# Paradigmas de Resolução de Problemas

Busca Completa – Definição

Prof. Edson Alves - UnB/FGA 2020

#### Sumário

- 1. Busca Completa
- 2. Conjuntos notáveis

# Busca Completa

#### Definição

- A busca completa, também denominada força bruta, consiste em avaliar todo o espaço de soluções do problema em busca de uma solução
- A complexidade de soluções de busca completa, em geral, são determinadas pelo tamanho do espaço de soluções
- Este espaço tende a ter um grande número de elementos, de modo que a força bruta é aplicada, com eficiência, em problemas cujo contradomínio seja computacionalmente tratável
- Algoritmos de força bruta, por outro lado, tendem a ter uma implementação simples e direta
- Em competições, mesmo que levem a um erro de TLE, estes algoritmos podem servir para testar soluções de menor complexidade assintótica, principalmente nos corner cases

# Exemplo de busca completa: localização de um elemento em um vetor

- A título de ilustração de um algoritmo de busca completa, considere o problema de se identificar se um elemento x está contido ou não em um vetor de elementos  $a = \{a_1, a_2, \dots, a_N\}$
- Se não for imposta nenhuma ordenação subjacente aos elementos de a, a única estratégia viável é a busca completa: olhar, um a um, todos os elementos de a, comparando-os com x, de modo que a complexidade da solução seria O(N)
- Se os elementos de a estão ordenados, é possível melhorar o algoritmo por meio de uma busca binária, obtendo uma complexidade  $O(\log N)$
- Observe que, se for preciso ordenar a a fim de usar a busca binária, a solução teria complexidade  $O(N\log N)$ , de modo que a busca completa seria mais eficiente

# Localização de um elemento por busca completa

```
1 #include <bits/stdc++ h>
3 using namespace std;
5 template<typename T> int find(const T& x, const vector<T>& xs)
6 {
      for (size_t i = 0; i < xs.size(); ++i)</pre>
7
          if (x == xs[i])
               return i;
10
     return -1;
12 }
14 int main()
15 {
     vector<int> xs { 2, 3, 5, 7, 11, 13, 17, 19 }:
16
      cout \ll find(13, xs) \ll '\n' \ll find(9, xs) \ll '\n';
18
      return 0;
20
21 }
```

#### **Geradores** e filtros

- ullet Uma etapa crucial de um algoritmo de busca completa é a geração de todos os elementos do espaço de soluções  ${\cal S}$  do problema
- As soluções que geram explicitamente todos os elementos de S, e então checam cada um destes elementos em busca da solução, são denominadas filtros
- Outra abordagem seria, na geração dos elementos de S, tentar construir diretamente aqueles que correspondem à uma solução do problema, ignorando aqueles que não possam a constituir uma solução do problema
- Algoritmos que utilizam esta segunda abordagem s\u00e3o chamados geradores
- Em geral, os filtros s\u00e3o mais f\u00e1ceis de implementar do que os geradores
- Contudo, o tempo de execução dos filtros tende a ser maior, embora a complexidade assintótica possa ser a mesma do gerador equivalente

#### Exemplo de geradores e filtros

- ullet Para ilustrar a diferença entre um gerador e um filtro, considere o problema de listar todos os inteiros positivos menores ou iguais a N que sejam múltiplos ou de a ou de b
- Por exemplo, para N=20, a=3 e b=5 a solução seria  $s=\{3,5,6,9,10,12,15,18\}$
- Uma solução por filtro seria olhar cada um dos elementos  $s\in\mathcal{S}$  e verificar se ele é composto apenas por múltiplo de 3 ou de 5
- Como |S| é muito grande, é mais eficiente olhar individualmente os elementos de A e escolher somente os múltiplos de 3 ou 5
- A solução por gerador seria construir múltiplos m de 3 e 5 diretamente, tomando o cuidado de excluir os elementos duplicados
- As estratégias são distintas, mas a complexidade assintótica de ambas é a mesma:  ${\cal O}(N)$

#### Exemplo de gerador e de filtro

```
1 #include <bits/stdc++.h>
3 using namespace std;
5 vector<int> filter(int N, int a, int b)
6 {
     vector<int> ms;
7
8
     for (int i = 1; i \le N; ++i)
9
          if (i % a == 0 or i % b == 0)
10
              ms.push_back(i);
      return ms;
13
14 }
```

