MC-202
Backtracking

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2018

Sequências

Como imprimir todas as sequências de tamanho k de números entre 1 e n?

Sequências

Como imprimir todas as sequências de tamanho k de números entre 1 e n?

Exemplo: $n = 4, k = 3$

Sequências

Como imprimir todas as sequências de tamanho k de números entre 1 e n?

Exemplo: $n = 4, k = 3$

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

Sequências

Como imprimir todas as sequências de tamanho k de números entre 1 e n?

Exemplo: $n = 4, k = 3$

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

Toda sequência que começa com i é seguida de uma sequência de tamanho $k - 1$ de números entre 1 e n

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

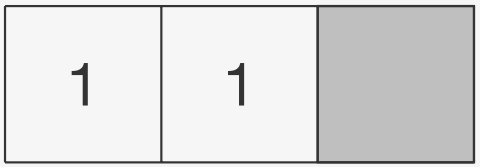
Simulação para $n = 4, k = 3$:

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

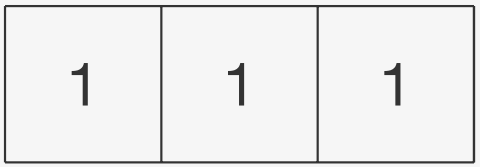


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:



Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

1	1	2

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

1	1	3

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

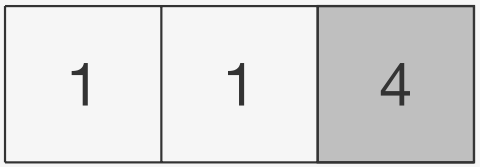
1	1	4

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

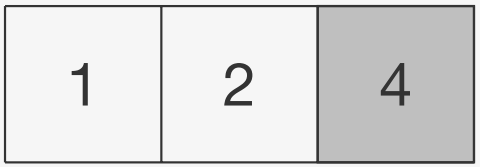


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

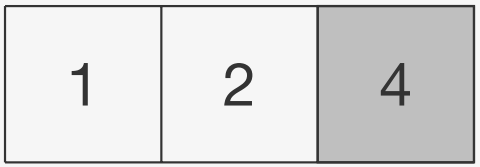


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

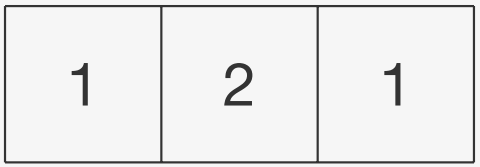


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:



Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

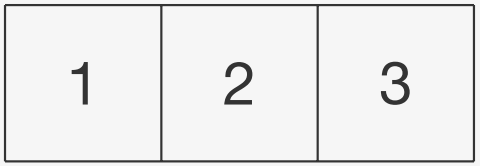
1	2	2

Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

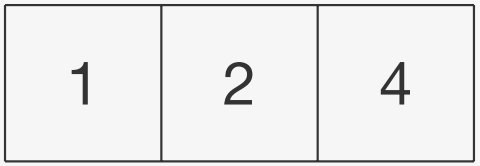


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

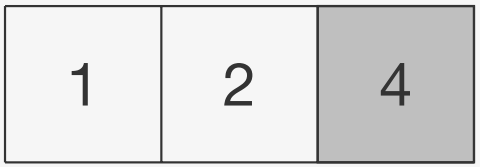


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:

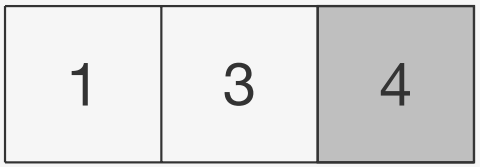


Sequências

Podemos resolver usando Recursão:

- Armazenamos o prefixo da sequência que estamos construindo
- Completamos com todos os possíveis sufixos recursivamente

Simulação para $n = 4, k = 3$:



Sequências - Implementação

```
1 void sequencias(int n, int k) {
2   int *seq = malloc(k * sizeof(int));
3   sequenciasR(seq, n, k, 0);
4   free(seq);
5 }
```

Sequências - Implementação

```
1  void sequencias(int n, int k) {
2    int *seq = malloc(k * sizeof(int));
3    sequenciasR(seq, n, k, 0);
4    free(seq);
5  }
6
7  void sequenciasR(int *seq, int n, int k, int i) {
8    int v;
9    if (i == k) {
10     imprimi_vetor(seq, k);
11     return;
12   }
13   for (v = 1; v <= n; v++) {
14     seq[i] = v;
15     sequenciasR(seq, n, k, i+1);
16   }
17 }
```

Sequências - Implementação

```
1  void sequencias(int n, int k) {
2    int *seq = malloc(k * sizeof(int));
3    sequenciasR(seq, n, k, 0);
4    free(seq);
5  }
6
7  void sequenciasR(int *seq, int n, int k, int i) {
8    int v;
9    if (i == k) {
10     imprimi_vetor(seq, k);
11     return;
12   }
13   for (v = 1; v <= n; v++) {
14     seq[i] = v;
15     sequenciasR(seq, n, k, i+1);
16   }
17 }
```

Sequências sem repetições

Queremos agora imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições

Sequências sem repetições

Queremos agora imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições

Primeiro algoritmo:

Sequências sem repetições

Queremos agora imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições

Primeiro algoritmo:

- já temos um algoritmo que gera todas as sequências com repetições

Sequências sem repetições

Queremos agora imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições

Primeiro algoritmo:

- já temos um algoritmo que gera todas as sequências com repetições
- testar se uma sequência tem repetição é fácil

Sequências sem repetições

Queremos agora imprimir todas as sequências de tamanho k de números entre 1 e n sem repetições

Primeiro algoritmo:

- já temos um algoritmo que gera todas as sequências com repetições
- testar se uma sequência tem repetição é fácil
- basta imprimir as sequências que passarem no teste!

