

Fontes principais

1. Cormen T. H.; Leiserson C. E.; Rivest R.; Stein C.. *Introduction to Algorithms*, 3^a edição, MIT Press, 2009
2. Análise de algoritmo - IME/USP (prof. Paulo Feofiloff)
http://www.ime.usp.br/~pf/analise_de_algoritmos

Backtracking

algoritmos de enumeração

Backtracking

É um refinamento da técnica de força bruta.

Considera uma série de tomadas de decisões entre várias opções.

Algumas consequências de decisões podem conduzir a uma solução do problema.

Backtracking

Ideia geral:

- (1) Soluções representadas por n tuplas ou vetores de solução (s_1, s_2, \dots, s_n) de maneira que cada s_i é escolhido a partir de um conjunto finito de opções v_i ;
- (2) Inicia um vetor vazio;

Backtracking

- (3) Em cada nova etapa do vetor é estendido com um novo valor;
- (4) O novo valor deve ser avaliado: se não for solução parcial, então o último valor do vetor é eliminado e outro candidato é considerado.

Note que em (4) pode ou não ocorrer retrocesso (backtracking).

Backtracking

Restrições

- Restrições explícitas:
Correspondem às regras que restringem cada s_i em tomar valores de um determinado conjunto (relacionado ao problema e escolhas possíveis).
- Restrições implícitas:
Determinam como os s_i 's se relacionam entre si.

Algoritmo de backtracking genérico

backtrack($s[1 \dots k]$)

- 1 ▷ s é um vetor promissor de tamanho k
- 2 **se** s é a solução **então**
- 3 escreva(s)
- 4 **senão**
- 5 **para** cada vetor promissor w de tamanho $k + 1$ **faça**
- 6 backtracking($w[1 \dots k + 1]$)
- 7 ▷ $w[1 \dots k] = s[1 \dots k]$

Sequências

Sequências

Suponha o seguinte problema:

Gerar todas as sequências possíveis de 3 dígitos com os dígitos 1,2 e 3 armazenados no vetor $v[0 \dots n - 1]$, onde $n = 3$

Solução: 111, 112, 113, 121, 122, 123, 131, 132, 133, 211, 212, 213, 221, 222, 223, 231, 232, 233, 311, 312, 313, 321, 322, 323, 331, 332, 333

A quantidade é de $3^3 = 27$ sequências.

