

# Paradigmas de Resolução de Problemas

## Busca Completa – Definição

---

Prof. Edson Alves - UnB/FGA

2019

1. Busca Completa
2. Geradores 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, 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 dos 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>
2
3 using namespace std;
4
5 template<typename T> int find(const T& x, const vector<T>& as)
6 {
7     for (size_t i = 0; i < as.size(); ++i)
8         if (x == as[i])
9             return i;
10
11     return -1;
12 }
13
14 int main()
15 {
16     vector<int> as { 2, 3, 5, 7, 11, 13, 17, 19 };
17
18     cout << find(13, as) << '\n' << find(9, as) << '\n';
19
20     return 0;
21 }
```

# Geradores e filtros

- Uma etapa crucial de um algoritmo de busca completa é a geração de todos os elementos do espaço de soluções  $\mathcal{S}$  do problema
- As soluções que geram explicitamente todos os elementos de  $\mathcal{S}$ , e então checam cada um destes elementos em busca de uma solução, são denominadas filtros
- Outra abordagem seria, na geração dos elementos de  $\mathcal{S}$ , 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 são chamados geradores
- Em geral, os filtros são mais fáceis de implementar do que os geradores
- Contudo, o tempo de execução dos filtros tende a ser maior, embora a complexidade assintótica de ambos possa ser a mesma

## Exemplo de geradores e filtros

- 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  $s = \{3, 5, 6, 9, 10, 12, 15, 18\}$
- O espaço de soluções  $\mathcal{S}$  seriam todos os subconjuntos de  $A = \{1, 2, \dots, N\}$
- Uma solução por filtro seria olhar cada um dos elementos  $s \in \mathcal{S}$  e verificar se ele é composto apenas por 3 ou por 5
- Como  $|\mathcal{S}|$  é muito grande, é mais eficiente olhar todos os elementos de  $A$  e escolher somente aqueles que são 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 sejam distintas, mas a complexidade assintótica de ambas é a mesma:  $O(N)$



## Exemplo de gerador e de filtro

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 vector<int> filter(int N, int a, int b)
6 {
7     vector<int> ms;
8
9     for (int i = 1; i <= N; ++i)
10         if (i % a == 0 or i % b == 0)
11             ms.push_back(i);
12
13     return ms;
14 }
```

## Exemplo de gerador e de filtro

```
16 vector<int> generator(int N, int a, int b)
17 {
18     vector<int> ms;
19
20     for (int i = a; i <= N; i += a)
21         ms.push_back(i);
22
23     for (int i = b; i <= N; i += b)
24         if (i % a)                // Evita duplicatas
25             ms.push_back(i);
26
27     return ms;
28 }
```

## Geradores notáveis

---

# Produto cartesiano

- Dados dois conjuntos  $A$  e  $B$ , o produto cartesiano  $A \times B$  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  $|A \times B| = mn$
- 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>
2
3 using namespace std;
4
5 template<typename T1, typename T2> vector<pair<T1, T2>>
6 cartesian_product(const vector<T1>& A, const vector<T2>& B)
7 {
8     vector<pair<T1, T2>> AB;
9
10    for (const auto& a : A)
11        for (const auto& b : B)
12            AB.push_back(make_pair(a, b));
13
14    return AB;
15 }
```

# Subconjuntos

- 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 é usar a recursão
- 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 recursiva dos subconjuntos de $A$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 void
6 process_subsets(int k, int n, vector<int>& S,
7                 function<void(const vector<int>&)> process)
8 {
9     if (k == n)
10         process(S);
11     else
12     {
13         process_subsets(k + 1, n, S, process);
14         S.push_back(k + 1);
15         process_subsets(k + 1, n, S, process);
16         S.pop_back();
17     }
18 }
19
```

# Implementação recursiva dos subconjuntos de $A$

```
20 int main()
21 {
22     int N;
23     cin >> N;
24
25     // Lista todos os subconjuntos de A = {1, 2, ..., N}
26     vector<int> S;
27
28     process_subsets(0, N, S, [](const vector<int>& s) {
29         cout << "{ ";
30
31         for (const auto x : s)
32             cout << x << " ";
33
34         cout << "}\n";
35     });
36
37     return 0;
38 }
```