Checando por repetições

```
1 int busca(int *vetor, int k, int valor) {
2   int i;
3   for (i = 0; i < k; i++)
4     if (vetor[i] == valor)
5       return 1;
6   return 0;
7 }
```

Checando por repetições

```
 1 int busca(int *vetor, int k, int valor) {
 2   int i;
 3   for (i = 0; i < k; i++)
 4     if (vetor[i] == valor)
 5       return 1;
 6   return 0;
 7 }
 8
 9 int tem_repeticao(int *vetor, int k) {
10   int i;
11   for (i = k-1; i > 0; i--)
12     if (busca(vetor, i, vetor[i]))
13       return 1;
14   return 0;
15 }
```

Checando por repetições

```
1 void sem_repeticao(int n, int k) {
2   int *seq = malloc(k * sizeof(int));
3   sem_repeticaoR(seq, n, k, 0);
4   free(seq);
5 }
```

Checando por repetições

```
1  void sem_repeticao(int n, int k) {
2    int *seq = malloc(k * sizeof(int));
3    sem_repeticaoR(seq, n, k, 0);
4    free(seq);
5  }
6
7  void sem_repeticaoR(int *seq, int n, int k, int i) {
8    int v;
9    if (i == k) {
10     if (!tem_repeticao(seq, k))
11       imprimi_vetor(seq, k);
12     return;
13   }
14   for (v = 1; v <= n; v++) {
15     seq[i] = v;
16     sem_repeticaoR(seq, n, k, i+1);
17   }
18 }
```

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

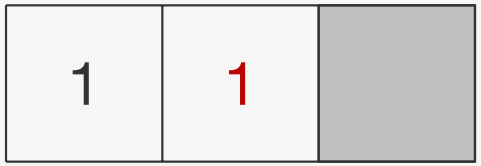
Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

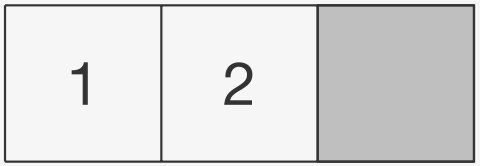


Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:



Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

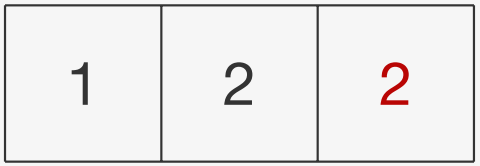
Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

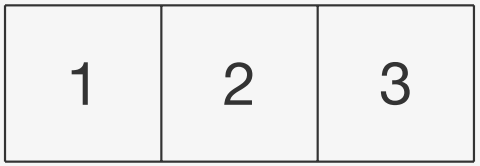


Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

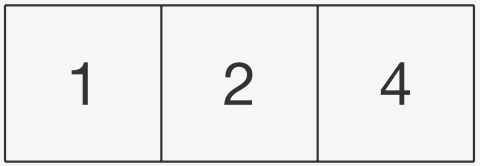


Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

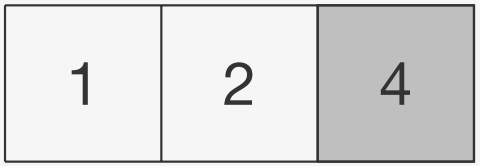


Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:



Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	3	1

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	3	2

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	3	4

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	4	1

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	4	3

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

1	4	4

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

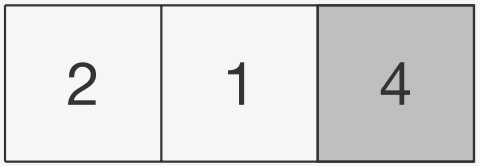
Simulação para $n = 4, k = 3$:

Segundo Algoritmo

Podemos ir construindo a sequência passo-a-passo sem permitir repetições

- Basta verificar se o número que queremos adicionar no vetor já está na sequência

Simulação para $n = 4, k = 3$:



Segundo Algoritmo

```
1  void sem_repeticaoR(int *seq, int n, int k, int i) {
2    int v;
3    if (i == k) {
4      imprimi_vetor(seq, k);
5      return;
6    }
7    for (v = 1; v <= n; v++) {
8      if (!busca(seq, i, v)) {
9        seq[i] = v;
10       sem_repeticaoR(seq, n, k, i+1);
11     }
12   }
13 }
```

Terceiro Algoritmo

Guardamos a informação de quais números já foram usados

Terceiro Algoritmo

Guardamos a informação de quais números já foram usados

- Vetor `usado` de `n+1` posições

Terceiro Algoritmo

Guardamos a informação de quais números já foram usados

- Vetor `usado` de `n+1` posições
- `usado[i] = 1` se `i` está no prefixo

Terceiro Algoritmo

Guardamos a informação de quais números já foram usados

- Vetor `usado` de `n+1` posições
- `usado[i] = 1` se `i` está no prefixo
- `usado[i] = 0` se `i` não está no prefixo

Terceiro Algoritmo

Guardamos a informação de quais números já foram usados

- Vetor `usado` de `n+1` posições
- `usado[i] = 1` se `i` está no prefixo
- `usado[i] = 0` se `i` não está no prefixo
- Bem mais rápido do que fazer a busca

Terceiro Algoritmo

```
1 void sem_repeticao(int n, int k) {
2   int *seq = malloc(k * sizeof(int));
3   int *usado = calloc(n + 1, sizeof(int));
4   sem_repeticaoR(seq, usado, n, k, 0);
5   free(seq);
6   free(usado);
7 }
```

Terceiro Algoritmo

```
 1 void sem_repeticao(int n, int k) {
 2   int *seq = malloc(k * sizeof(int));
 3   int *usado = calloc(n + 1, sizeof(int));
 4   sem_repeticaoR(seq, usado, n, k, 0);
 5   free(seq);
 6   free(usado);
 7 }
 8
 9 void sem_repeticaoR(int *seq, int *usado, int n, int k, int i) {
10   int v;
11   if (i == k) {
12     imprimi_vetor(seq, k);
13     return;
14   }
15   for (v = 1; v <= n; v++) {
16     if (!usado[v]) {
17       seq[i] = v;
18       usado[v] = 1;
19       sem_repeticaoR(seq, usado, n, k, i+1);
20       usado[v] = 0;
21     }
22   }
23 }
```

Comparação

Comparação

Primeiro algoritmo:

- Gera todas as sequências com repetições
- Testa para ver se a sequência tem repetições
- Tempo para $n = k = 10$: 116,98s