Sequências

algoritmo `mostra_sequencias(s, i, v, n)`

entrada: vetor solução: $s[0 \dots n - 1]$

inteiro $i = 0$ que controla a profundidade da recursão,

vetor de entrada: $v[0 \dots n - 1]$

inteiro n o tamanho do vetor v

saída: apresenta todas as sequências de n elemntos de v

Sequências

`mostra_sequencias(s, i, v, n)`

1 **se** $i = n$ **então**

2 `escreva(s, n)`

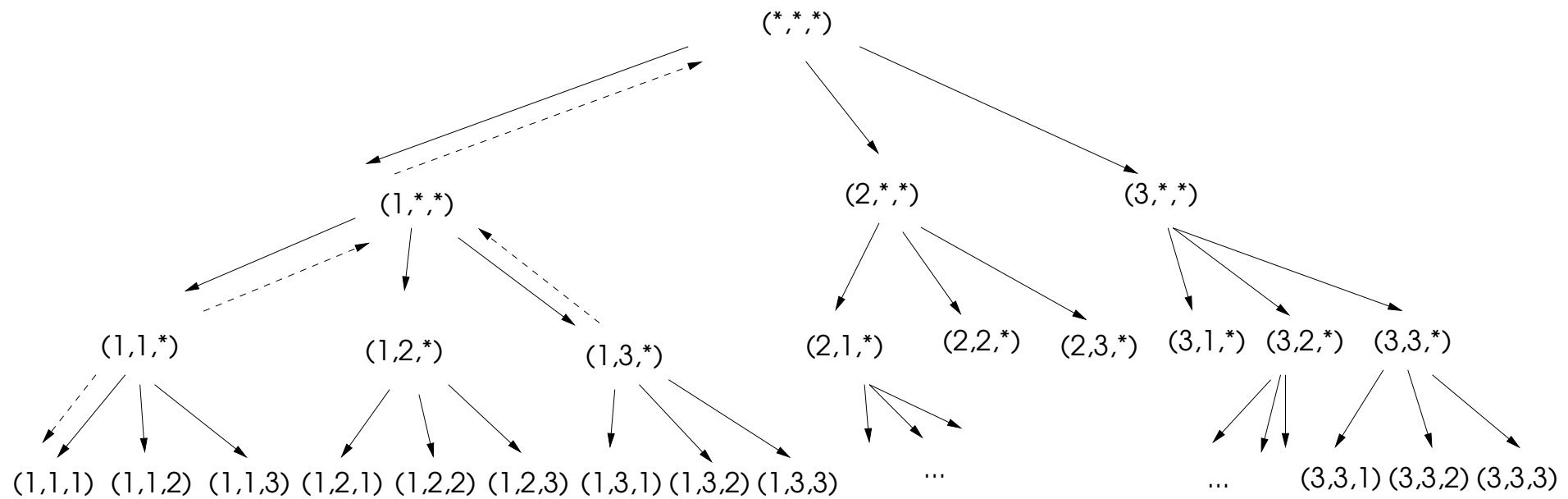
3 **senão**

4 **para** $j = 0$ **até** $n - 1$ **faça**

5 $s[i] = v[j]$

6 `mostra_sequencias($s, i + 1, v, n$)`

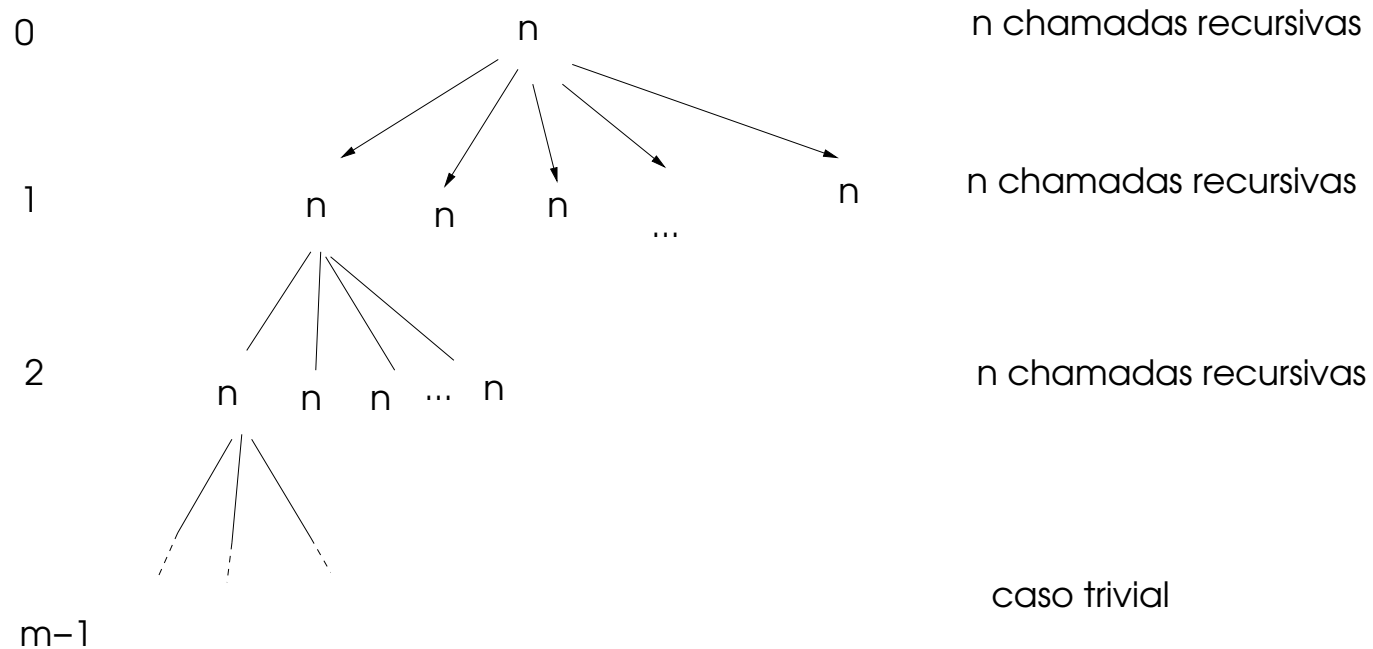
Sequências



Sequências

Suponha que agora queremos gerar todas as combinações de tamanho 3 utilizando os dígitos $v = \{1, 2, 3, 4\}$. Nesta caso, temos que definir $s[0 \cdots m - 1]$, com $m = 3$, e o vetor $v[0 \cdots n - 1]$, com $n = 4$.

