Algoritmos de Enumeração

O sentido de enumeração é listar ou gerar todos os objetos de um determinado conjunto.
Dizemos que estamos enumerando objetos, ou gerando uma lista de objetos com uma determinada característica.

Existem várias aplicações para algoritmos deste tipo. Uma delas é na simulação de dados para testar um programa. Em muitos casos para fazer a simulação de um algoritmo é necessário testar-se com um conjunto exaustivo de dados, ou seja, gerar várias ou todas as sequências possíveis de dados e verificar o comportamento do algoritmo para estas sequências. O teste automático é bastante interessante, pois podemos ter um programa que simule todas as possíveis sequências de dados que serão submetidas a este algoritmo.

O caso geral é a geração de todos os objetos de um conjunto para submetê-los a algum teste de hipótese.

Sequências

Suponha o seguinte problema:
Gerar todas as sequências possíveis de 3 dígitos com os dígitos 0, 1 e 2.
Solução: 000, 001, 002, 010, 011, 012, 020, 021, 022, 100, 101, 102,... 220, 221, 222.
A quantidade é de 3^3=27 sequências.

E se fosse com 3 dígitos e com os dígitos 0 a 9.
Seriam todas as sequências 000,..., 999.
A quantidade é de 10^3=1000 sequências.

E se fosse sequências com 5 dígitos com os dígitos 0, 1 e 2.
Seriam as sequências 00000,..., 22222.
A quantidade é de 3^5=243 sequências.

Genericamente n posições e m algarismos possíveis em cada posição.
A quantidade é de m^n sequências.

Esse problema é equivalente a escrever todos os números de n algarismos na base m.
Basta começar com o menor possível 00... 0 (n dígitos) e somar 1 na base m no último algarismo levando em conta o "vai um" para todos os dígitos.
Estamos falando em algarismos de 0 a 9 (na base 10), ou algarismo de 0 a m-1 (na base m), mas poderiam ser objetos quaisquer.

```
objetos = [None] * m
```
Onde `objetos[i]` é o objeto associado ao algarismo `i`.

A função `Proxima(a, N, M)` abaixo gera a próxima sequência àquela que está na lista a. A função `Imprime_Sequencias(N, M)` abaixo, imprime todas as sequências, ou todos os números com n dígitos na base m.

```
# Devolve a próxima sequência de N dígitos na base M
# àquela da lista a - Devolve na própria lista a nova
# Devolve False se não mais nada a gerar
def Proxima(a, N, M):
    t = N - 1
    # Soma 1 (base M) à lista a
    while t >= 0:
        a[t] = (a[t] + 1) % M
        if a[t] == 0: t -= 1
        else: return True
    # Se t ficou < 0 então era a última
    return False

# Imprime todas as sequencias de N elementos com
# os dígitos 0 a M-1
def Imprime_Sequencias(N, M):
    # inicia seq
    seq = [0] * N
    # Imprime sequência atual e calcula a próxima
    tem_proxima = True
    cont = 0
    while tem_proxima:
        print("\n%05d - " %cont, end = '')
        for k in range(N): print("%3d" %seq[k], end = '')
        # Gera a próxima
        cont += 1
        tem_proxima = Proxima(seq, N, M)
```

Outra forma de resolver este problema é gerar todos os números de 0 até $\mathbf{m^n - 1}$ na base **m**, colocando cada dígito em um elemento do vetor **seq[0..n-1]**.

```
for i in range(m^n):
   # Transforme i para a base m, usando n dígitos e
   # Coloque cada um dos n dígitos em seq[0..n-1]
   ...
```

Exercício:

1) Completar a solução usando o algoritmo acima.

2) A solução acima monta as sequências na ordem crescente nos números na base m. Escreva o algoritmo somando 1 aos dígitos mais significativos em primeiro lugar.
   Exemplo para n = 3 e m = 2:
   000, 100, 010, 110, 001, 101, 011, 111.

3) Outra ordem é subtrair 1 à anterior em vez de somar 1.
   Exemplo para n = 3 e m=2
   111, 110, 101, 100, 011, 010, 001, 000

O problema da Mochila – um exemplo

Dado um conjunto de itens, cada um com um peso e um valor, determinar a quantidade de cada item a incluir numa mochila de tal forma que o peso total esteja limitado à capacidade da mochila e cujo valor total seja o maior possível.