Comparação

Primeiro algoritmo:

- Gera todas as sequências com repetições
- Testa para ver se a sequência tem repetições
- Tempo para $n = k = 10$: 116,98s

Segundo algoritmo:

- Gera apenas sequências sem repetições
- Usa `busca` para ver se o número já está na sequência
- Tempo para $n = k = 10$: 4,16s

Comparação

Primeiro algoritmo:

- Gera todas as sequências com repetições
- Testa para ver se a sequência tem repetições
- Tempo para $n = k = 10$: 116,98s

Segundo algoritmo:

- Gera apenas sequências sem repetições
- Usa `busca` para ver se o número já está na sequência
- Tempo para $n = k = 10$: 4,16s

Terceiro algoritmo:

- Gera apenas sequências sem repetições
- Usa um vetor para ver se o número já está na sequência
- Tempo para $n = k = 10$: 3,83s

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

- Ex.: para quebrar uma senha, podemos gerar cada senha sistematicamente e testamos se é a senha válida

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

- Ex.: para quebrar uma senha, podemos gerar cada senha sistematicamente e testamos se é a senha válida
- Podemos enumerar estruturas (como sequências)

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

- Ex.: para quebrar uma senha, podemos gerar cada senha sistematicamente e testamos se é a senha válida
- Podemos enumerar estruturas (como sequências)
- Podemos encontrar todas as soluções de um problema

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

- Ex.: para quebrar uma senha, podemos gerar cada senha sistematicamente e testamos se é a senha válida
- Podemos enumerar estruturas (como sequências)
- Podemos encontrar todas as soluções de um problema

Porém, a força bruta pode ser muito lenta para resolver determinados problemas

Força Bruta

Geramos os candidatos a solução do problema e testamos para ver se é de fato uma solução

- Ex.: para quebrar uma senha, podemos gerar cada senha sistematicamente e testamos se é a senha válida
- Podemos enumerar estruturas (como sequências)
- Podemos encontrar todas as soluções de um problema

Porém, a força bruta pode ser muito lenta para resolver determinados problemas

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia
- Enquanto for possível, adicionamos um elemento à solução parcial

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia
- Enquanto for possível, adicionamos um elemento à solução parcial
- Se encontrarmos uma solução completa, terminamos

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia
- Enquanto for possível, adicionamos um elemento à solução parcial
- Se encontrarmos uma solução completa, terminamos
- Se não é possível adicionar mais nenhum elemento à solução parcial, retrocedemos

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia
- Enquanto for possível, adicionamos um elemento à solução parcial
- Se encontrarmos uma solução completa, terminamos
- Se não é possível adicionar mais nenhum elemento à solução parcial, retrocedemos
  - removemos um ou mais elementos da solução parcial

Backtracking - Retrocesso

Resolver um problema de forma recursiva, podendo tomar decisões erradas

- Nesse caso, escolhemos outra decisão

Construímos soluções passo-a-passo, retrocedendo se a solução parcial atual não é válida

- Começamos com uma solução parcial vazia
- Enquanto for possível, adicionamos um elemento à solução parcial
- Se encontrarmos uma solução completa, terminamos
- Se não é possível adicionar mais nenhum elemento à solução parcial, retrocedemos
  - removemos um ou mais elementos da solução parcial
  - e tomamos decisões diferentes das que foram tomadas

Sudoku

No Sudoku, nós temos uma matriz 9×9 com algumas entradas preenchidas com números entre 1 e 9

			2	4	3	1		
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Objetivo: completar a matriz com números entre 1 e 9 sem repetir números nas linhas, nas colunas e nas células

Sudoku

No Sudoku, nós temos uma matriz 9×9 com algumas entradas preenchidas com números entre 1 e 9

7	5	9	2	4	3	1	6	8
3	2	8	9	1	6	4	5	7
1	4	6	8	5	7	9	2	3
9	7	2	6	8	1	3	4	5
4	8	5	7	3	9	6	1	2
6	1	3	4	2	5	7	8	9
8	9	7	5	6	4	2	3	1
2	6	1	3	9	8	5	7	4
5	3	4	1	7	2	8	9	6

Objetivo: completar a matriz com números entre 1 e 9 sem repetir números nas linhas, nas colunas e nas células

Sudoku

No Sudoku, nós temos uma matriz 9×9 com algumas entradas preenchidas com números entre 1 e 9

7	5	9	2	4	3	1	6	8
3	2	8	9	1	6	4	5	7
1	4	6	8	5	7	9	2	3
9	7	2	6	8	1	3	4	5
4	8	5	7	3	9	6	1	2
6	1	3	4	2	5	7	8	9
8	9	7	5	6	4	2	3	1
2	6	1	3	9	8	5	7	4
5	3	4	1	7	2	8	9	6

Objetivo: completar a matriz com números entre 1 e 9 sem repetir números nas linhas, nas colunas e nas células

Sudoku

No Sudoku, nós temos uma matriz 9×9 com algumas entradas preenchidas com números entre 1 e 9

7	5	9	2	4	3	1	6	8
3	2	8	9	1	6	4	5	7
1	4	6	8	5	7	9	2	3
9	7	2	6	8	1	3	4	5
4	8	5	7	3	9	6	1	2
6	1	3	4	2	5	7	8	9
8	9	7	5	6	4	2	3	1
2	6	1	3	9	8	5	7	4
5	3	4	1	7	2	8	9	6

Objetivo: completar a matriz com números entre 1 e 9 sem repetir números nas linhas, nas colunas e nas células

Sudoku

No Sudoku, nós temos uma matriz 9×9 com algumas entradas preenchidas com números entre 1 e 9

7	5	9	2	4	3	1	6	8
3	2	8	9	1	6	4	5	7
1	4	6	8	5	7	9	2	3
9	7	2	6	8	1	3	4	5
4	8	5	7	3	9	6	1	2
6	1	3	4	2	5	7	8	9
8	9	7	5	6	4	2	3	1
2	6	1	3	9	8	5	7	4
5	3	4	1	7	2	8	9	6

Objetivo: completar a matriz com números entre 1 e 9 sem repetir números nas linhas, nas colunas e nas células