Sequências



Complexidade do algoritmo: $O(n^m)$

Permutações

Permutações

Uma permutação da sequência $1 \cdots n$ é qualquer rearranjo dos termos dessa sequência. Em outras palavras, uma permutação da sequência $1 \cdots n$ troca os elementos de lugar gerando uma nova sequência com estes mesmos elementos.

Permutações - Exemplo

Dado uma vetor v , escreva um programa que imprima todas as permutações. Se v tem n elementos, então o programa deve imprimir $n!$ permutações.

entrada: $v = \{1, 2, 3\}$

saída: $\{1, 2, 3\}, \{1, 3, 2\}, \{2, 1, 3\},$
 $\{2, 3, 1\}, \{3, 1, 2\}, \{3, 2, 1\}.$

Permutações

algoritmo `permuta(s, i, v, n, x)`

entrada: vetor solução: $s[0 \dots n - 1]$

inteiro $i = 0$ que controla a profundidade da recursão,

vetor de entrada: $v[0 \dots n - 1]$

inteiro n o tamanho do vetor a

vetor de marcação: $x[0 \dots n - 1]$ marca os elementos que já foram inseridos no conjunto solução.

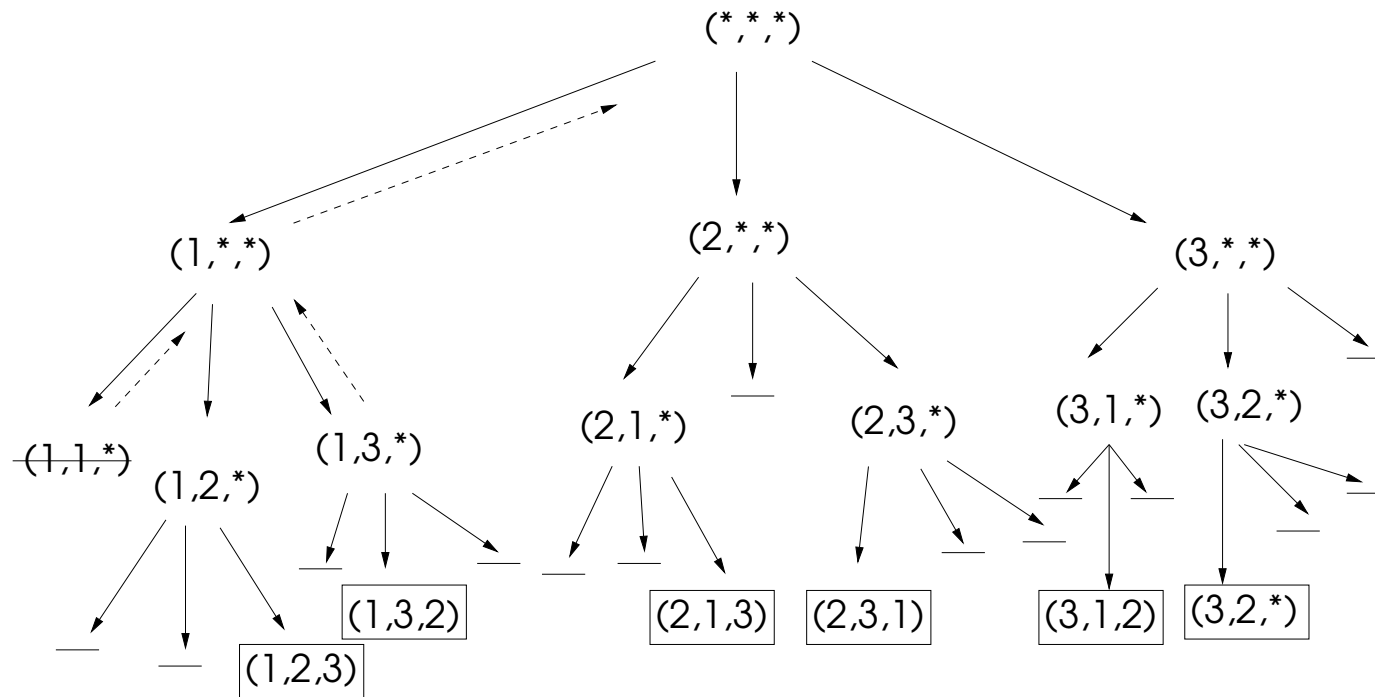
saída: apresenta a permutação dos elementos do vetor a

