Técnicas para Programação Competitiva
Paradigmas de Resolução de Problemas

Prof. Andrei Braga

UNIVERSIDADE FEDERAL DA FRONTEIRA SUL

Conteúdo

- Busca completa (força bruta)
- Divisão e conquista (introdução)
- Referências

Motivação

- Vamos estudar paradigmas de resolução de problemas comumente utilizados para tratar problemas de competições de programação
- Para ter sucesso em competições de programação, precisamos ter um bom domínio sobre estes paradigmas, sabendo usar a opção apropriada para o problema em questão

Busca completa (força bruta)

- A **busca completa** (**força bruta**) é um método que inspeciona todo o (em alguns casos, parte do) espaço de soluções de um problema para resolvê-lo
- Durante a execução do método, podemos decidir não inspecionar certa parte do espaço de soluções caso tenhamos determinado que esta parte não pode conter a solução buscada
- A busca completa sempre retorna a solução buscada – também chamada de solução *ótima* ou *melhor* solução – caso esta solução exista

Busca completa (força bruta)

- É apropriado usarmos busca completa quando
    - não é possível utilizar um método mais eficiente ou
    - é possível utilizar um método mais eficiente, mas, pelo tamanho da entrada ser muito pequeno, isto é desnecessário
- Se for fácil e rápido de implementar (o que geralmente acontece), um método de busca completa também pode ajudar a testar se um algoritmo mais complexo está correto (através de testes com entradas pequenas)

Problema

Descrição adaptada do problema “441 - Lotto” do UVa Online Judge

- Um jogo de loteria é formado por 6 números do conjunto { 1, 2, …, 60 }. Uma estratégia (não muito boa :/) de jogo é selecionar *k* entre estes 60 números e fazer todas as apostas possíveis utilizando apenas os *k* números selecionados. Escreva um programa que imprima as apostas feitas através desta estratégia.
  Restrições: $6 < k < 13$.
- **Entrada:** *k* e os *k* números selecionados em ordem crescente.
- **Saída:** Todas as apostas possíveis utilizando apenas os *k* números selecionados, os números de cada aposta em ordem crescente.

Solução – Busca completa (iterativa)

- No pior caso, teremos 924 combinações – *C*(12,6)
- Dado um tempo limite de pelo menos 1 seg., provavelmente podemos resolver este problema escrevendo 6 laços
  - A complexidade destes laços é menor que 3×10^6

```
for (int i = 0; i < k; i++) cin >> vet[i];
for (int a = 0; a < k-5; a++)
    for (int b = a+1; b < k-4; b++)
        for (int c = b+1; c < k-3; c++)
            for (int d = c+1; d < k-2; d++)
                for (int e = d+1; e < k-1; e++)
                    for (int f = e+1; f < k; f++)
                        cout << vet[a] << " " << vet[b]
                            << " " << vet[c] << " " << vet[d]
                            << " " << vet[e] << " " << vet[f] << "\n";
```

Problema

Descrição adaptada do problema "12455 - Bars" do UVa Online Judge

- Dado um conjunto *S* de *n* inteiros, existe um subconjunto *S'* de *S* tal que a soma dos elementos de *S'* seja igual a um dado inteiro *X*?
  Restrições: $1 \le n \le 20$.

- **Entrada:** *X*, *n* e os *n* inteiros em *S*.
- **Saída:** O texto `SIM` caso a resposta para a pergunta do problema seja sim ou o texto `NAO` caso contrário.

Solução – Busca completa (iterativa)

- Possível estratégia: examinar todos os subconjuntos de S e verificar se algum deles satisfaz ao requisito do problema
- Complexidade desta estratégia:
  - Existem 2^n subconjuntos de S
  - A verificação de um subconjunto tem complexidade $O(n)$
  - Complexidade da estratégia: $O(2^n \times n)$
  - No pior caso ($n = 20$), a complexidade é menor que 4×10^7
- Dado um tempo limite de pelo menos 1 seg., provavelmente podemos resolver o problema com esta estratégia

Solução – Busca completa (iterativa)

- Na solução do problema anterior, escrevemos um código para gerar subconjuntos de tamanho 6 de um conjunto
- Como podemos gerar subconjuntos de um tamanho arbitrário de um conjunto?