A função recebe o número de itens n, uma lista com os seus pesos (pesos[0..n-1]), uma lista com os seus valores (valores(0..n-1] e a capacidade máxima da mochila peso_max. Devolve uma lista de quantidades (quantidade[0..n-1]) com a quantidade de cada item para que o valor contido na mochila seja máximo.

```
def Mochila (n, pesos, valores, peso_max):
```

No caso geral, outras restrições podem se aplicar, como quantidade mínima e máxima de cada item que a mochila deve conter, conseguir colocar a maior quantidade possível de cada item, etc.

Assim, o caso geral poderia considerar preencher n posições, cada uma delas com um peso unitário, um valor unitário, um mínimo e um máximo permitido, um peso total máximo. Queremos maximizar o valor. Exemplo:

item	1	2	3	...	...	n
peso do item (KG)	0.5	2.3	1.2			4.1
valor do item (R$)	3,00	3,50	2,20			3.21
quant. mínima	1	5	2			1
quant. máxima	5	10	3			4

peso máximo da mochila: 30 KG

Enumeração de subconjuntos

Considere o conjunto $A = \{a_1, a_2, \ldots, a_n\}$.

Queremos enumerar, ou listar todos os subconjuntos de A.

Já sabemos que são 2^n elementos, considerando também o conjunto vazio.

Esse problema é equivalente a enumerar todas as subsequências de $1\ 2\ \ldots\ n$.

Exemplo para `n=3`.

```
1
2
3
1 2
1 3
2 3
1 2 3
```

A ordem em que cada elemento aparece na sequência não é importante, mas vamos colocá-los em ordem crescente. Ou seja, o subconjunto 2 3 é o mesmo que 3 2.

Considere a sequência `1 2 ... n`. Vamos abreviá-la por `1..n`.
Uma subsequência de `1..n` é uma sequência `s[1], s[2], ..., s[k]` (vamos abreviá-la por `s[1..k]`), onde:
`1 ≤ s[1] < s[2] < . . . < s[k] ≤ n.`

No exemplo acima, usamos o algoritmo de listar as sequências de 1 elemento, de 2 elementos, ..., de n elementos. Outra maneira de obter as subsequências em ordem crescente é a seguinte:

A partir da sequência `1..n`, apagar alguns elementos de todas as formas possíveis.

Exemplo para `n=3`.

```
1 2 3 ┐
1 2 3 ├─ 2 elementos
1 2 3 ┘
1 2 3 ┐
1 2 3 ├─ 1 elemento
1 2 3 ┘
1 2 3 ──  0 elementos
```

A ordem lexicográfica

Outra ordem possível é chamada de ordem lexicográfica. É a ordem que os elementos aparecem quando os listamos na ordem alfabética. Como exemplo suponha que a sequência fosse `a b c`. A ordem alfabética de todas as sequências possíveis seria:

```
a
a b
a b c
a c
b
b c
c
```

No caso de `1 2 3`

```
1
1 2
1 2 3
1 3
2
2 3
3
```

Também é a ordem que os elementos apareceriam se fossem itens de um texto:

```
1
     1.2
          1.2.3
     1.3
2
     2.3
3
```

Observe que os elementos da sequência estão em ordem crescente.

Uma subsequência `r[1..j]` é *lexicograficamente menor* que `s[1..k]` se

1. Existe `i` tal que `r[1..i-1] = s[1..i-1]` e `r[i] < s[i]` ou
2. `j < k` e `r[1..j] = s[1..j]`.

Vamos então fazer um algoritmo que dado `n`, imprima todas as subsequências de `1..n` na ordem lexicográfica.

Em primeiro lugar, vamos fazer uma função que dada uma sequência `s[1..k]` gere a próxima sequência na ordem lexicográfica, devolvendo o seu tamanho que será `k-1` ou `k+1`.

Note que:
Se `s[k]<n`, a próxima será de tamanho `k+1` acrescentando-se a esta `s[k+1] = s[k]+1`;
Se `s[k]=n`, a próxima será de tamanho `k-1` fazendo `s[k-1] = s[k-1]+1`;

Nos algoritmos abaixo vamos usar a lista `s` a partir do índice `1`. Ou seja, vamos ignorar `s[0]`.