Permutações

`permuta(s, i, v, n, x)`

```
1  se  $i = n$  então  
2      escreva( $s, n$ )  
3  senão  
4      para  $j = 0$  até  $n - 1$  faça  
5           $\triangleright$  verifica se a solução é promissora  
6          se  $x[j] == 0$  então  
7               $x[j] = 1$   $\triangleright$  inclui na solução  
8               $s[i] = v[j]$   
9              permuta( $s, i + 1, v, n, x$ )  
10              $x[j] = 0$ 
```

Permutações



Permutações

Restrições explícitas: O conjunto solução s tem uma permutação ao chegar ao caso base.

Restrições explícitas: Na permutação não há repetição de elementos.

Sudoku

Sudoku

É um jogo (quebra-cabeça) com foco no posicionamento de números inteiros.

Cada coluna, linha e região só deve ter um único valor de $\{1, 2, \dots, n\}$.

O problema presente no Sudoku é NP-Completo.

Sudoku simples

Tomemos $n = 4$

	2	4	
1			3
4			2
	1	3	

Solução:

- 4-upla (v_1, v_2, v_3, v_4) em que cada v_i é uma linha.
- Restrição explícita:
$$S_i = \{1, 2, 3, 4\}, \quad 1 \leq i \leq 4$$
- Restrição implícita:
Um número pode aparecer apenas uma vez em cada linha e coluna.

Sudoku simples

$$n = 4$$

	2	4	
1			3
4			2
	1	3	

Vamos construir a solução posição por posição.

Sudoku simples

$$n = 4$$

1	2	4	
1			3
4			2
	1	3	

Tente colocar o número 1 no primeiro espaço vazio.

Sudoku simples

$$n = 4$$

1	2	4	
1			3
4			2
	1	3	

Não é possível! O 1 já se encontra na coluna.

Sudoku simples

$$n = 4$$

2	2	4	
1			3
4			2
	1	3	

Tente o número 2.

Sudoku simples

$$n = 4$$

2	2	4	
1			3
4			2
	1	3	

Não é possível! O 2 já se encontra na linha.

Sudoku simples

$$n = 4$$

3	2	4	
1			3
4			2
	1	3	

Tente o número 3.

Sudoku simples

$$n = 4$$

3	2	4	
1			3
4			2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1			3
4			2
	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1			3
4			2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	1		3
4			2
	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1	1		3
4			2
	1	3	

Não é possível! Pois já existe o número 1 na linha. Agora, tente o número 2.

Sudoku simples

$$n = 4$$

3	2	4	1
1	2		3
4			2
	1	3	

Não é possível! Pois já existe o número 2 na coluna. Agora, tente o número 3.

Sudoku simples

$$n = 4$$

3	2	4	1
1	3		3
4			2
	1	3	

Não é possível! Pois já existe o número 3 na linha. Agora, tente o número 4.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4		3
4			2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	1	3
4			2
	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	1	3
4			2
	1	3	

Não é possível! Pois já existe o número 1 na linha. Agora, tente o número 2.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4			2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	1		2
	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	1		2
	1	3	

Não é possível! Pois já existe o número 1 na coluna. Agora, tente o número 2.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	2		2
	1	3	

Não é possível! Pois já existe o número 2 na linha. Agora, tente o número 3.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3		2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
1	1	3	

Comece pelo 1.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
1	1	3	

Não é possível! Pois já existe o número **1** na linha e na coluna.
Agora, tente o número 2.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
2	1	3	

Esta é uma solução parcial promissora! Agora tentaremos preencher a próxima posição vazia.