Solução – Busca completa (iterativa)

- Podemos usar um inteiro de n bits para representar um subconjunto do conjunto { 0, 1, 2, . . . , n - 1 }
- Isto é feito associando as posições (da direita para à esquerda) dos 1's da representação binária do inteiro aos elementos do subconjunto
- Com inteiros do tipo `int` do C++ (valor máximo: 2^{31} - 1), podemos representar subconjuntos do conjunto { 0, 1, 2, . . . , 31 }
- Exemplo:
  - O inteiro 282 representa o subconjunto { 1, 3, 4, 8 }
  - A representação binária de 282 é 00000000000000000000000100011010, que tem 1's nas posições (da direita para à esquerda) 1, 3, 4, e 8

Solução – Busca completa (iterativa)

- Utilizando inteiros para representar subconjuntos, podemos resolver o problema com a seguinte ideia:

```
for (int i = 0; i < (1<<n); i++) { // Para i de 0 a 2^n-1 (obs.: n deve ser ≤ 31)
    int sum = 0;
    for (int j = 0; j < n; j++)    // Para cada possivel elemento do subconjunto
        if (i & (1<<j))            // O elemento j pertence ao subconjunto?
            sum += vet[j];
    if (sum == X) break;           // A soma dos elementos do subconjunto eh X
}
```

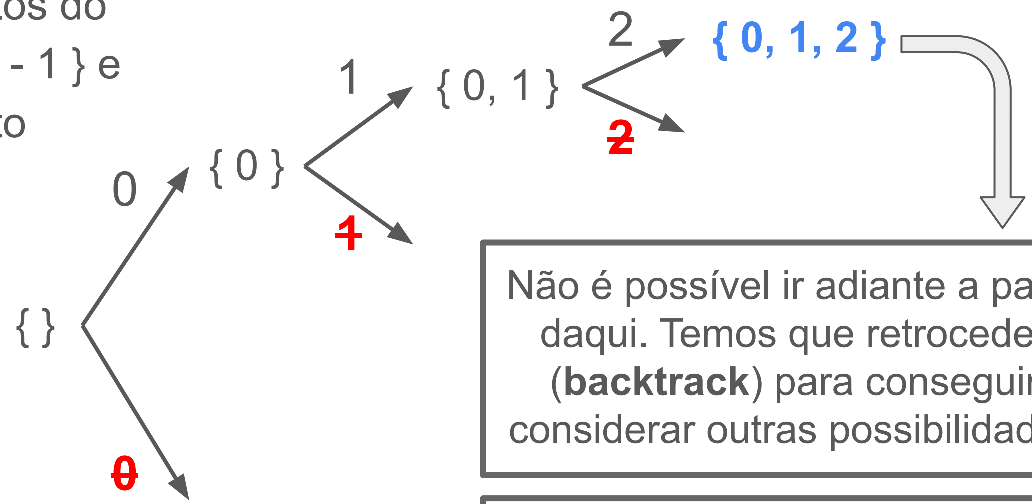
Gerando subconjuntos de um conjunto

- A solução anterior consistiu em
  - gerar todos os subconjuntos do conjunto $\{ 0, 1, 2, . . . , n - 1 \}$ e
  - processar cada subconjunto para verificar se uma condição era satisfeita
- Vamos ver como realizar este tipo de tarefa de outra maneira: usando **recursão** (**backtracking**)

Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , *n* - 1 } e
  - realiza um processamento para cada subconjunto

Ideia:



Considerando o conjunto { 0, 1, 2 }

Não é possível ir adiante a partir daqui. Temos que retroceder (**backtrack**) para conseguir considerar outras possibilidades

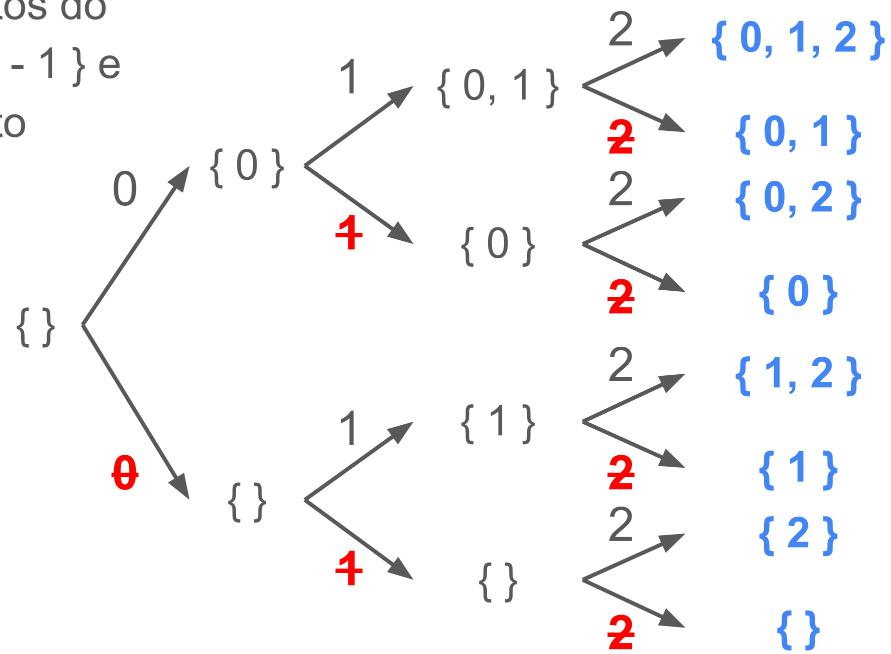
Uma abordagem como esta é chamada de **backtracking**

Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
    - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , *n* - 1 } e
    - realiza um processamento para cada subconjunto

Ideia:

Considerando o conjunto { 0, 1, 2 }



Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , *n* - 1 } e
  - realiza processamento para cada subconjunto

```
vector<int> subset;
```

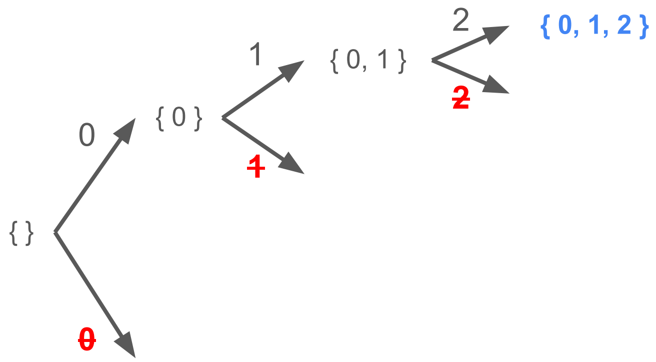
```
void search(int k) {



        subset.push_back(k);
        search(k+1);



}
```



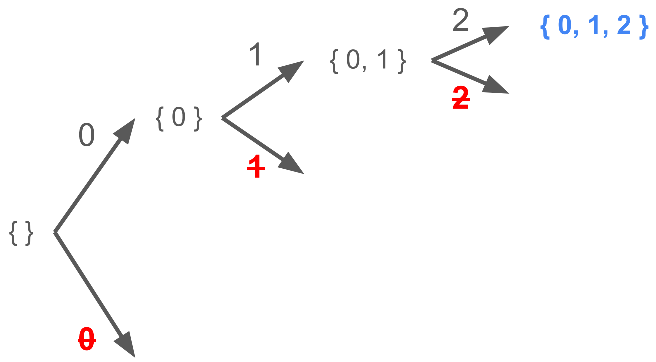
Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , n - 1 } e
  - realiza processamento para cada subconjunto

```
vector<int> subset;
```

```
void search(int k) {
    if (k == n) {
        // Processa subconjunto
    } else {
        subset.push_back(k);
        search(k+1);


    }
}
```