A função abaixo gera a próxima sequência na ordem lexicográfica.

```
# Gera a próxima sequência de 1..n na ordem lexicográfica
# àquela que está em s. k é o tamanho da sequência em s.
```

```
# Devolve o tamanho da sequência gerada (k-1 ou k+1)
def Proxima_Lex(s, k, n):
    # Caso particular - o primeiro elemento
    if k == 0:
        s[1] = 1
        return 1
    # Caso particular - último elemento
    if s[1] == n: return 0
    # Caso geral
    if s[k] < n:
        s[k+1] = s[k] + 1
        return k + 1
    s[k-1] += 1
    return k - 1
```

A função abaixo imprime todas as subsequências na ordem lexicográfica usando a função acima.

```
# Imprime todas as sub-sequências de 1..n na ordem lexicográfica
def Imprime_Lex(n):
    # inicia s com n + 1 elementos
    # vamos usar s[1..n] - s[0] não é usado
    s = (n + 1) * [0]
    # Gera a próxima e imprime
    k = 0    # primeira sequência: s[1] = 1
    cont = 1 # contador de sequências
    while True:
        k = Proxima_Lex(s, k, n)
        # Verifica se não há mais
        if k == 0: break
        # Imprime a sequência
        print("\n%05d - " %cont, end = '')
        for i in range(1, k + 1): print("%3d" %s[i], end = '')
        cont += 1
```

A função abaixo devolve uma lista com todos os subconjuntos (listas) ou todas as subsequências na ordem lexicográfica usando a função acima.

```
def SubConjuntos(n):
    # inicia s com n + 1 elementos
    # vamos usar s[1..n]. O elemento s[0] não é usado
    s = (n + 1) * [0]
    sub = []
    # Gera a próxima
    k = 0    # primeira sequência: s[1] = 1
    # repete até terminarem as sequências
    while True:
```

```
        k = Proxima_Lex(s, k, n)
        # verifica se não há mais sequências para gerar
        if k == 0: break
        # adiciona a sequência gerada à lista sub
        sub.append(s[1:k+1])
    return sub
```

Outras ordens de enumeração de subconjuntos

1) Subconjuntos de `1..n` gerados a partir de subconjuntos de `1..(n-1)`

A ideia é tomar todos os subconjuntos de `1..k` e introduzir o elemento `k+1`.
Exemplo para n=4.

Com 1 elemento:

1

Introduzir o 2:
2
12

Introduzir o 3:
3
13
23
123

Introduzir o 4:
4
14
24
124
34
134
234
1234

Note que a cada ao introduzirmos o elemento `k`, acrescentamos mais 2^{k-1} elementos.
Assim a quantidade total para `n` é exatamente:
$1+2+4+8+...+2^{n-1} = 2^n-1$

A função abaixo gera os elementos nessa ordem:

```
# Gera como strings todos os subconjunto de 1..n
```

```
def Gera_Subconjuntos(n):
    tab = []
    for k in range(1, n + 1):
        # acrescenta k em todos os elementos de tab
        # colocando os novos elementos numa outra lista
        tabx = []
        for x in tab:
            tabx.append(x + str(k))
        # concatena as duas listas e inclui k
        tab = tab + tabx + [str(k)]
    # retorna a list com todos os subconjuntos
    return tab
```

---

2) Como um número

Um subconjunto é uma sequência de dígitos 1 2 3 ... n.
Podemos entender como um número entre `1` e `123...n`.
Todos sem repetição e em ordem crescente dos dígitos.
Portanto, usando o método da força bruta: gerar todos os números neste intervalo e testar cada um deles, verificando se atendem a condição acima.
Se `n` for pequeno, até 9, dá para gerar como inteiros e separar os dígitos.

```
for i in range(1, 123456789 + 1):
   # Separar dígitos de i e testar se são todos diferentes
   # e estão em ordem crescente
```

Exercícios

1) Baseado na sugestão 2

Permutações – ordem lexicográfica

Considere a sequência `1..n`.
O problema agora é gerar todas as permutações dos elementos desta sequência.
Também existem algumas ordens que podemos seguir. A quantidade é `n!`.
Vamos considerar a lexicográfica.

