MCZA038

Prática Avançada de Programação A

Combinatória

Prof. Alexandre Donizeti Alves

Universidade Federal do ABC

Bacharelado em Ciência da Computação

Terceiro Quadrimestre - 2017

Problemas Combinatórios

- **Combinatória** é a matemática do contar
  - Existem diversos problemas básicos de contagem que ocorrem repetidamente em ciência da computação e em programação.
- **Problemas Combinatórios** são aqueles em que uma solução é a combinação de um subconjunto de elementos
  - São famosos por sua relação com a “esperteza” e *insights*;
  - Uma vez que se coloca o problema sob a perspectiva certa, a solução pode se tornar óbvia.

Problemas Combinatórios

- O **espaço de busca** de um problema combinatório é o conjunto de todas as soluções possíveis, podendo ser restrito as soluções viáveis ou não;
- Uma **solução viável** é uma solução completa, que atende aos requisitos do problema;
- Uma **solução inviável** é uma solução incompleta ou mesmo uma solução nula.

Problemas Combinatórios

- Ainda, podemos ter **soluções equivalentes**, ou seja, mais de uma solução viável para o mesmo problema;
- Alguns problemas possuem apenas uma solução melhor que todas as outras, chamada de **solução ótima**.

Estratégias de Projeto de Algoritmos

- Existem diferentes maneiras de se projetar um algoritmo para solução de um determinado problema combinatório;
- Ao projetar um algoritmo, pensamos no problema e em suas características
  - Que tipo de algoritmo teria melhor desempenho para este problema?

Estratégias de Projeto de Algoritmos

- Ao tratarmos de problemas difíceis, como os NP-Completos, sabemos que não são conhecidas técnicas eficientes para a sua resolução
  - Então, podemos analisar as características da entrada para o problema para obter alguma vantagem no projeto de uma solução?

Tentativa e Erro

- **Tentativa e Erro**, **Força Bruta** ou **Busca Exaustiva** é a estratégia mais trivial e intuitiva para solução de problemas
  - Consiste em enumerar todas as combinações possíveis para uma solução e avaliar se satisfazem ao problema
    - Escolhe a melhor das soluções, ou **solução ótima**.
  - Apesar de trivial, possui desempenho muito ruim.

Tentativa e Erro

- Consideremos o **Problema da Mochila**:
  - Dada uma mochila com capacidade **C**, e ***n*** objetos com peso $\mathbf{p_i}$ (i=1…*n*), o objetivo é preencher a mochila com o maior peso total, respeitando a capacidade **C**.

Tentativa e Erro

- Suponha uma mochila de capacidade **15 kg** e objetos de peso **12 kg**, **2 kg**, **4 kg** e **8 kg**;
- Este problema possui mais que uma solução ótima, ou seja, possui soluções ótimas equivalentes:
  - 12 kg + 2 kg;
  - 8 kg + 2 kg + 4 kg.
- **Soluções viáveis** seriam, entre outras:
  - 12 kg, 2 kg, 4 kg, 8 kg, 2kg + 4 kg, *etc*.
- Uma **solução inviável** seria
  - 12 kg + 4 kg.

Tentativa e Erro

- Um método baseado em tentativa e erro testaria todas as combinações entre elementos checando:
  - Se a solução é viável;
  - Se a solução possui valor melhor que outra encontrada anteriormente.
- Para determinar se uma solução é a melhor de todas desta forma, é necessário **enumerar** todas.

Tentativa e Erro

- Consideremos agora, o problema decifrar uma senha que contém apenas números:
  - Só existe uma solução ótima;
  - Não são consideradas soluções parciais.

Tentativa e Erro

- Um método baseado em tentativa e erro:
  - Enumera todas as combinações dos números 0-9 para cada dígito da senha;
  - Quanto maior a senha, pior o desempenho;
  - Poderia ser utilizado um método que levasse em consideração alguma estatística sobre composição de senhas e dar prioridade às mais prováveis.