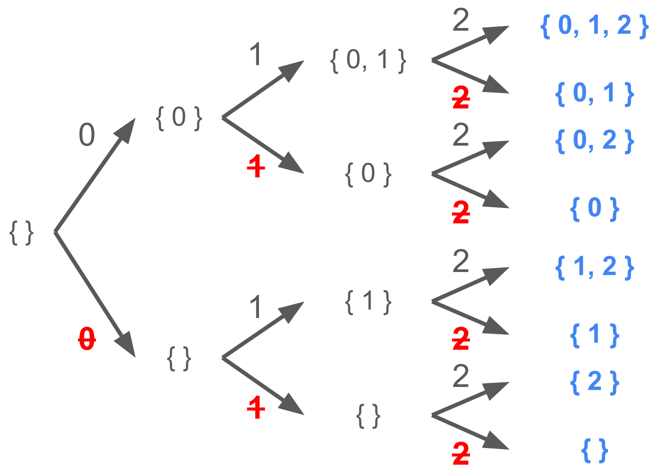


Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , *n* - 1 } e
  - realiza processamento para cada subconjunto

```
vector<int> subset;
```

```
void search(int k) {
    if (k == n) {
        // Processa subconjunto
    } else {
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        search(k+1);
    }
}
```



Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todos os subconjuntos do conjunto { 0, 1, 2, . . . , *n* - 1 } e
  - realiza processamento para cada subconjunto

```
vector<int> subset;
```

```
void search(int k) {
    if (k == n) {
        // Processa subconjunto
    } else {
        subset.push_back(k);
        search(k+1);
        subset.pop_back();
        search(k+1);
    }
}
```

Na primeira chamada, k vale 0 (valor mínimo do conjunto)

n corresponde ao valor máximo do conjunto + 1, ou seja, (n - 1) + 1

Adiciona k ao subconjunto

Não adiciona k ao subconjunto

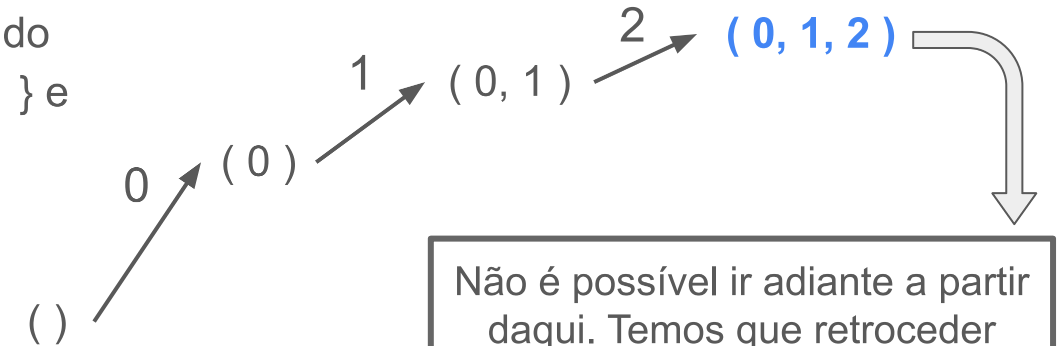
Gerando permutações de um conjunto

- Outro tipo de tarefa que comumente precisamos resolver consiste no seguinte:
  - gerar todas as permutações do conjunto $\{ 0, 1, 2, \ldots, n - 1 \}$ e
  - processar cada permutação para verificar se uma condição é satisfeita
- Vamos ver como realizar este outro tipo de tarefa usando **recursão** (**backtracking**)

Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto { 0, 1, 2, . . . , n - 1 } e
  - realiza um processamento para cada permutação

Ideia:



Considerando o conjunto { 0, 1, 2 }

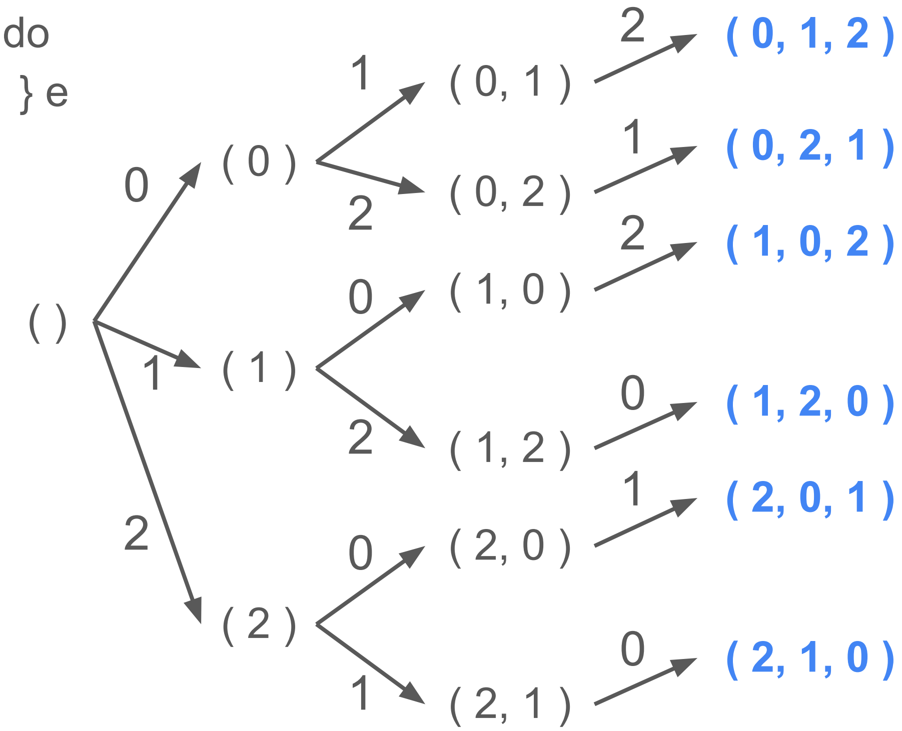
Não é possível ir adiante a partir daqui. Temos que retroceder (**backtrack**) para conseguir considerar outras possibilidades

Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto { 0, 1, 2, . . . , *n* - 1 } e
  - realiza um processamento para cada permutação

Ideia:

Considerando o conjunto { 0, 1, 2 }



Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto $\{0, 1, 2, \ldots, n - 1\}$ e
  - realiza um processamento para cada permutação

```
vector<int> permut; vector<bool> chosen(n);
```

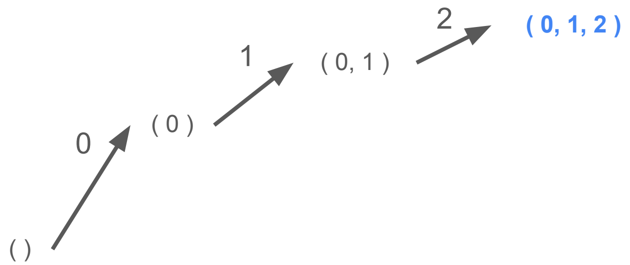
```
void search() {



        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permut.push_back(i);
            search();


        }

}
```



