Análise Combinatória

Permutações

Prof. Edson Alves

Faculdade UnB Gama

Sumário

Permutações

Princípio Multiplicativo

- O princípio multiplicativo está relacionado ao número de elementos do produto cartesiano de dois conjuntos
- Se A e B são dois conjuntos finitos não vazios, com $|A| = n$ e $|B| = m$, então o produto cartesiano $A \times B$ terá nm elementos
- Este princípio é útil em contagem de n-uplas de elementos, onde o i-ésimo elemento da n-upla vem do i-ésimo conjunto C_i
- Deste princípio derivam os conceitos de permutação, arranjo e combinação

Permutações

Definição de permutação

Seja A um conjunto com n elementos distintos. Uma **permutação** dos elementos de A consiste em um ordenação destes elementos tal que duas permutações são distintas se dois ou mais elementos ocuparem posições distintas.

Por exemplo, se $A = \{1, 2, 3\}$, há 6 permutações distintas, a saber:

$$123, 132, 213, 231, 312, 321$$

Cálculo do número de permutações

- Considere um conjunto com n elementos distintos
- Para a primeira posição há n escolhas possíveis
- Para a segunda, $(n-1)$ escolhas, uma vez que o primeiro elemento já foi escolhido
- Pelo mesmo motivo, há $(n-2)$ escolhas para o terceiro elemento, e assim sucessivamente, até restar uma única escolha para o último elemento
- Portanto,

$$P(n) = n \times (n-1) \times (n-2) \times ... \times 2 \times 1 = n!$$

Caracterização das permutações

- Em combinatória é útil associar os conceitos de contagem à situações práticas e tentar encontrar soluções por analogia
- As permutações, por exemplo, podem ser visualizadas como a retirada de n bolas distintas de uma caixa, sem reposição
- Veja que tanto as bolas quanto a ordem de retirada importam, no sentido que duas permutações são distintas se a ordem de alguma das bolas é diferente

Permutações com repetição

- Um permutação com repetição consiste em uma ordenação de n elementos, não necessariamente distintos
- Considere um conjunto de k elementos distintos, onde cada um deles ocorre n_i vezes, com $i = 1, 2, \ldots, k$, de forma que $n_1 + n_2 + \ldots + n_k = n$
- Dentre as $n!$ permutações dos n elementos, várias delas serão repetidas
- De fato, como o elemento i se repete n_i vezes, uma permutação p em particular se repetirá $n_i!$ vezes

Permutações com repetição

- Isto porque todas as permutações de posições dentre as cópias de i levam a uma mesma permutação
- Por exemplo, para o conjunto $A = \{1, 2, 1\}$, apenas 3 das 6 permutações são distintas: $112, 121$ e 211
- Assim, o número de permutações distintas, com repetições, é dado por

$$PR(n; n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! \times n_2! \times \ldots \times n_k!}$$

Implementação da permutação com repetições em C++

```cpp
template<typename T>
long long permutations(const vector<T>& A)
{
    map<T, int> hist;

    for (auto a : A)
        ++hist[a];

    long long res = factorial(A.size());

    for (auto [a, ni] : hist)
        res /= factorial(ni);

    return res;
}
```

Permutações circulares

- Se, em uma permutação, os objetos devem ser dispostos em uma formação circular, sem uma marcação clara de início de fim, algumas permutações se tornam idênticas, a menos de uma rotação
- Para contabilizar apenas as permutações que não podem ser geradas a partir de rotações das demais, é preciso fixar um elemento em uma dada posição e permutar os demais nas posições restantes
- Deste modo, o número de permutações circulares de n elementos distintos é dado por

$$PC(n) = P(n-1) = (n-1)!$$

Enumeração das permutações

- É possível enumerar todas as possíveis permutações de n elementos por meio de *backtracking*
- A função `next_permutation()` da biblioteca `algorithm` do C++ também enumera as permutações distintas
- Ela retorna verdadeiro se é possível gerar a próxima permutação, na ordem lexicográfica, a partir da permutação atual, e falso, caso contrário
- Assim, para enumerar todas as permutações distintas, é preciso começar com a primeira permutação na ordem lexicográfica, que consiste em todos os elementos ordenados

prev_permutation()

