);
261         pontuacao = avaliar_posicao(estado, profundidade + 1);
262         if (pontuacao < pontuacao_minima)
263         {
264             melhor_movimento = movimento;
265             pontuacao_minima = pontuacao;
266         }
267         desfazer_movimento(estado, movimento);
268     }
269     *ptr_pontuacao = -pontuacao_minima;
270     return melhor_movimento;
271 }
```

# Backtracking: implementação do algoritmo minimax

- A função “`gerar_lista_de_movimentos`” é implementada separadamente para cada jogo e causa o preenchimento dos elementos no array “`arr_movimentos`”, com a lista dos movimentos válidos na posição atual. Essa função retorna o número de movimentos disponíveis.
- A linha  

```
pontuacao_minima = POSICAO_VENCEDORA + 1;
```

inicializa a pontuação mínima em um número grande o suficiente para garantir que esse valor será modificado no primeiro ciclo do loop for.

# Backtracking: implementação do algoritmo minimax

- A linha

```
*ptr_pontuacao = -pontuacao_minima;
```

armazena a pontuação do melhor movimento na variável fornecida pelo cliente (através do ponteiro).

O sinal negativo é necessário porque a perspectiva foi alterada: as posições foram avaliadas do ponto de vista de seu oponente, mas as pontuações expressam o valor de um movimento a partir do seu próprio ponto de vista. Um movimento que deixa seu oponente em uma posição negativa é bom para você e portanto tem um valor positivo.

# Backtracking: implementação do algoritmo minimax

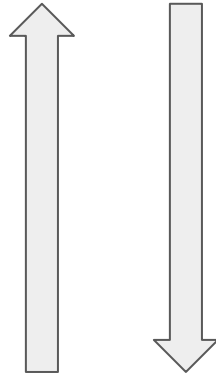
```
282 static int avaliar_posicao (estadoT estado, int profundidade)
283 {
284     int pontuacao;
285
286     if (jogo_terminou(estado) || profundidade >= PROFUNDIDADE_MAX)
287         return avaliar_posicao_estatica(estado);
288
289     (void) encontrar_melhor_movimento(estado, profundidade, &pontuacao);
290     return pontuacao;
291 }
```

# Backtracking: implementação do algoritmo minimax

- Os casos simples que terminarão a recursão são:
  - O jogo terminou
  - A profundidade máxima da recursão foi atingida
- Nos casos simples o programa deve avaliar a posição sem fazer outras chamadas recursivas, o que é feito com a nova função “`avaliar_posicao_estatica`” (que é criada separadamente para cada jogo).
- Na decomposição recursiva, a pontuação de uma posição é simplesmente a pontuação da melhor movimentação possível dado o estado atual e, por isso, chamamos novamente a função `encontrar_melhor_movimento`.

# Backtracking: implementação do algoritmo minimax

`encontrar_melhor_movimento`

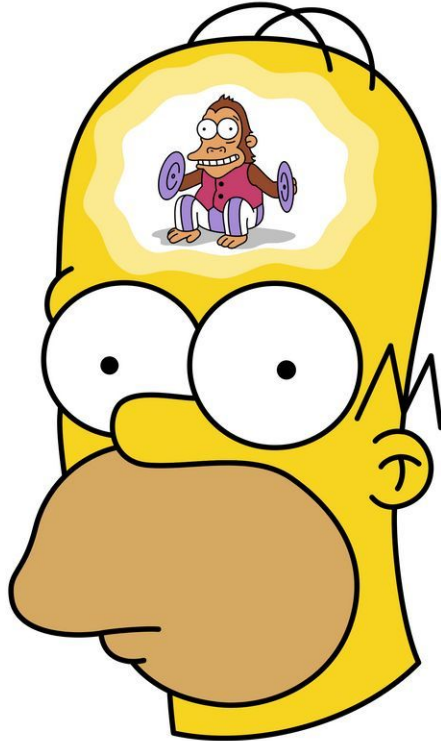


`avaliar_posicao`

**Note que são funções abstratas, estão implementadas de modo a não depender dos detalhes de nenhum jogo!**



# Backtracking: implementação do algoritmo minimax



Na melhor das hipóteses, seu cérebro está assim agora...

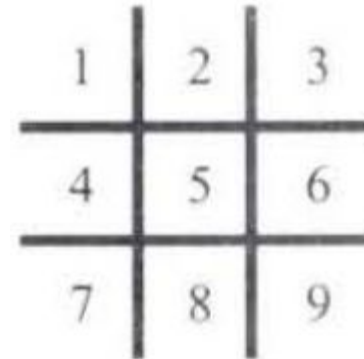
Infelizmente não há mais nada que se possa fazer, **VOCÊ DEVE SE ESFORÇAR POR CONTA PRÓPRIA PARA ENTENDER.** Faça isso até conseguir!

# Backtracking com minimax: jogo da velha

- Uma vez que você tenha as funções “encontrar\_melhor\_movimento” e “avaliar\_posicao”, você já tem o que precisa para o minimax:
  - Essas funções resolvem o trabalho conceitualmente difícil de encontrar o melhor movimento a partir de uma determinada posição;
  - Como são abstratas não dependem dos detalhes de nenhum jogo em particular
- Para criar um novo jogo de estratégia entre 2 jogadores, basta resolver o problema de projetarmos as estruturas `estadoT` e `movimentoT` adequadas para o jogo que estamos criando, e escrever códigos para funções que devem ser específicas de cada jogo. Mas esse é um trabalho mais “braçal” pois a parte intelectual complicada (minimax) já está resolvida.

# Backtracking com minimax: jogo da velha

- Para o Jogo da Velha, as funções MAIN, AVALIAR\_POSIÇÃO e ENCONTRAR\_MELHO\_MOVIMENTO são completamente independentes dos detalhes do Jogo da Velha e implementam o minimax.
- O que torna o jogo um jogo de “Jogo da Velha” são as definições dos tipos e funções que auxiliam esse “framework” básico que implementa o minimax.
- Os tipos `movimentoT` e `estadoT` são definidos de forma apropriada para o Jogo da Velha. Se o “tabuleiro” for representado como na figura ao lado, um movimento é um número inteiro que representa o quadrado a ser marcado. `typedef int movimentoT;`



1	2	3
4	5	6
7	8	9

# Backtracking com minimax: jogo da velha

- Já o `estadoT` pode representar o tabuleiro como um array de caracteres onde os “X” e os “O” serão marcados nas posições indicadas pelo números inteiros. O tabuleiro pode ser representado por uma array unidimensional.