# Implementação iterativa dos subconjuntos de $A$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 void process_subsets(int n, function<void(int)> process)
6 {
7     // Cada inteiro s é um subconjunto: o i-ésimo bit de s
8     // indica a presença ou ausência do elemento a_i
9     for (int s = 0; s < (1 << n); ++s)
10         process(s);
11 }
12
```

# Implementação iterativa dos subconjuntos de $A$

```
13 int main()
14 {
15     int N;
16     cin >> N;
17
18     // Lista todos os subconjuntos de A = {1, 2, ..., N}
19     process_subsets(N, [N])(int s) {
20         cout << "{ ";
21
22         for (int i = 0; i < N; ++i)
23             if (s & (1 << i))
24                 cout << (i + 1) << " ";
25
26         cout << "}\n";
27     });
28
29     return 0;
30 }
```

# Permutações

- Uma permutação de  $N$  elementos  $\{a_1, a_2, \dots, a_N\}$  consiste em uma reordenação de seus índices  $\{b_1, b_2, \dots, b_N\}$  tal que, para cada  $i = 1, 2, \dots, N$ , existe um  $j$  tal que  $a_i = b_j$
- Um conjunto  $A$  composto por  $N$  elementos 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 recursiva das permutações de $A$

```
5 void
6 process_permutations(size_t n, int chosen, vector<int>& P,
7                       function<void(const vector<int>&)> process)
8 {
9     if (P.size() == n)
10         process(P);
11     else
12     {
13         for (size_t i = 1; i <= n; ++i)
14         {
15             if (chosen & (1 << i))
16                 continue;
17
18             chosen |= (1 << i);
19             P.push_back(i);
20             process_permutations(n, chosen, P, process);
21             chosen &= ~(1 << i);
22             P.pop_back();
23         }
24     }
25 }
```

# Implementação iterativa das permutações de $A$

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 void
6 process_permutations(size_t n, function<void(const vector<int>&)> process)
7 {
8     vector<int> ns(n);
9
10    // ns = { 1, 2, 3, ..., n }
11    iota(ns.begin(), ns.end(), 1);
12
13    // Para gerar todas as permutações com next_permutation(), o
14    // vector ns deve estar inicialmente ordenado
15    do {
16        process(ns);
17    } 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  $i_j \leq i_k$  se  $i \leq k$
- O número de combinações de  $m$  elementos de  $A$  é igual a

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

- 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 recursiva das combinações

```
5 void
6 process_combinations(size_t n, size_t m, int chosen, vector<int>& P,
7                       function<void(const vector<int>&)> process)
8 {
9     if (P.size() == m)
10         process(P);
11     else
12     {
13         for (size_t i = P.empty() ? 1 : P.back() + 1; i <= n; ++i)
14         {
15             if (chosen & (1 << i))
16                 continue;
17
18             chosen |= (1 << i);
19             P.push_back(i);
20             process_combinations(n, m, chosen, P, process);
21             chosen &= ~(1 << i);
22             P.pop_back();
23         }
24     }
25 }
```

# Implementação iterativa das combinações

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 void process_combinations(int n, int m,
6                           function<void(const vector<int>&)> process)
7 {
8     // ns = { 1, 1, ..., 1, 0, 0, ..., 0 }, m 1s, (n - m) zeros
9     vector<int> ns(m, 1);
10    ns.resize(n);
11
12    // As combinações são geradas em ordem lexicográfica
13    do {
14        process(ns);
15    } while (prev_permutation(ns.begin(), ns.end()));
16 }
17
```



1. **LAARKSONEN**, Antti. *Competitive Programmer's Handbook*, 2017.
2. **HALIM**, Steve; **HALIM**, Felix. *Competitive Programming 3*, Lulu, 2013.
3. Rosetta Code. [Combinations](#), acesso em 05/09/2019.
4. **RUSKEY**, Frank; **WILLIAMS**, Aaron. [The Coolest Way to Generate Combinations](#), 2009.
5. Stack Overflow. [Algorithm to Find Next Greater Permutation of a Given String](#), acesso em 05/09/2019.