- A biblioteca `algorithm` também contém a função `prev_permutation()`, que também enumera permutações
- Contudo, ela o faz em sentido oposto em relação à `next_permutation()`
- Assim, para listar todas as permutações distintas usando `prev_permutation()`, é preciso iniciar na última permutação, segundo a ordem lexicográfica
- Ambas funções tem complexidade $O(N)$

Exemplo de enumeração das permutações em C++

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

int main()
{
    vector<int> A { 5, 3, 4, 1, 2 };

    sort(A.begin(), A.end());          // Primeira permutação na ordem lexicográfica

    do {
        for (size_t i = 0; i < A.size(); ++i)
            cout << A[i] << (i + 1 == A.size() ? '\n' : ' ');
    } while (next_permutation(A.begin(), A.end()));

    return 0;
}
```

Permutações em competições

Permutações em competições

- Listar todas as permutações tem complexidade $O(n \times n!)$
- A enumeração de todas as permutações só é viável para valores pequenos de n (por exemplo, $n \approx 10$)
- Em problemas que envolvam permutações sujeitas a uma série de restrições, caso seja possível, listar todas elas e filtrá-las individualmente é mais simples de implementar do que computar as permutações desejadas diretamente

Problemas propostas

1. AtCoder Beginner Contest 103A – Task Scheduling Problem
2. AtCoder Beginner Contest 123B – Five Dishes
3. Codeforces 222B – Cosmic Tables
4. Codeforces 961C – Chessboard
5. OJ 216 – Getting in Line

Referências

1. **SANTOS**, José Plínio O., **MELLO**, Margarida P., **MURARI**, Idani T. *Introdução à Análise Combinatória*, Editora Ciência Moderna, 2007.

Soluções dos problemas propostos

AtCoder Beginner Contest 123B – Five Dishes

Versão resumida do problema: determinar a sequência em que os pratos devem ser pedidos para que o tempo total para servir todos eles seja o menor possível.

Restrições:

- somente um prato pode ser servido por vez,
- um prato só pode ser pedido em quando o tempo for um múltiplo de 10, e
- um novo pedido só pode ser feito quando o prato anterior for servido.

Solução com complexidade $O(1)$

- A solução consiste em determinar uma permutação dos pratos A, B, C, D e E que minimize o tempo para serví-los
- Se não houvesse a restrição de que os pratos só podem ser pedidos em instantes de tempo que são múltiplos de 10, qualquer permutação levaria ao mesmo resultado
- Há $5! = 120$ permutações possíveis, as quais podem ser geradas por meio da função `next_permutation()`
- O i-ésimo prato requer t_i minutos para ser servido após ser pedido e, exceto pelo último, é preciso esperar o próximo múltiplo de 10 para ser pedido
- Uma forma de tratar esta espera é substituir t_i pelo menor múltiplo de 10 maior ou igual a t_i para todos os pratos, exceto o último

Solução com complexidade $O(1)$

```
int solve(vector<int> xs)
{
    sort(xs.begin(), xs.end());
    int ans = oo;

    do {
        int t = xs.back();

        for (int i = 0; i < 4; ++i)
            t += 10 * ((xs[i] + 9)/10);

        ans = min(ans, t);
    } while (next_permutation(xs.begin(), xs.end()));