Exemplo: `1..4`

```
1  2  3  4
1  2  4  3
1  3  2  4
1  3  4  2
1  4  2  3
1  4  3  2
2  1  3  4
2  1  4  3
2  3  1  4
2  3  4  1
2  4  1  3
2  4  3  1
3  1  2  4
3  1  4  2
3  2  1  4
3  2  4  1
3  4  1  2
3  4  2  1
4  1  2  3
4  1  3  2
4  2  1  3
4  2  3  1
4  3  1  2
4  3  2  1
```

O algoritmo a seguir usa o princípio back-tracking. Vai montando uma permutação mas mantém o caminho de volta para gerar as demais.

```
# Para facilitar usamos uma lista com n+1 posições.
# Não usamos a posição zero da lista.
# p = lista contendo a permutação
# k = posição a ser preenchida (1, 2, 3, ..., n)
def perm(p, k, n):
    if k > n:
        # permutação completa - imprime e retorna
        print(p[1:])
        return
    # escolha um candidato para a casa k
    for j in range(1, n + 1):
        if j not in p[1 : k]:
            # j é candidato
            # chama perm novamente com a posição k preenchida
            p[k] = j
            perm(p, k + 1, n)
    # não tem mais candidatos – return implícito
```

```
# A chamada inicial solicita o preenchimento da posição 1
n = 3 # exemplo
q = (n + 1) * [0] # inicia a lista que conterá a permutação
perm(q, 1, n)
```

Saída:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

Permutações – outra versão desse mesmo algoritmo

Veja agora outra versão desse mesmo algoritmo. Vamos permutar quaisquer objetos em vez de números.

```
# Obj = Lista com os objetos a serem permutados
#       Permutações geradas na ordem crescente dos elementos de Obj
# Nsol = Número da solução
# Obj e Nsol são globais

# Define o elemento em lista[livre]
# Chama recursivamente Perm para o elemento livre + 1
# Chamada inicial: Perm(lista, n, 0)
def Perm(lista, n, livre):
    # lista[0..n-1] = lista em construção
    # n = número de elementos da permutação
    # livre = elemento a ser preenchido
    #
    # chamada inicial: Perm(lista, n, 0)
    #                  lista = n * [None]
    global Nsol, Obj
    # Verifica se lista está totalmente preenchida
    if livre == n:
        # Mostra essa permutação
        Nsol += 1
        print("solução", Nsol, " : ", lista)
        return
    # vamos achar os candidatos para esta posição
    cand = sorted(list(set(Obj) - set(lista) - {None}))
    for x in cand:
```

```
            lista[livre] = x
            # chama recursivamente a Perm
            Perm(lista, n, livre + 1)
        # Terminaram os candidatos
        # Retorna para preencher elementos anteriores a livre
        lista[livre] = None
        return

# teste
Nsol = 0
Obj = ['joão', 'maria', 'alfredo', 'antonio']
ta = [None] * len(Obj)
Perm(ta, len(Obj), 0)
```

Saída:

```
solução 1  :  ['alfredo', 'antonio', 'joão', 'maria']
solução 2  :  ['alfredo', 'antonio', 'maria', 'joão']
solução 3  :  ['alfredo', 'joão', 'antonio', 'maria']
solução 4  :  ['alfredo', 'joão', 'maria', 'antonio']
solução 5  :  ['alfredo', 'maria', 'antonio', 'joão']
solução 6  :  ['alfredo', 'maria', 'joão', 'antonio']
solução 7  :  ['antonio', 'alfredo', 'joão', 'maria']
solução 8  :  ['antonio', 'alfredo', 'maria', 'joão']
solução 9  :  ['antonio', 'joão', 'alfredo', 'maria']
solução 10  :  ['antonio', 'joão', 'maria', 'alfredo']
solução 11  :  ['antonio', 'maria', 'alfredo', 'joão']
solução 12  :  ['antonio', 'maria', 'joão', 'alfredo']
solução 13  :  ['joão', 'alfredo', 'antonio', 'maria']
solução 14  :  ['joão', 'alfredo', 'maria', 'antonio']
solução 15  :  ['joão', 'antonio', 'alfredo', 'maria']
solução 16  :  ['joão', 'antonio', 'maria', 'alfredo']
solução 17  :  ['joão', 'maria', 'alfredo', 'antonio']
solução 18  :  ['joão', 'maria', 'antonio', 'alfredo']
solução 19  :  ['maria', 'alfredo', 'antonio', 'joão']
solução 20  :  ['maria', 'alfredo', 'joão', 'antonio']
solução 21  :  ['maria', 'antonio', 'alfredo', 'joão']
solução 22  :  ['maria', 'antonio', 'joão', 'alfredo']
solução 23  :  ['maria', 'joão', 'alfredo', 'antonio']
solução 24  :  ['maria', 'joão', 'antonio', 'alfredo']
```

Exercícios:

1) Tente achar outro algoritmo que gere as permutações de `1..n` na ordem lexicográfica ou não, recursivo ou não.