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

			2	4	3	1		
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5			2	4	3	1		
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6		2	4	3	1		
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1		
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	8	
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	8	?
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1		8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	2	8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	2	8	9		6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	2	8	9	?	6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	2	8	9		6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	2	8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	3	8			6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

5	6	7	2	4	3	1	9	8
1	3	8	9		6		5	
	4							
			6					5
4	8		7	3	9	6	1	
			4					9
	9							
		1			8		7	
			1	7	2	8		

Sudoku - Resolução por Backtracking

Preenchemos o Sudoku gradualmente:

- até encontrar uma posição sem valor válido
- retrocedemos e continuamos a busca

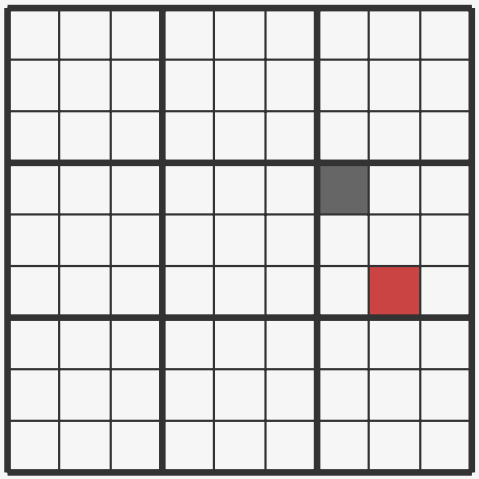
Após várias iterações...

7	5	9	2	4	3	1	6	8
3	2	8	9	1	6	4	5	7
1	4	6	8	5	7	9	2	3
9	7	2	6	8	1	3	4	5
4	8	5	7	3	9	6	1	2
6	1	3	4	2	5	7	8	9
8	9	7	5	6	4	2	3	1
2	6	1	3	9	8	5	7	4
5	3	4	1	7	2	8	9	6

Sudoku - Código

Sudoku - Código

```
1  int pode_inserir(int m[9][9], int l, int c, int v) {
2    int i, j, cel_l, cel_c;
3    for(i = 0; i < 9; i++)
4      if (m[l][i] == v) /* aparece na linha l? */
5        return 0;
6    for(i = 0; i < 9; i++)
7      if (m[i][c] == v) /* aparece na coluna c? */
8        return 0;
9
10   cel_l = 3 * (l / 3);
11   cel_c = 3 * (c / 3);
12   for(i = cel_l; i < cel_l + 3; i++)
13     for(j = cel_c; j < cel_c + 3; j++)
14       if(m[i][j] == v) /* aparece na célula? */
15         return 0;
16   return 1;
17 }
```



Sudoku - Código

```
 1 int sudoku(int m[9][9]) {
 2   int i, j, fixo[9][9];
 3   for (i = 0; i < 9; i++)
 4     for (j = 0; j < 9; j++)
 5       fixo[i][j] = m[i][j]; /* diferente de zero é verdadeiro */
 6   return sudokuR(m, fixo, 0, 0);
 7 }
 8
 9 void proxima_posicao(int l, int c, int *nl, int *nc) {
10   if (c < 8) {
11     *nl = l;
12     *nc = c+1;
13   } else {
14     *nl = l+1;
15     *nc = 0;
16   }
17 }
```

Sudoku - Código

```
1  int sudokuR(int m[9][9], int fixo[9][9], int l, int c) {
2    int v, nl, nc;
3    if (l == 9) {
4      imprimi_sudoku(m);
5      return 1;
6    }
7    proxima_posicao(l, c, &nl, &nc);
8    if (fixo[l][c])
9      return sudokuR(m, fixo, nl, nc);
10   for (v = 1; v <= 9; v++) {
11     if (pode_inserir(m, l, c, v)) {
12       m[l][c] = v;
13       if(sudokuR(m, fixo, nl, nc))
14         return 1;
15     }
16   }
17   m[l][c] = 0;
18   return 0;
19 }
```

Passeio do Cavalo no Tabuleiro de Xadrez

Passeio do Cavalo no Tabuleiro de Xadrez

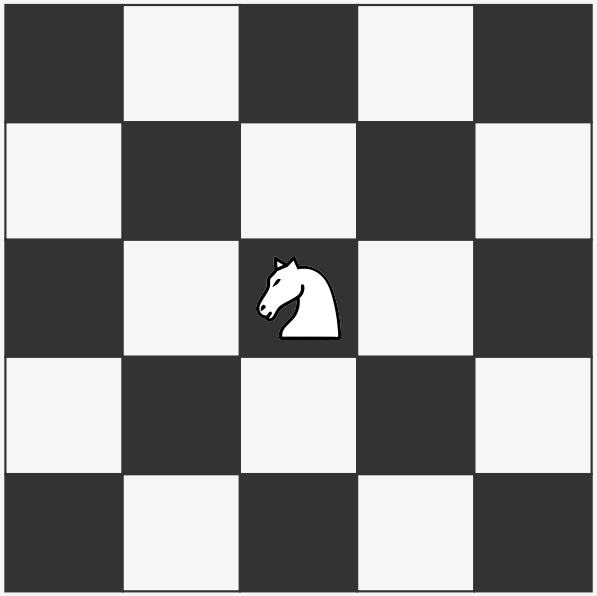
Movimento do cavalo no xadrez - formato de **L**:

- dois quadrados horizontalmente e um verticalmente, ou
- dois quadrados verticalmente e um horizontalmente

Passeio do Cavalo no Tabuleiro de Xadrez

Movimento do cavalo no xadrez - formato de **L**:

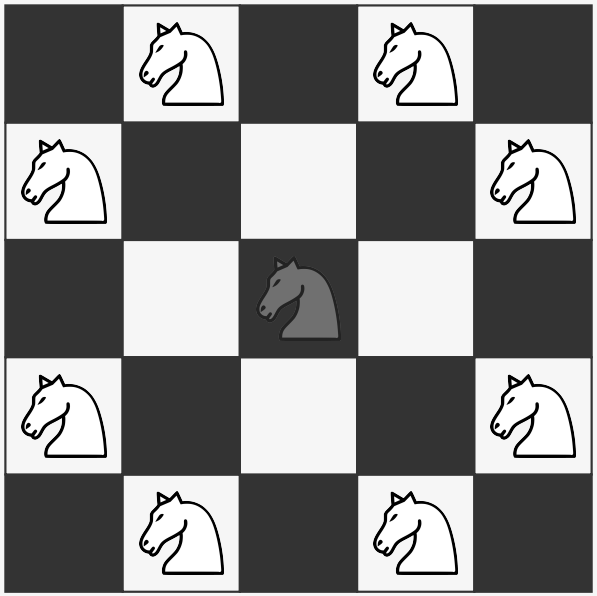
- dois quadrados horizontalmente e um verticalmente, ou
- dois quadrados verticalmente e um horizontalmente



Passeio do Cavalo no Tabuleiro de Xadrez

Movimento do cavalo no xadrez - formato de **L**:

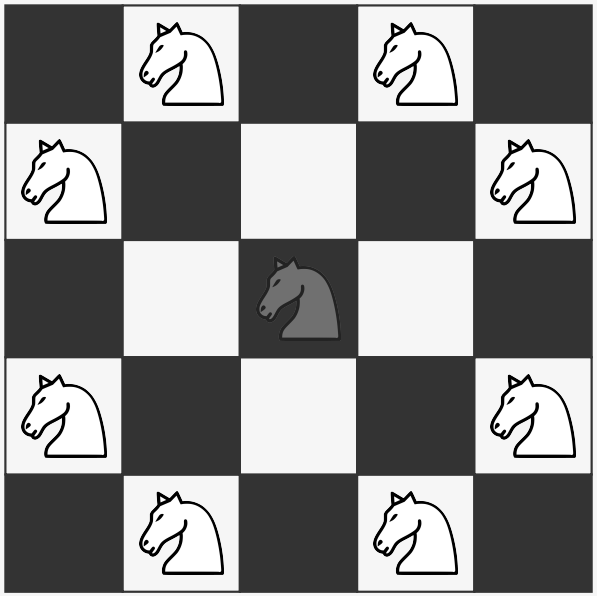
- dois quadrados horizontalmente e um verticalmente, ou
- dois quadrados verticalmente e um horizontalmente



Passeio do Cavalo no Tabuleiro de Xadrez

Movimento do cavalo no xadrez - formato de **L**:

- dois quadrados horizontalmente e um verticalmente, ou
- dois quadrados verticalmente e um horizontalmente

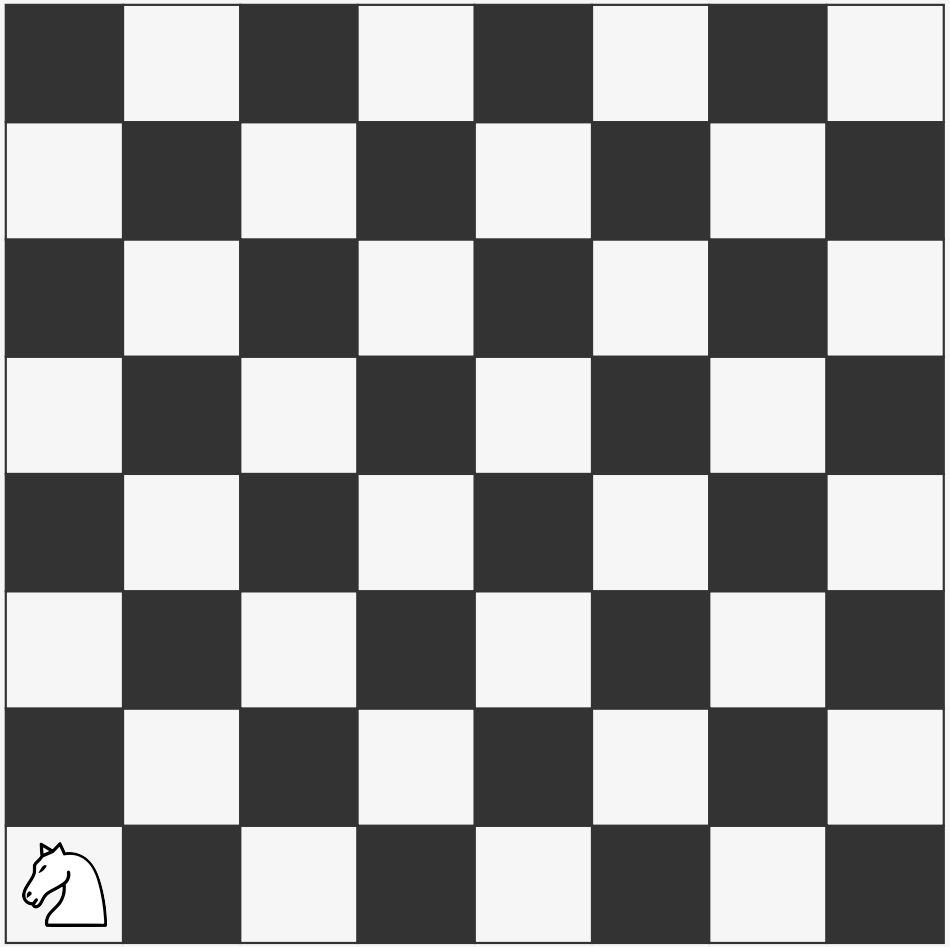


Dado um tabuleiro de xadrez $n \times n$ e uma posição (x, y) do tabuleiro queremos encontrar um passeio de um cavalo que visite cada casa exatamente uma vez