```
struct st_estadoT
{
    char tabuleiro[3 * 3 + 1];
    jogadorT jogador;
    int numero_jogadas_feitas;
};

typedef struct st_estadoT *estadoT;
```

1	2	3
4	5	6
7	8	9

# Backtracking com minimax: jogo da velha

- Agora já temos a representação dos movimentos e do estado do jogo. Mas e as regras do Jogo da Velha?
- Estão implementadas em outros subprogramas, aí sim, específicos para o Jogo da Velha, tais como: “`gerar_lista_de_movimentos`”, “`movimento_e_valido`”, “`avaliar_posicao_estatica`”. Mesmo que esses subprogramas precisem de esforço para codificar, são conceitualmente muito menos complexos que o minimax.
- Codificar as funções do minimax uma única vez e reutilizá-las em diferentes jogos evitam termos que recriar a mesma lógica múltiplas vezes.

# Backtracking com minimax: jogo da velha

**Estude o código do Jogo da Velha** que está no portal da disciplina, várias e várias vezes, até entender como esse código funciona!!!

1	2	3
4	5	6
7	8	9

# Algoritmos de Backtracking: resumo

- Algoritmos de backtracking são úteis para resolver problemas que envolvem a tomada de uma seqüência de decisões (escolhas) na busca de um determinado objetivo.
- A estratégia básica é escrever algoritmos que possam retroceder (backtrack) para pontos prévios de decisão se as escolhas atuais não forem corretas.
- Confie no salto recursivo.
- Evite codificar explicitamente os detalhes do processo de backtracking e desenvolva soluções gerais que possam se aplicar a diversos problemas.

# Algoritmos de Backtracking: resumo

- A maioria dos problemas de backtracking podem ser solucionadas com a seguinte abordagem recursiva:

```
// Se você já está na solução, retorne sucesso.  
for ( /* cada escolha possível a partir da posição atual */ )  
{  
    // Faça a escolha e "ande" um passo ao longo do caminho.  
    // Use recursividade para resolver o problema a partir da nova posição.  
    // Se a chamada recursiva for um sucesso, retorne para nível anterior.  
    // Retroceda a partir da escolha atual para retornar ao estado original.  
}  
// Retorne que falhou.
```



# Algoritmos de Backtracking: resumo

- As chamadas recursivas em problemas de backtracking são muito complexas para rastrear em detalhe no stack. Aceite o salto de fé!
- Jogos de estratégia entre 2 jogadores podem geralmente ser solucionados com backtrack recursivo. Como o objetivo em tais jogos envolve minimizar a oportunidade máxima do seu oponente vencer, a estratégia convencional é o algoritmo minimax.
- É possível codificar o algoritmo minimax de forma geral e manter os detalhes específicos de um jogo separadas da implementação do minimax em si. Isso torna possível adaptar um programa minimax existente para novos jogos!

# Algoritmos de Backtracking:

- Não se engane: nesse momento **VOCÊ NÃO ENTENDEU** como backtracking funciona, ainda mais se estiver utilizando um algoritmo de backtracking recursivo com minimax.
- A única maneira de aprender é **ESTUDAR OS CÓDIGOS** até que você tenha um insight (até a “ficha cair”). Isso pode demorar dias, semanas, meses ou anos, mas eventualmente você vai entender. Não desanime!