Sudoku simples

$$n = 4$$

3	2	4	1
1	4	2	3
4	3	1	2
2	1	3	4

Só é possível colocar o número 4.

Sudoku simples

```
void sudoku(int m[4][4], int linha, int coluna) {  
    if (coluna == 4){  linha = linha + 1;  coluna = 0; }  
    if (linha == 4) mostra_matriz(m);  
    else if (m[1][c] > 0) sudoku(m, linha, coluna + 1);  
    else {  
        for (int i = 1; i <= 4; i++){  
            if (valor_valido(i, m, linha, coluna)){  
                m[1][c] = i;  
                sudoku(m, linha, coluna + 1);  
            }  
        }  
    }  
}  
} // fim
```

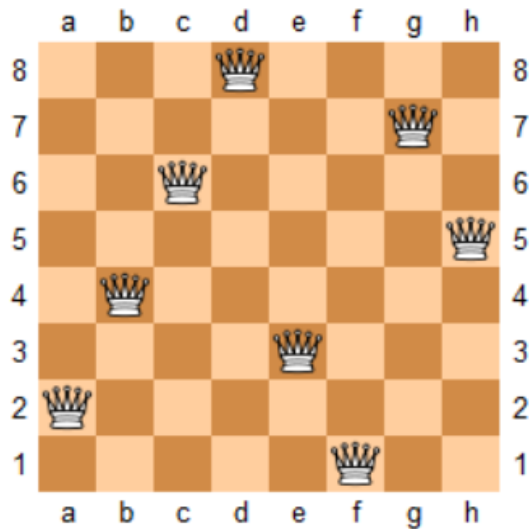
Sudoku simples

```
int valor_valido(int valor, int m[4][4], int linha, int coluna) {  
    for (int i = 0; i < 4; i++) {  
        if (m[linha][i] == valor || m[i][coluna] == valor)  
            return 0;  
    }  
    return 1;  
}
```

Oito rainhas

Oito rainhas

Oito rainhas devem ser dispostas num tabuleiro de xadrez de tal modo que nenhuma delas seja atacada por outra.



Oito rainhas

Utilizamos a técnica de backtracking, que consiste em, a cada passo, caso a solução parcial encontrada seja inválida, voltar ao passo anterior, reformulando-o.

Oito rainhas

posiciona_rainha(tabuleiro, coluna)

```
1      se coluna == N então mostra_matriz(tabuleiro)
2      senão
3          para i = 0 até N faça
4              se "rainha está salva" então
5                  tabuleiro[i][coluna] = 'Q'
6                  posiciona_rainha(tabuleiro, i + 1)
7                  tabuleiro[i][coluna] = '—'
```

Simulação com quatro rainhas

Q_1			

Selecione uma posição. A rainha Q_1 está salva.
Agora vamos tentar incluir mais uma rainha.

Simulação com quatro rainhas

Q_1	Q_2		

Selecione uma posição. A rainha Q_2 não está salva.
Procure outra posição para Q_2 .

Simulação com quatro rainhas

Q_1			
	Q_2		

A rainha Q_2 não está salva.
Procure outra posição para Q_2 .

Simulação com quatro rainhas

Q_1			
	Q_2		

A rainha Q_2 está salva.

Agora vamos tentar incluir mais uma rainha.

Simulação com quatro rainhas

Q_1		Q_3	
	Q_2		

Selecione uma posição. A rainha Q_3 não está salva.
Procure outra posição para Q_3 .

Simulação com quatro rainhas

Q_1			
		Q_3	
	Q_2		

A rainha Q_3 não está salva.
Procure outra posição para Q_3 .

Simulação com quatro rainhas

Q_1			
	Q_2	Q_3	

A rainha Q_3 não está salva.
Procure outra posição para Q_3 .

Simulação com quatro rainhas

Q_1			
	Q_2		
		Q_3	

A rainha Q_3 não está salva.

Não há mais posição válida para Q_3 .

Simulação com quatro rainhas

Q_1			
	Q_2		

Retrocede e reposicione Q_2 .

Agora tente novamente para Q_3 .

Simulação com quatro rainhas

Q_1		Q_3	
	Q_2		

A rainha Q_3 não está salva.
Procure outra posição para Q_3 .

Simulação com quatro rainhas

Q_1			
		Q_3	
	Q_2		

A rainha Q_3 está salva.

