Paradigmas de Resolução de Problemas

Busca Completa – Definição

Prof. Edson Alves – UnB/FGA

Sumário

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 $a = \{a_1, a_2, \ldots, a_N\}$
- 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

```cpp
#include <bits/stdc++.h>

template<typename T>
int find(const T& x, const std::vector<T>& xs)
{
    for (size_t i = 0; i < xs.size(); ++i)
        if (x == xs[i])
            return i;

    return -1;
}

int main()
{
    std::vector<int> xs { 2, 3, 5, 7, 11, 13, 17, 19 };

    std::cout << find(13, xs) << '\n' << find(9, xs) << '\n';

    return 0;
}
```

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 da solução, são denominadas filtros
- Outra abordagem seria, na geração dos elementos de $\mathcal{S}$, tentar construir diretamente aqueles que correspondem a uma solução do problema, ignorando aqueles que não possam 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 possa ser a mesma do gerador equivalente

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 $A = \{1, 2, \ldots, N\}$
- 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 $|\mathcal{S}|$ 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: $O(N)$

Exemplo de gerador e de filtro

```cpp
#include <bits/stdc++.h>

using namespace std;

vector<int> filter(int N, int a, int b)
{
    vector<int> ms;

    for (int i = 1; i <= N; ++i)
        if (i % a == 0 or i % b == 0)
            ms.push_back(i);

    return ms;
}
```

Exemplo de gerador e de filtro

```cpp
vector<int> generator(int N, int a, int b)
{
    vector<int> ms;

    for (int i = a; i <= N; i += a)
        ms.push_back(i);

    for (int i = b; i <= N; i += b)
        if (i % a)                  // Evita duplicatas
            ms.push_back(i);

    return ms;
}
```

Conjuntos 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

```cpp
#include <bits/stdc++.h>

using namespace std;

template<typename T1, typename T2>
vector<pair<T1, T2>> cartesian_product(const vector<T1>& A, const vector<T2>& B)
{
    vector<pair<T1, T2>> AB;

    for (const auto& a : A)
        for (const auto& b : B)
            AB.emplace_back(a, b);

    return AB;
}
```

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 mais adequada é usar 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 iterativa dos subconjuntos de A

```cpp
#include <bits/stdc++.h>

using namespace std;

void process_subsets(int n, function<void(int)> process)
{
    // 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)
        process(s);
}
```

Implementação iterativa dos subconjuntos de A

```cpp
int main()
{
    int N;
    cin >> N;

    // Lista todos os subconjuntos de A = {1, 2, ..., N}
    process_subsets(N, [N](int s) {
        cout << "{ ";

        for (int i = 0; i < N; ++i)
            if (s & (1 << i))
                cout << (i + 1) << " ";

        cout << "}\n";
    });

    return 0;
}
```

Permutações

- Uma permutação de N elementos $\{a_1, a_2, \ldots, a_N\}$ consiste em uma reordenação de seus índices
- 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

```
#include <bits/stdc++.h>

using namespace std;

void
process_permutations(size_t n, function<void(const vector<int>&)> process)
{
    vector<int> ns(n);

    // ns = { 1, 2, 3, ..., n }
    iota(ns.begin(), ns.end(), 1);

    // Para gerar todas as permutações com next_permutation(), o
    // vector ns deve estar inicialmente ordenado
    do {
        process(ns);
    } while (next_permutation(ns.begin(), ns.end()));
}
```

Combinações

- Seja A um conjunto de n elementos
- Uma combinação de m elementos de A é uma sequência de elementos de A $\{a_{i1}, a_{i2}, \ldots, a_{im}\}$ tal que $i_j < i_k$ se $j < k$
- 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 iterativa das combinações

```
#include <bits/stdc++.h>

using namespace std;

void process_combinations(int n, int m, function<void(const vector<int>&)> process)
{
    // ns = { 1, 1, ..., 1, 0, 0, ..., 0 }, m 1s, (n - m) zeros
    vector<int> ns(m, 1);
    ns.resize(n);

    // As combinações são geradas em ordem lexicográfica
    // ns[i] = 1 significa que (i + 1) pertence a combinação
    do {
        process(ns);
    } while (prev_permutation(ns.begin(), ns.end()));
}
```

Implementação iterativa das combinações

```
int main()
{
    int N, M;
    cin >> N >> M;

    process_combinations(N, M, [N](const vector<int>& p) {
        cout << "( ";

        for (int i = 0; i < N; ++i)
            if (p[i])
                cout << i + 1 << " ";

        cout << ")\n";
    });

    return 0;
}
```

Combinações com repetições

- 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
- O número de combinações de m elementos de A com repetição é igual a

$$CR_{n,m} = \left(\!\!\binom{n}{k}\!\!\right) = \binom{n+m-1}{m}$$

- 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

```
#include <bits/stdc++.h>

using namespace std;

vector<vector<int>> combinations_with_repetition(int N, int M)
{
    vector<vector<int>> cs;
    vector<int> xs(M, 1);
    int pos = M - 1;

    while (true)
    {
        cs.push_back(xs);

        xs[pos]++;

        while (pos > 0 and xs[pos] > N) {
            --pos;
            xs[pos]++;
        }
```

Implementação iterativa das combinações com repetições

```
        if (pos == 0 and xs[pos] > N)
            break;

        for (int i = pos + 1; i < M; ++i)
            xs[i] = xs[pos];

        pos = M - 1;
    }

    return cs;
}

int main() {
    int N = 5, M = 3;
    auto cs = combinations_with_repetition(N, M);

    for (auto xs : cs)
        for (int i = 0; i < M; ++i)
            cout << xs[i] << (i + 1 == M ? '\n' : ' ');
}
```

Referências

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. Rosetta Code. Combinations with repetitions, acesso em 12/04/2020.
5. **RUSKEY**, Frank; **WILLIAMS**, Aaron. The Coolest Way to Generate Combinations, 2009.
6. 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.