#### Exemplo de gerador e de filtro

```
16 vector<int> generator(int N, int a, int b)
17 {
     vector<int> ms;
18
19
     for (int i = a; i <= N; i += a)
20
         ms.push_back(i);
     for (int i = b; i <= N; i += b)
         if (i % a) // Evita duplicatas
24
             ms.push_back(i);
25
26
     return ms;
28 }
```

Conjuntos notáveis

#### Produto cartesiano

 Dados dois conjuntos A e B, o produto cartesiano A × B de A por B é o conjunto de todos os pares ordenados cujo primeiro elemento pertence a A e o segundo pertence a B, isto é,

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

- Se |A| = m e |B| = n então  $|A \times B| = mn$
- $\bullet$  Por exemplo, se  $A=\{a,b,c\}$  e  $B=\{1,2\}$ , então

$$A\times B=\{(a,1),(a,2),(b,1),(b,2),(c,1),(c,2)\}$$

- Observe que  $A \times B \neq B \times A$  e que  $A \times A = A^2$
- Se ambos conjuntos são finitos, o produto cartesiano pode ser obtido diretamente por meio de um laço duplo

#### Implementação da geração do produto cartesiano

```
1 #include <bits/stdc++.h>
3 using namespace std;
5 template<typename T1, typename T2> vector<pair<T1, T2>>
6 cartesian_product(const vector<T1>& A, const vector<T2>& B)
7 {
     vector<pair<T1, T2>> AB;
8
      for (const auto& a : A)
10
          for (const auto& b : B)
              AB.push back(make pair(a, b)):
      return AB:
14
15 }
```

#### Subconjuntos

- ullet Um conjunto A composto por N elementos tem  $2^N$  subconjuntos
- • Por exemplo, para N=2, os subconjuntos de  $A=\{1,2\}$  são:  $\emptyset,\{1\},\{2\},\{1,2\}$
- É possível listar todos estes conjuntos, em aproximadamente 1 segundo, para  $N \leq 23$ , pois  $2^{23} \approx 10^7$
- As formas de se gerar estes subconjuntos estão vinculadas a forma escolhida para representar estes subconjuntos
- Se cada subconjunto  $S\subset A$  é um vetor contendo os índices dos elementos de A que pertencem a S, a estratégia mais adequada é usar recursão
- ullet Se S corresponde a um vetor de N bits, onde o i-ésimo bit indica a presença ou a ausência do elemento  $a_i$  em S, os subconjuntos podem ser listados diretamente, por meio de um laço

#### Implementação iterativa dos subconjuntos de A

```
1 #include <bits/stdc++ h>
3 using namespace std;
5 void process_subsets(int n, function<void(int)> process)
6 {
     // Cada inteiro s é um subconjunto: o i-ésimo bit de s
     // indica a presença ou ausência do elemento a_i
      for (int s = 0; s < (1 << n); ++s)
q
          process(s);
10
11 }
```

### Implementação iterativa dos subconjuntos de A

```
13 int main()
14 {
15
     int N;
    cin >> N;
16
     // Lista todos os subconjuntos de A = \{1, 2, ..., N\}
18
      process_subsets(N, [N](int s) {
19
          cout << "{ ":
20
          for (int i = 0; i < N; ++i)
              if (s & (1 << i))
                   cout << (i + 1) << " ";
24
          cout << "}\n";
26
     });
28
      return 0;
29
30 }
```

#### Permutações

- Uma permutação de N elementos  $\{a_1, a_2, \dots, a_N\}$  consiste em uma reordenação de seus índices
- ullet Um conjunto A composto por N elementos distintos tem N! permutações distintas
- Por exemplo, para N=3, as permutações de  $A=\{1,2,3\}$  são:  $\{1,2,3\},\ \{1,3,2\},\ \{2,1,3\},\ \{2,3,1\},\ \{3,1,2\}$  e  $\{3,2,1\}$
- É possível listar estas permutações, em aproximadamente 1 segundo, para N=10, pois  $10! \leq 10^7$
- Assim como no caso dos subconjuntos, é possível gerar todas as permutações por meio de recursão
- Contudo, a biblioteca algorithm da linguagem C++ provê duas funções para a geração de permutações: next\_permutation() e prev\_permutation(), baseadas em implementações iterativas