Simulação

Matriz `m` armazena os movimentos do cavalo

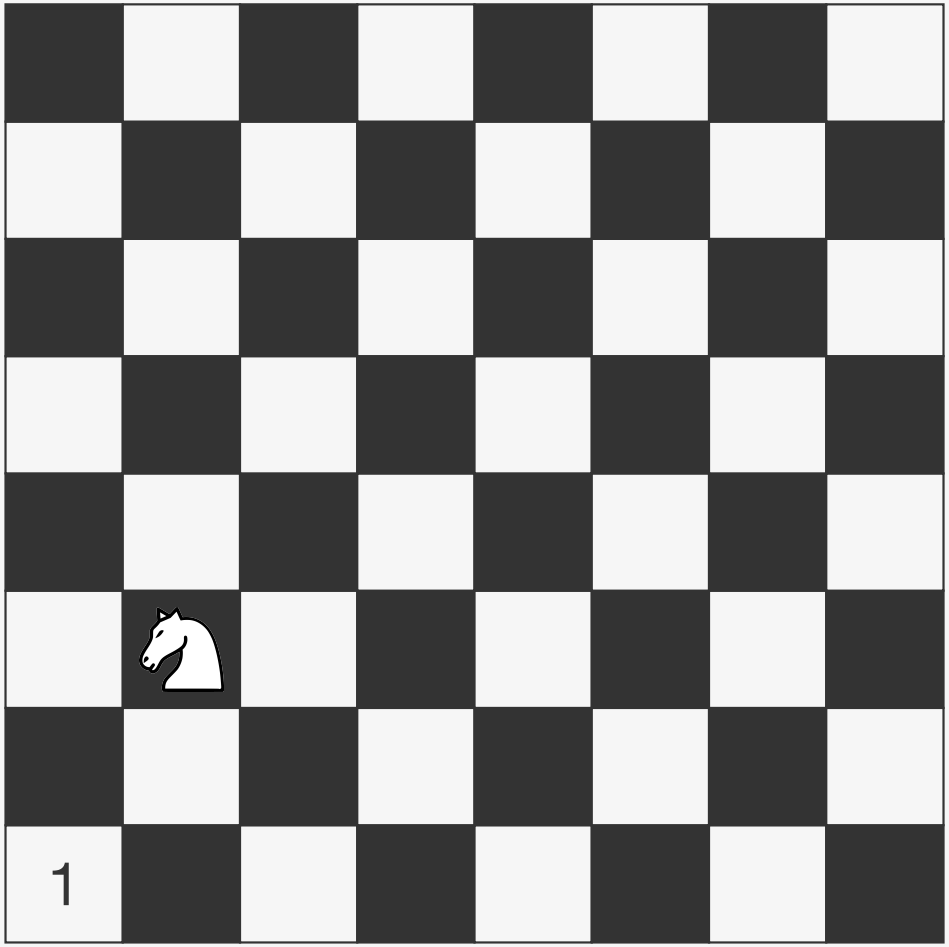
- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

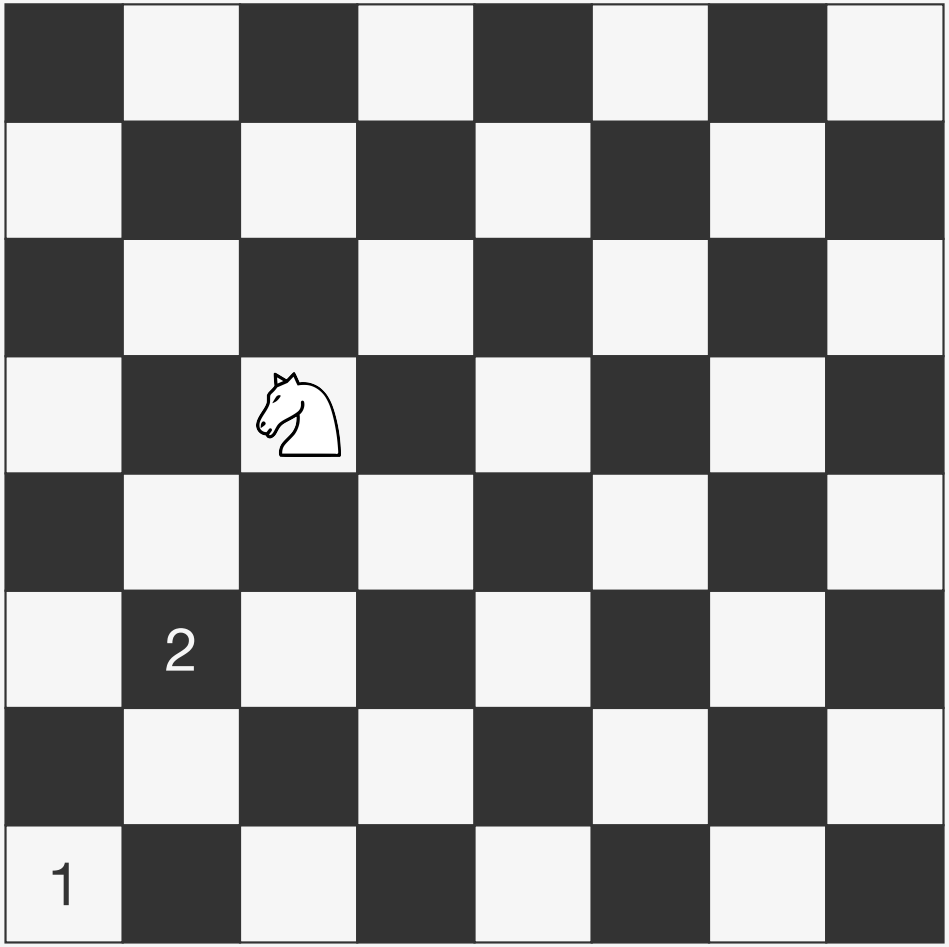
- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