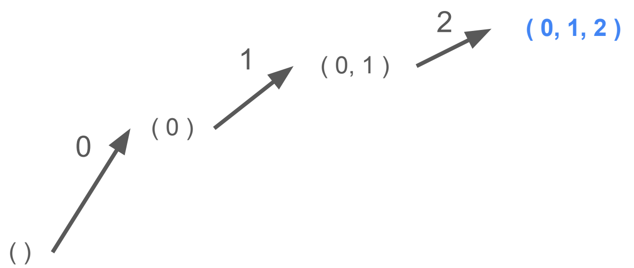
Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto $\{0, 1, 2, \ldots, n-1\}$ e
  - realiza um processamento para cada permutação

```
vector<int> permut; vector<bool> chosen(n);
```

```
void search() {
    if (permut.size() == n) {
        // Processa permutacao
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permut.push_back(i);
            search();


        }
    }
}
```

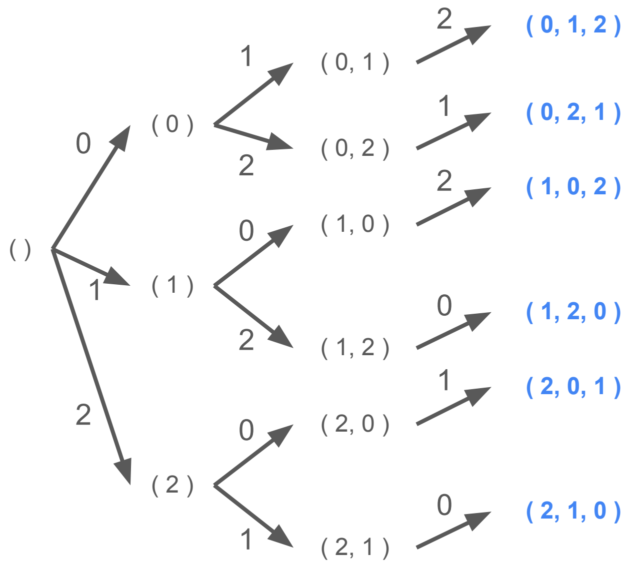


Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto $\{0, 1, 2, \ldots, n - 1\}$ e
  - realiza um processamento para cada permutação

```
vector<int> permut; vector<bool> chosen(n);
```

```
void search() {
    if (permut.size() == n) {
        // Processa permutacao
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permut.push_back(i);
            search();
            chosen[i] = false;
            permut.pop_back();
        }
    }
}
```



Solução – Busca completa (recursiva – backtracking)

- Função recursiva que
  - gera todas as permutações do conjunto $\{0, 1, 2, \ldots, n-1\}$ e
  - realiza um processamento para cada permutação

```
vector<int> permut; vector<bool> chosen(n);
```

```
void search() {
    if (permut.size() == n) {
        // Processa permutacao
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            permut.push_back(i);
            search();
            chosen[i] = false;
            permut.pop_back();
        }
    }
}
```

n corresponde ao tamanho do conjunto

Adiciona i à permutação

Gerando permutações de um conjunto

- A biblioteca padrão do C++ disponibiliza a função `next_permutation`, que pode ser usada para gerar as permutações de um conjunto
- O código a seguir é baseado nesta função

Solução – Busca completa

- Código que
  - gera todas as permutações do conjunto $\{0, 1, 2, \ldots, n - 1\}$ e
  - realiza um processamento para cada permutação

```
vector<int> permut;
for (int i = 0; i < n; i++) {
    permut.push_back(i);
}

do {
    // Processa permutacao
} while (next_permutation(permut.begin(), permut.end()));
```

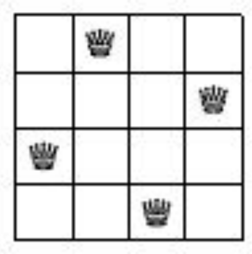
Problema

Descrição adaptada do problema “1624 - Chessboard and Queens” do CSES

- Dado um tabuleiro *n* x *n* de xadrez, de quantas maneiras podemos posicionar *n* rainhas no tabuleiro de forma que nenhuma rainha ataque outra? Restrições: $3 \le n \le 8$.
- **Entrada:** *n*.
- **Saída:** A quantidade requisitada no problema.