#### Implementação iterativa das permutações de A

```
1 #include <bits/stdc++ h>
₃ using namespace std:
5 void
6 process_permutations(size_t n, function<void(const vector<int>&)> process)
7 {
     vector<int> ns(n);
8
     // \text{ ns} = \{ 1, 2, 3, \dots, n \}
10
      iota(ns.begin(), ns.end(), 1);
      // Para gerar todas as permutações com next_permutation(), o
      // vector ns deve estar inicialmente ordenado
14
      do {
15
          process(ns):
16
      } while (next_permutation(ns.begin(), ns.end()));
18 }
19
```

#### Combinações

- Seja A um conjunto de n elementos
- Uma combinação de m elementos de A é uma sequência de elementos de A  $\{a_{i1}, a_{i2}, \dots, a_{im}\}$  tal que  $i_j < i_k$  se j < k
- ullet O número de combinações de m elementos de A é igual a

$$C_{n,m} = \binom{n}{m} = \frac{n!}{m!(n-m)!}$$

- $\bullet$  As combinações de m elementos de A podem ser geradas recursivamente, a partir de uma modificação na rotina que gera as permutações
- Também é possível usar as funções prev\_permutation() ou next\_permutation() para gerar tais combinações

### Implementação iterativa das combinações

```
#include <bits/stdc++.h>
3 using namespace std;
5 void process_combinations(int n, int m,
                            function<void(const vector<int>&)> process)
6
7 {
     // ns = { 1, 1, ..., 1, 0, 0, ..., 0 }, m 1s, (n - m) zeros
8
     vector<int> ns(m, 1);
9
     ns.resize(n);
10
     // As combinações são geradas em ordem lexicográfica
     // ns[i] = 1 significa que (i + 1) pertence a combinação
      do {
14
          process(ns);
15
      } while (prev_permutation(ns.begin(), ns.end()));
16
17 }
18
```

# Implementação iterativa das combinações

```
19 int main()
20 {
      int N, M;
     cin >> N >> M;
      process_combinations(N, M, [N](const vector<int>& p) {
24
          cout << "( ";
26
          for (int i = 0; i < N; ++i)
              if (p[i])
28
                   cout << i + 1 << " ":
30
          cout << ")\n";
      });
      return 0;
34
35 }
```

## Combinações com repetições

- ullet Seja A um conjunto de n elementos
- Uma combinação de m elementos de A com repetição é uma sequência de elementos de A  $\{a_{i1}, a_{i2}, \ldots, a_{im}\}$  tais que os índices  $i_k$  estão em ordem não-decrescente
- $\bullet\,$  O número de combinações de m elementos de A com repetição é igual a

$$CR_{n,m} = \binom{n}{k} = \binom{n+m-1}{m}$$

- $\bullet$  As combinações de m elementos de A podem ser geradas recursivamente, a partir de uma modificação na rotina que gera as combinações
- Também é possível gerar tais combinações iterativamente, simulando uma soma com vai-um, e saltando os números cuja sequência não obedeça a ordenação não-decrescente

# Implementação iterativa das combinações com repetições

```
1 #include <hits/stdc++ h>
3 using namespace std;
5 vector<vector<int>> combinations_with_repetition(int N, int M)
6 {
     vector<vector<int>> cs:
7
     vector<int> xs(M, 1);
     int pos = M - 1:
9
10
      while (true)
          cs.push_back(xs);
14
          xs[pos]++;
16
          while (pos > 0 and xs[pos] > N)
18
              --pos;
              xs[pos]++;
20
```

# Implementação iterativa das combinações com repetições

```
if (pos == 0 and xs[pos] > N)
              break;
24
          for (int i = pos + 1; i < M; ++i)
26
              xs[i] = xs[pos];
28
          pos = M - 1:
29
30
      return cs;
33 }
34
35 void main()
36 {
     int N = 5, M = 3;
      auto cs = combinations_with_repetition(N, M);
38
      for (auto xs : cs)
40
          for (int i = 0; i < M; ++i)
41
              cout << xs[i] << (i + 1 == M ? '\n' : ' '):
42
43 }
```

#### Referências

- 1. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
- HALIM, Steve; HALIM, Felix. Competitive Programming 3, Lulu, 2013.
- 3. Rosetta Code. Combinations, acesso em 05/09/2019.
- 4. Rosetta Code. Combinations with repetitions, acesso em 12/04/2020.
- RUSKEY, Frank; WILLIAMS, Aaron. The Coolest Way to Generate Combinations, 2009.
- Stack Overflow. Algorithm to Find Next Greater Permutation of a Given String, acesso em 05/09/2019.
- 7. Wikipédia. Combination, acesso em 13/04/2020.