Tentativa e Erro

- Dependendo da característica do problema caímos em uma das situações para tentativa e erro:
  - Gerar todas as **Combinações**;
  - Gerar todos os **Arranjos**;
  - Gerar todas as **Permutações**.

Permutação

- Dado um conjunto ***U*** com ***n*** elementos, uma **permutação** de seus elementos é uma ordenação dos mesmos;
- A quantidade de permutações possíveis para n elementos é definida pelo fatorial da cardinalidade do conjunto
  - Ou seja $n!$;
  - Que equivale a $n \times (n-1) \times (n-2) \times \ldots \times 1$.

Permutação

- Por exemplo, o conjunto *U*={1, 2, 3} de cardinalidade 3, possui 6 permutações:
  - **1 2 3**;
  - 1 3 2;
  - 2 1 3;
  - 2 3 1;
  - 3 1 2;
  - **3 2 1**.

Gerando Permutações

- A geração de todas as permutações pode ser feita seguindo uma ordem lógica
  - Estabelecendo uma relação entre uma permutação e a próxima;
  - Desta forma, é possível numerar cada uma das permutações dentro da sequência gerada;
    - Dada uma determinada posição dentro da sequência de todas permutações, é possível determinar qual é a permutação correspondente (*rank*);
    - Ainda, dada uma permutação, é possível determinar qual é sua posição (número) dentro da sequência de todas as permutações (*unrank*).

Gerando Permutações

- Na linguagem C++ a função **next_permutation()** gera permutações de um vetor
  - Em ordem lexicográfica;
    - A partir de uma permutação, gera a próxima em ordem lexicográfica crescente e *true*;
    - Caso não haja tal permutação, gera a menor permutação em ordem lexicográfica e *false*;
    - Não gera permutações de repetições.

Gerando Permutações

```
#include <algorithm>
#include <cstdio>

using namespace std;

int main()
{
    char xs[] = "AAA";
    do
    {
        std::puts(xs);
    }
    while (std::next_permutation(xs, xs + sizeof(xs) - 1));

    return 0;
}
```

Gerando Permutações

- Como dito anteriormente, o número de permutações é fatorial no número de elementos
  - O que significa crescimento muito rápido.
- Diferentes algoritmos de geração de permutações possuem velocidades diferentes
  - Muito depende da complexidade da relação entre uma permutação e a próxima.
- Um algoritmo de bom desempenho é o ***Trotter-Johnson***.

Gerando Permutações

- O algoritmo *Trotter-Johnson* (ou *Steinhaus–Johnson–Trotter*) gera permutações de troca mínima:
  - A partir de uma permutação, apenas dois elementos são trocados de lugar (*i*. *e*., transpostos) para gerar a próxima permutação.

1 2 3

1 3 2

3 1 2

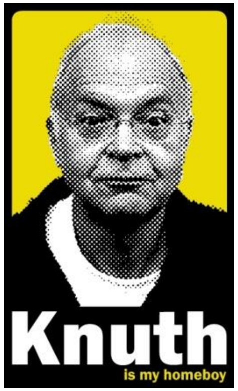
3 2 1

2 3 1

2 1 3

Gerando Permutações

- Curiosamente, este algoritmo foi proposto independentemente por Trotter (1962) e Johnson (1963), mas ambos levaram o crédito;
- Porém, Donald Knuth, no terceiro volume do "*The Art of Programming*" diz que nem um nem outro. Tocadores de sinos, segundo registrado por um livro antigo teriam desenvolvido o algoritmo.



Combinação

- Dado um conjunto ***U*** com ***n*** elementos, uma **combinação sem repetição** indica quantos subconjuntos diferentes de ***s*** elementos do conjunto ***U*** podem ser formados, não considerando a ordem dos elementos;
- Denotada por

$$C_s^n = \frac{n!}{s!(n-s)!}$$

Combinação

- Por exemplo, o conjunto ***U*** = {A, B, C, D} possui 6 subconjuntos de dois elementos, são eles:
  - AB, AC, AD, BC, BD, CD.