Solução – Busca completa (recursiva – backtracking)

- Sabemos que a seguinte condição tem que ser respeitada: duas rainhas não podem estar em uma mesma coluna ou em uma mesmo linha
- Para *n* = 4, no posicionamento ao lado, a rainha da coluna 1 está na linha 0, a rainha da coluna 3 está na linha 1, a rainha da coluna 0 está na linha 2 e a rainha da coluna 2 está na linha 3



- De forma geral, um posicionamento que respeita a condição acima pode ser representado como uma permutação das colunas do tabuleiro
    - Se `permut[i] == j`, então a rainha da coluna *j* está na linha *i*
    - Nesta representação, todas as rainhas estão em linhas e colunas diferentes

Solução – Busca completa (recursiva – backtracking)

- Também sabemos que a seguinte condição tem que ser respeitada: duas rainhas não podem estar em uma mesma diagonal
- Para *n* = 4, podemos representar as colunas, as diagonais principais e as diagonais secundárias do tabuleiro da seguinte maneira:

0	1	2	3
0	1	2	3
0	1	2	3
0	1	2	3

`col`

0	1	2	3
1	2	3	4
2	3	4	5
3	4	5	6

`diag1`

3	4	5	6
2	3	4	5
1	2	3	4
0	1	2	3

`diag2`

- Desta forma, para uma rainha que está na coluna *c* e linha *r*, podemos acessar a diagonal principal e a diagonal secundária onde a rainha está como `diag1[c+r]` e `diag2[c-r+n-1]` (acima, *n* = 4)

Solução – Busca completa (recursiva – backtracking)

- Possível estratégia: gerar todas as permutações do conjunto { 0, 1, 2, . . . , n - 1 } que respeitem à última condição citada e contar a quantidade de permutações
- Complexidade desta estratégia:
  - Existem *n*! permutações
  - No pior caso (n = 8), a complexidade é menor que 2×10^7
- Dado um tempo limite de pelo menos 1 seg., provavelmente podemos resolver o problema com esta estratégia

Solução – Busca completa (recursiva – backtracking)

- Função recursiva que gera todas as permutações do conjunto { 0, 1, 2, . . . , n - 1 } que respeitem à última condição citada e contar a quantidade de permutações

```
void search(int r) {
    if (r == n) {
        count++;
    } else {
        for (int c = 0; c < n; c++) {
            if (col[c] || diag1[c+r] || diag2[c-r+n-1]) continue;
            col[c] = diag1[c+r] = diag2[c-r+n-1] = 1;
            search(r+1);
            col[c] = diag1[c+r] = diag2[c-r+n-1] = 0;
        }
    }
}
```

Divisão e conquista

- **Divisão e conquista** é um método de resolução de problemas que executa os três seguintes passos:
  - Divida o problema em um ou mais subproblemas menores – os subproblemas são instâncias menores do mesmo problema (**divisão**)
  - Resolva os subproblemas (**conquista**)
  - Combine (se necessário) as soluções dos subproblemas para formar uma solução para o problema original

- Um exemplo típico de um algoritmo que segue a estratégia de divisão e conquista é o mergesort

Tarefa

1. Revise o mergesort identificando os três passos da estratégia de divisão e conquista.

Referências

- Esta apresentação é baseada nos seguintes materiais:
  1. Capítulo 3 do livro
     HALIM, S.; HALIM, F.; EFFENDY, S. Competitive Programming 4: The Lower Bound of Programming Contests in the 2020s, book 1, chs. 1-4. Lulu, 2018.
  2. Capítulo 2 do livro
     LAAKSONEN, A. Guide to Competitive Programming: Learning and Improving Algorithms Through Contests, 2. ed. Springer, 2020.