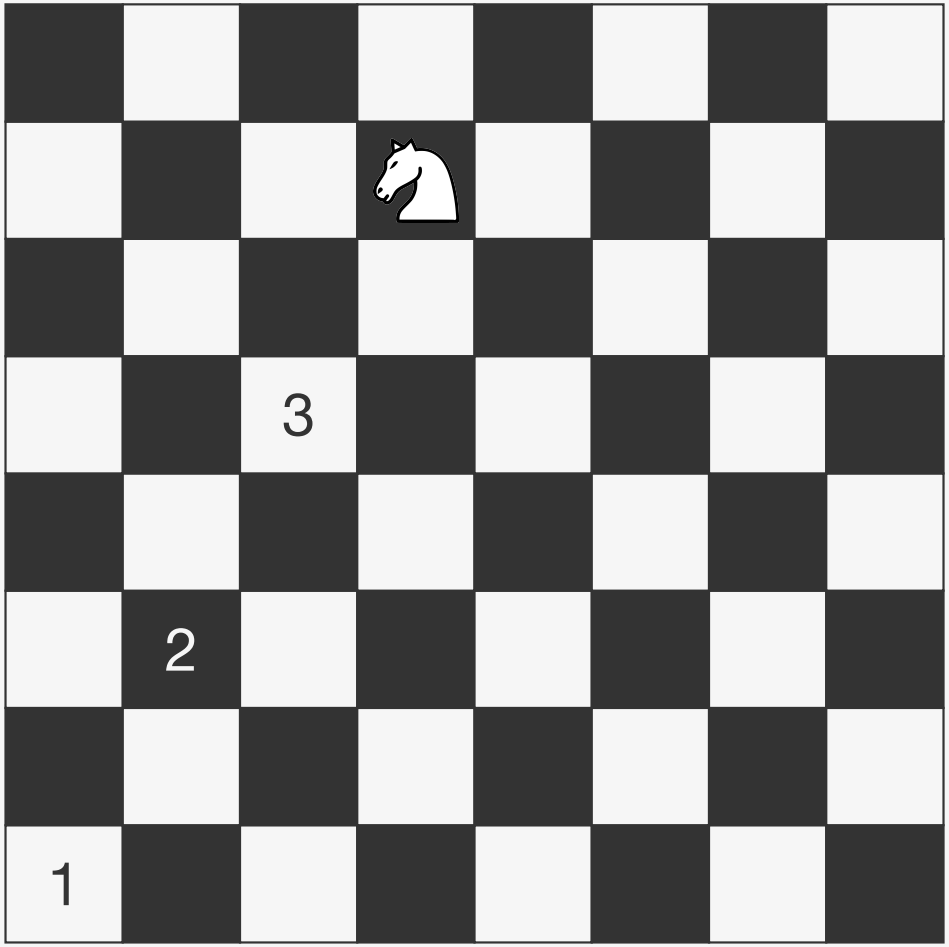
- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

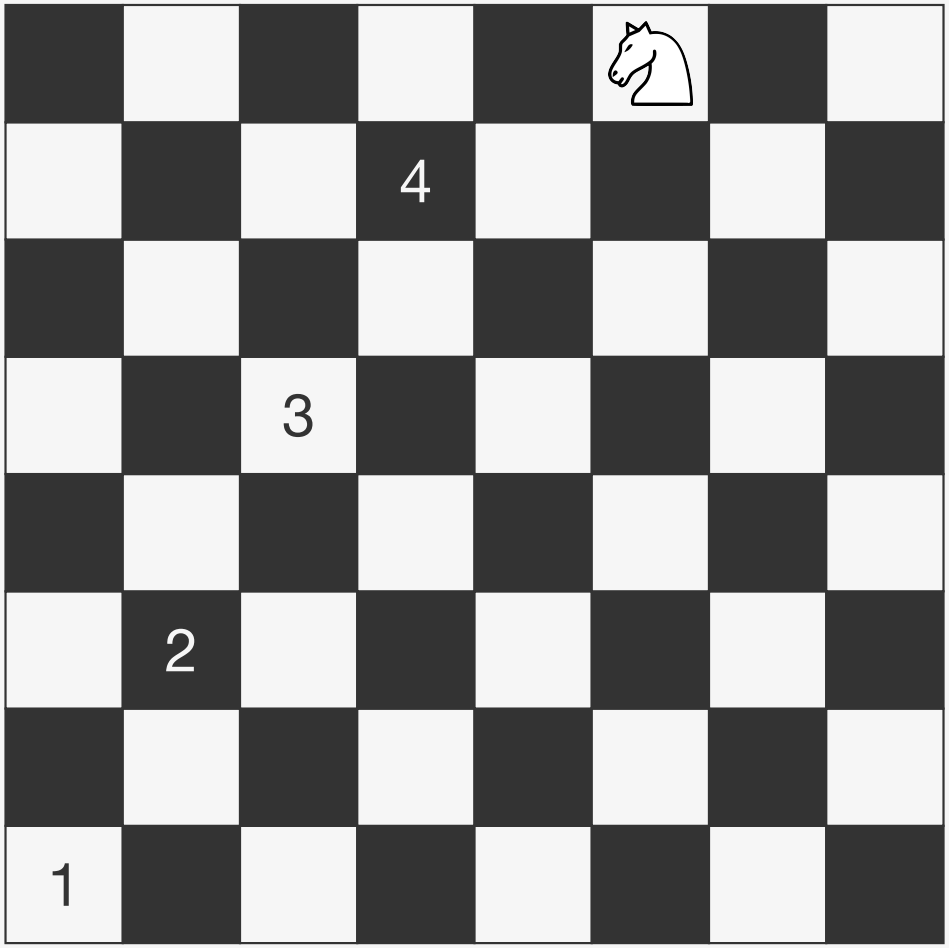
- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