Arranjo

- Dado um conjunto ***U*** com ***n*** elementos, um **arranjo simples** indica quantas ordenações de dos elementos de ***U*** tomados ***r*** a ***r*** são possíveis, em que se diferencia a ordem dos elementos, sem repetições;
- Denotada por

$$A_r^n = \frac{n!}{(n-r)!}$$

Arranjo

- Dado um conjunto ***U*** com ***n*** elementos, um **arranjo com repetições** indica quantas ordenações de dos elementos de ***U*** tomados ***r*** a ***r*** são possíveis, em que se diferencia a ordem dos elementos, com repetições;
- Denotada por

$$A_r^n = n^r$$

Arranjo

- Por exemplo, o conjunto ***U*** = {A, B, C, D} possui 12 arranjos simples e 16 arranjos com repetição
  - AB,AC,AD,BA,BC,BD,CA,CB,CD,DA,DB,DC;
  - **AA**,AB,AC,AD,BA,**BB**,BC,BD,CA,CB,**CC**,CD,DA,DB,DC,**DD**.

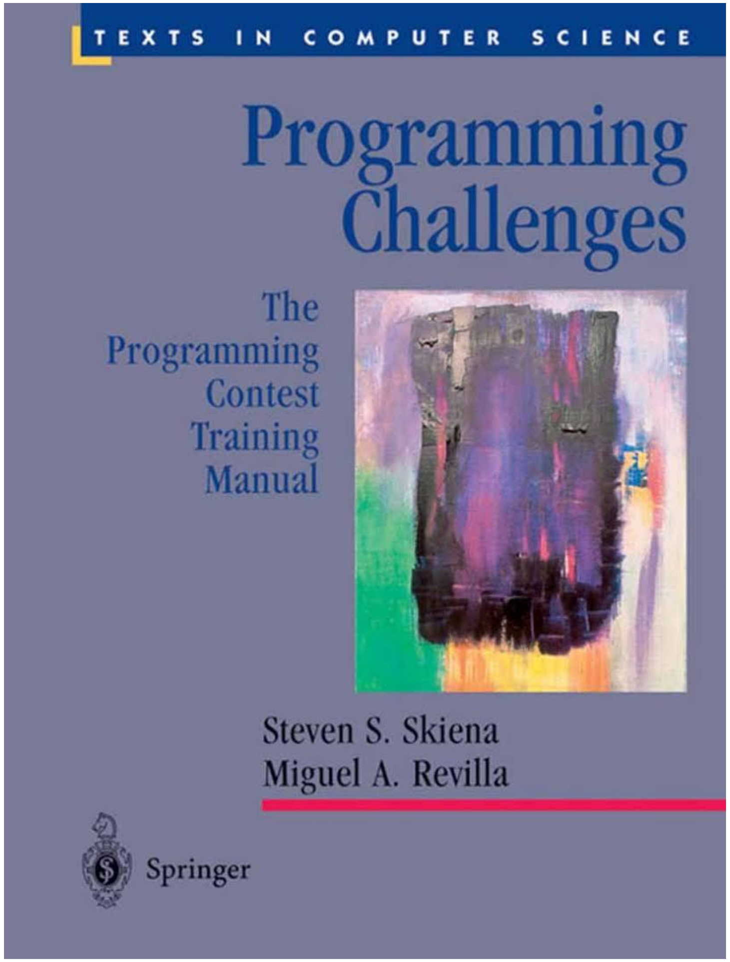
Geração de Subconjuntos

- Através de ***backtracking***, é possível gerar todos os subconjuntos e também todas as permutações;

Agradecimentos

- Slides baseados nas aulas dos professores **Túlio A. M. Toffolo** (**UFOP**) e **Marco Antonio M. Carvalho** (**UFOP**)

Referência


TEXTS IN COMPUTER SCIENCE
Programming Challenges
The Programming Contest Training Manual
Steven S. Skiena
Miguel A. Revilla
Springer