Agora vamos tentar incluir mais uma rainha.

Simulação com quatro rainhas

Q_1			Q_4
		Q_3	
	Q_2		

A rainha Q_4 não está salva.
Procure outra posição para Q_4 .

Simulação com quatro rainhas

Q_1			
		Q_3	Q_4
	Q_2		

A rainha Q_4 não está salva.
Procure outra posição para Q_4 .

Simulação com quatro rainhas

Q_1			
		Q_3	
			Q_4
	Q_2		

A rainha Q_4 não está salva.
Procure outra posição para Q_4 .

Simulação com quatro rainhas

Q_1			
		Q_3	
	Q_2		Q_4

A rainha Q_4 não está salva.

Não há mais posição válida para Q_4 .

Simulação com quatro rainhas

Q_1			
		Q_3	
	Q_2		

Retrocede e reposicione Q_3 .

Simulação com quatro rainhas

Q_1			
		Q_3	
	Q_2		

A rainha Q_3 não está salva.
Procure outra posição para Q_3 .

Simulação com quatro rainhas

Q_1			
	Q_2	Q_3	

A rainha Q_3 não está salva.

Não há mais posição válida para Q_3 .

Simulação com quatro rainhas

Q_1			
	Q_2		

Retrocede e reposicione Q_2 .

Não há mais posição válida para Q_2 .

Simulação com quatro rainhas

Q_1			

Retrocede e reposicione Q_1 .

Agora tente novamente a rainha Q_2

Simulação com quatro rainhas

	Q_2		
Q_1			

A rainha Q_2 não está salva.
Procure outra posição para Q_2 .

Simulação com quatro rainhas

Q_1	Q_2		

A rainha Q_2 não está salva.
Procure outra posição para Q_2 .

Simulação com quatro rainhas

Q_1			
	Q_2		

A rainha Q_2 não está salva.
Procure outra posição para Q_2 .

Simulação com quatro rainhas

		Q_3	
Q_1			
	Q_2		

A rainha Q_2 está salva.

Agora tente novamente para Q_3 .

Simulação com quatro rainhas

		Q_3	
Q_1			
	Q_2		

A rainha Q_3 está salva.

Agora tente novamente para Q_4 .

Simulação com quatro rainhas

		Q_3	Q_4
Q_1			
	Q_2		

A rainha Q_4 não está salva.
Procure outra posição para Q_4 .

Simulação com quatro rainhas

		Q_3	
Q_1			Q_4
	Q_2		

A rainha Q_4 não está salva.
Procure outra posição para Q_4 .

Simulação com quatro rainhas

		Q_3	
Q_1			
			Q_4
	Q_2		

A rainha Q_4 está salva.

Todas as rainhas estão salvas!

Oito rainhas

Mais importante, num determinado passo do que saber a exata posição das rainhas que já estão no tabuleiro é saber se ela está salva.

Oito rainhas - implementação

```
bool esta_salva(char tabuleiro[][N], int linha, int coluna){
    int i, j;
    for (i = 0; i < coluna; i++) // verifica a linha
        if (tabuleiro[linha][i] == 'Q') return false;

    // verifica as diagonais
    for (i = linha, j = coluna; i >= 0 && j >= 0; i--, j--)
        if (tabuleiro[i][j] == 'Q') return false;

    for (i = linha, j = coluna; i < N && j > 0; i++, j--)
        if (tabuleiro[i][j] == 'Q') return false;

    return true;
}
```

Oito rainhas - implementação

```
void posiciona_rainha(char tabuleiro[][N], int coluna) {  
    if (coluna == N) mostra_matriz(tabuleiro);  
    else {  
        for (int i = 0; i < N; i++) {  
            if (esta_salva(tabuleiro, i, coluna)){  
                tabuleiro[i][coluna] = 'Q';  
                posiciona_rainha(tabuleiro, coluna + 1);  
                tabuleiro[i][coluna] = '-';  
            }  
        }  
    }  
}
```



```
int main(int argc, char *argv[]) {  
    char tabuleiro[N][N];  
    memset(tabuleiro, '-', sizeof tabuleiro);  
    oito_rainhas(tabuleiro, 0);  
    return 0;  
}
```

Obrigado