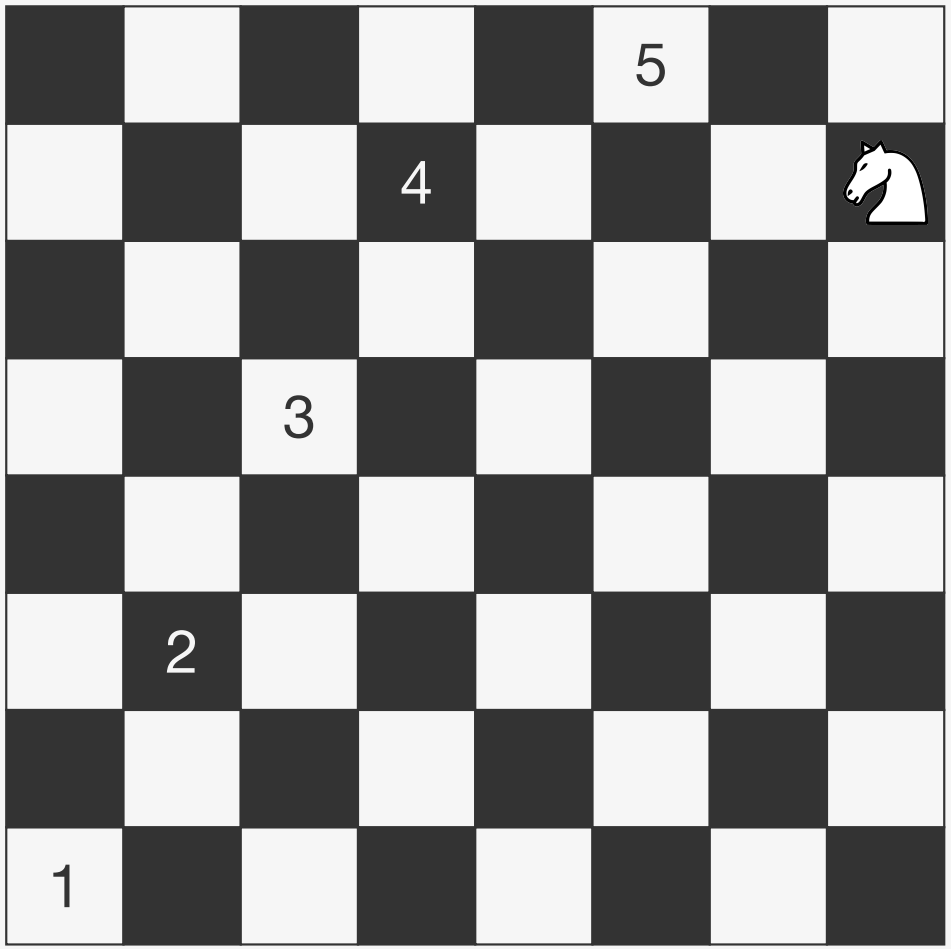
- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i



Simulação

Matriz `m` armazena os movimentos do cavalo

- `m[l][c]` $= 0$: posição $(\texttt{l}, \texttt{c})$ ainda não foi visitada
- `m[l][c]` $= i > 0$: posição $(\texttt{l}, \texttt{c})$ foi visitada no passo i

52	47	56	45	54	5	22	13
57	44	53	4	23	14	25	6
48	51	46	55	26	21	12	15
43	58	3	50	41	24	7	20
36	49	42	27	62	11	16	29
59	2	37	40	33	28	19	8
38	35	32	61	10	63	30	17
1	60	39	34	31	18	9	64

Passeio do Cavalo - Código

Passeio do Cavalo - Código

```
1  int cavalo(int **m, int n, int x, int y) {
2    int i, j;
3    for (i = 0; i < n; i++)
4      for (j = 0; j < n; j++)
5        m[i][j] = 0;
6    m[x][y] = 1;
7    return cavaloR(m, n, x, y);
8  }
9
10 void proxima_posicao(int l, int c, int k, int *nl, int *nc) {
11   static int movimentos[8][2] = {{2, 1}, {1, 2}, {-1, 2},
12                                  {-2, 1}, {-2,-1}, {-1, -2},
13                                  {1, -2}, {2, -1}};
14   *nl = l + movimentos[k][0];
15   *nc = c + movimentos[k][1];
16 }
```

Cavalo - Código

```
1  int cavaloR(int **m, int n, int l, int c) {
2    int k, nl, nc;
3    if (m[l][c] == n * n)
4      return 1;
5    for (k = 0; k < 8; k++) {
6      proxima_posicao(l, c, k, &nl, &nc);
7      if ((nl >= 0) && (nl < n) && (nc >= 0) && (nc < n)
8          && (m[nl][nc] == 0)) {
9        m[nl][nc] = m[l][c] + 1;
10       if (cavaloR(m, n, nl, nc))
11         return 1;
12       m[nl][nc] = 0;
13     }
14   }
15   return 0;
16 }
```

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez
- Implementação simples, mas pode ser lento para problemas onde temos muitas soluções parciais possíveis

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez
- Implementação simples, mas pode ser lento para problemas onde temos muitas soluções parciais possíveis

Como fazer um algoritmo de Backtracking rápido?

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez
- Implementação simples, mas pode ser lento para problemas onde temos muitas soluções parciais possíveis

Como fazer um algoritmo de Backtracking rápido?

- Ter um algoritmo para decidir se uma solução parcial pode ser estendida para uma solução completa que seja

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez
- Implementação simples, mas pode ser lento para problemas onde temos muitas soluções parciais possíveis

Como fazer um algoritmo de Backtracking rápido?

- Ter um algoritmo para decidir se uma solução parcial pode ser estendida para uma solução completa que seja
  - Bom: Evita explorar muitas soluções parciais

Eficiência do Backtracking

- Em geral, mais rápido que a Força Bruta pois eliminamos vários candidatos a solução de uma só vez
- Implementação simples, mas pode ser lento para problemas onde temos muitas soluções parciais possíveis

Como fazer um algoritmo de Backtracking rápido?

- Ter um algoritmo para decidir se uma solução parcial pode ser estendida para uma solução completa que seja
  - Bom: Evita explorar muitas soluções parciais
  - Rápido: Processa cada solução parcial rapidamente

Aplicações para Backtracking

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo

- Problemas de Otimização Combinatória

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo
- Programação Lógica (Prolog, por exemplo)

Aplicações para Backtracking

Para aplicar Backtracking é necessário que o problema tenha um conceito de solução parcial

- Problemas de satisfação de restrições
  - Encontrar uma solução que satisfaça as restrições
  - Como o Sudoku, por exemplo
- Problemas de Otimização Combinatória
  - Conseguimos enumerar as soluções do problema
  - Queremos encontrar a de valor mínimo
- Programação Lógica (Prolog, por exemplo)
  - Prova automática de teoremas

Exercício

Modifique o algoritmo que resolve o Sudoku para saber rapidamente se um valor já foi usado numa linha, coluna ou célula. **Dica**: use matrizes auxiliares.

Exercício

Crie um algoritmo que, dado n e C, imprime todas as sequências de números não-negativos $x_1, x_2, \ldots, x_n$ tal que

$$x_1 + x_2 + \cdots + x_n = C$$

a) Modifique o seu algoritmo para considerar apenas sequências sem repetições

b) Modifique o seu algoritmo para imprimir apenas sequências com $x_1 \leq x_2 \leq \cdots x_n$