    return ans;
}
```

OJ 216 – Getting in Line

Versão resumida do problema: determine uma permutação de N computadores tal que o comprimento total do cabo necessário para interligar estes computadores, por meio conexões diretas entre os computadores adjacentes na permutação, seja mínimo.

Restrições:

- $2 \leq N \leq 8$, e
- o comprimento do cabo necessário para conectar dois computadores adjacentes na permutação é igual a distância euclidiana entre eles mais 16 metros.

Solução em $O(N \times N!)$

- Como o número total de computadores é relativamente pequeno, é possível resolver este problemas por meio da enumeração e avaliação de todas as $N!$ permutações distintas
- Cada permutação pode ser avaliada em $O(N)$, de modo que esta solução terá complexidade $O(N \times N!)$
- A distância euclidiana entre os pontos $P = (x_P, y_P)$ e $Q = (x_Q, y_Q)$ é dada por

$$d(P, Q) = \sqrt{(x_P - x_Q)^2 + (y_P - y_Q)^2}$$

- É preciso acrescentar, a cada distância entre pares de computadores, 16 metros, conforme instrui o texto do problema

Solução

```
struct Point { int x, y; };

struct Answer {
    double min_dist;
    vector<int> ps;
    vector<vector<double>> dist;
};

Answer solve(int N, const vector<Point>& ps) {
    vector<vector<double>> dist(N, vector<double>(N));

    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
            dist[i][j] = hypot(ps[i].x - ps[j].x, ps[i].y - ps[j].y) + 16.0;

    vector<int> is(N), ans;
    iota(is.begin(), is.end(), 0);
```

Solução

```
    double min_dist = 1e30;

    do {
        double d = 0.0;

        for (int i = 1; i < N; ++i)
            d += dist[is[i - 1]][is[i]];

        if (d < min_dist)
        {
            min_dist = d;
            ans = is;
        }
    } while (next_permutation(is.begin(), is.end()));

    return { min_dist, ans, dist };
}
```

Codeforces 222B – Cosmic Tables

Versão resumida do problema: dada uma matriz $A_{n \times m}$, responda k consultas de um dos 3 tipos abaixo:

- troque duas linhas de lugar,
- troque duas colunas de lugar, e
- imprima um elemento da matriz.

Restrições:

- $1 \leq n, m \leq 1000$
- $1 \leq k \leq 500000$

Solução com complexidade $O(k)$

- Trocar efetivamente os elementos de duas linhas de lugar tem complexidade $O(m)$
- De forma equivalente, a troca de duas colunas tem complexidade $O(n)$
- Assim, no pior caso o algoritmo teria complexidade $O(k \times \max(n, m))$, o que extrapolaria o limite de tempo, dadas as restrições do problema
- A solução do problema depende, portanto, de um processamento mais eficiente das consultas que envolvem trocas de linhas e de colunas
- De fato, estas consultas podem ser respondidas em $O(1)$

Solução com complexidade $O(k)$

- A cada troca de linhas ou de colunas, a nova matriz obtida tem as mesmas linhas e colunas da matriz A, porém em ordem distinta
- A ideia, portanto, é utilizar duas permutações, denominadas `rs` e `cs`, que registrem a ordem em que as linhas e as colunas da matriz A foram rearranjadas até uma consulta de elemento
- Inicialmente, ambas permutações são identidades, isto é, `rs[i] = i` e `cs[j] = j` para $i \in [1, n]$ e $j \in [1, m]$
- Com estas permutações, a troca de linhas ou de colunas é feita apenas pela troca dos elementos nas respectivas permutações, mantendo a matriz A inalterada
- Nas consultas de elementos, basta utilizar os índices registrados nas permutações para localizar o elemento correto na matriz A

Solução com complexidade $O(k)$

```cpp
vector<int> solve(const vector<vector<int>>& A, int N, int M, const vector<Query>& qs)
{
    vector<int> rs(N + 1), cs(M + 1), ans;

    iota(rs.begin(), rs.end(), 0);
    iota(cs.begin(), cs.end(), 0);

    for (const auto& q : qs)
    {
        switch (q.c.front()) {
        case 'c':
            swap(cs[q.x], cs[q.y]);
            break;

```

Solução com complexidade $O(k)$

```
        case 'r':
            swap(rs[q.x], rs[q.y]);
            break;

        default:
            ans.push_back(A[rs[q.x]][cs[q.y]]);
        }
    }

    return ans;
}
```