Habilidades BÁSICAS de processamento de conjuntos

HABILIDADE 1:

Na computação é muito comum o problema de ***gerar todos os subconjuntos*** de um determinado conjunto com *n* elementos. Sabemos que a quantidade de subconjuntos é igual a $2^{|n|}$, ou seja, 2 elevado ao número de elementos do conjunto (incluindo o conjunto vazio e o próprio conjunto como um todo). Sua tarefa é descobrir, para C++ ou C (a linguagem de sua escolha), um algoritmo recursivo capaz de gerar todos os subconjuntos de um conjunto de *n* elementos. Por exemplo: dado o conjunto A = {1, 2, 3}, como *n* = 3, temos $2^3 = 8$ subconjuntos, que são:

```
{}
{1}
{2}
{3}
{1, 2}
{1, 3}
{2, 3}
{1, 2, 3}
```

Programe e execute um algoritmo recursivo para isso e verifique se:

a) Seu programa está funcionando corretamente
b) Você sabe explicar como o algoritmo funciona
c) Você sabe a complexidade (Big-O) do algoritmo

HABILIDADE 2:

Outra habilidade importante na computação é a capacidade de ***gerar todas as permutações*** de uma coleção de *n* elementos. Suponha que você tenha a coleção (1, 2, 3) e quer gerar todas as permutações dessa coleção. Fazendo manualmente encontramos 6 permutações:

```
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

Ocorre que há um problema: ao fazer as permutações temos que separar 2 casos diferentes: a) se a coleção não tem elementos repetidos, e b) se a coleção tem elementos repetidos.

- Se a coleção **não tem** elementos repetidos, o número total de permutações é dado por *P(n)*, onde:

$$P(n) = n!$$

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

- Se a coleção **tem um ou mais elementos repetidos**, o número total de permutações é dado por $P(n; n_1, n_2, \ldots, n_k)$, onde:

$$P(n; n_1, n_2, ..., n_k) = \frac{n!}{n_1! \times n_2! \times \cdots \times n_k!}$$

n = número total de elementos
$n_1, n_2, \ldots, n_k$ = quantidades de repetições de cada elemento

Por exemplo: a coleção (AABB) terá apenas 6 permutações:
(AABB), (ABAB), (ABBA), (BAAB), (BABA), (BBAA), (ABBA)

Sua tarefa é descobrir, para C++ ou C (a linguagem de sua escolha), um algoritmo recursivo capaz de gerar todas as permutações de uma coleção de *n* elementos que funcione corretamente nas duas situações: quando a coleção não tem elementos repetidos e quando a coleção tem elementos repetidos.

Programe e execute um algoritmo recursivo para isso e verifique se:

a) Seu programa está funcionando corretamente
b) Você sabe explicar como o algoritmo funciona
c) Você sabe a complexidade (Big-O) do algoritmo

HABILIDADE 3:

Frequentemente na computação temos que ser capazes de **contar de quantos modos um problema pode ser resolvido**. Um exemplo clássico é o problema das *n*-Rainhas (genericamente chamado de n-Queens), e consiste basicamente em descobrir de quantas maneiras é possível colocar *n* rainhas em um tabuleiro de xadrez de tamanho *n* x *n*, de forma que nenhuma rainha consiga atacar a outra. O método clássico para resolver o problema das *n*-Rainhas é usar um algoritmo de *backtracking* recursivo. Sua tarefa é pesquisar sobre o algoritmo de *backtracking* recursivo para resolver o problema das *n*-Rainhas, mas não apenas para resolver o problema: seu algoritmo deve ser capaz de achar todas as soluções possíveis e contar quantas são.

Programe e execute um algoritmo de *backtracking* recursivo para isso e verifique se:

a) Seu programa está funcionando corretamente
b) Você sabe explicar como o algoritmo funciona
c) Você sabe a complexidade (Big-O) do algoritmo

HABILIDADE 4:

Para resolver problemas sofisticados de teoria dos números e/ou de conjuntos, uma habilidade que deve ser dominada é a **manipulação de bits** armazenados na memória, pois isso tem muitas aplicações como, por exemplo, representar todos os elementos de um conjunto. Você deve aprender a realizar operações tais como:

bitwise AND
bitwise OR
bitwise XOR
bitwise NOT
bit shifts (para a esquerda e para a direita)
bit masks

Sua tarefa é descobrir como realizar operações bit a bit (bitwise) em C++ ou C, e criar um programa que consiga representar conjuntos de *n* elementos como um número inteiro de *n* bits. Descubra como implementar as operações de interseção, união, complemento e diferença através da manipulação de bits.

Programe e execute um algoritmo que manipula bits para representar elementos de um conjunto e as diversas operações que podem ser realizadas (união, interseção, etc.) e verifique se:

a) Seu programa está funcionando corretamente
b) Você sabe explicar como o algoritmo funciona
c) Você sabe a complexidade (Big-O) do algoritmo