2) Existe uma solução imediata a partir do primeiro problema acima. Basta gerar todos os números na base n com n dígitos e verificar quais são permutações. Considere os dígitos 1 a n e não 0 a n-1. Adapte o algoritmo acima para esta solução. Encontre um algoritmo rápido (linear), que descubra se uma sequência `s[1..N]` é uma permutação de `1..N`.
3) Otimizando a solução anterior, note que para as permutações de `1..5` por exemplo, todas as permutações serão números entre `12345` e `54321`.

Permutações – outra ordem – não lexicográfica

Uma maneira de construir permutações de 1..n, é pensar que temos n posições a preencher cada uma delas com um certo número de possiblidades.

Exemplo – permutações de 1..3

```
A primeira casa pode conter 1, 2 ou 3
  Com 1 na primeira, restam 2 e 3 para as 2 casas restantes
    Com 2 na segunda resta apenas o 3 para a terceira casa: 1 2 3
    Com 3 na segunda resta apenas o 2 para a terceira casa: 1 3 2
  Com 2 na primeira, restam 1 e 3 para as 2 casas restantes
    Com 1 na segunda casa resta apenas o 3 a terceira casa: 2 1 3
    Com 3 na segunda casa resta apenas o 1 a terceira casa: 2 3 1
  Com 3 na primeira, restam 1 e 2 para as 2 casas restantes
    Com 1 na segunda casa resta apenas o 2 a terceira casa: 3 1 2
    Com 2 na segunda casa resta apenas o 1 a terceira casa: 3 2 1
```

O algoritmo abaixo, recursivo, usa exatamente esse princípio de ir preenchendo casa a casa. A ordem é quase a lexicográfica.

```
# k = índice do elemento a ser gerado - k = 0 inicio
# perm = lista com a permutação atual sendo gerada
# perm[0] não é usado - perm[1..n] conterá a permutação
# possib = lista com as possibilidades para perm[k]
def GeraPermutacao(k, n, perm, possib):
    # verifica se é a primeira chamada
    if k == 0:
        # define possibilidades
        possib = [i for i in range(1, n+1)]
        perm = [0] * (n + 1)
        k = 1
    # Se chegou ao fim, imprime
    if k > n:
        ImprimePerm(n, perm)
        return
    # Gerar todas as possibilidades em perm[k]
    for i in range(len(possib)):
        prim = possib[0]
        # retira esse elemento da lista de possibilidades
```

```
        del possib[0]
        perm[k] = prim
        GeraPermutacao(k + 1, n, perm, possib)
        # insere o elmento novamente na lista de possibilidades
        possib.append(prim)

# Imprime uma permutação
def ImprimePerm(nn, pp):
    print("\npermutação = ", end = '')
    for i in range(1, nn + 1):
        print(pp[i], end = '')
    print()

# Entra com um número n > 0 e gera as permutações de 1..n
while True:
    num_elem = int(input("Entre com o número de elementos da
permutação:"))
    if num_elem <= 0: break
    print("Serão geradas permutações de:", end = '')
    for i in range(1, num_elem + 1):
        print(i, end = '  ')
    print()
    # Construir a lista
    GeraPermutacao(0, num_elem, [], [])
```

Permutações – outra ordem

Considere a sequência `1..n`.
Outra forma de pensar na enumeração das permutações é gerar permutações de `n` elementos a partir das permutações de `n-1` elementos. Cada permutação de `1..n-1`, gera `n` permutações de `1..n`. Basta colocar `n` em todas as `n` posições possíveis.
Exemplo: vamos gerar todas as permutações de `1..3`, começando com a permutação de `1..1`.

```
1
```

Gerar todas as de `1..2`

```
12
21
```

Para cada uma delas gerar todas de `1..3`

```
123
132
312
```

```
213
231
321
```

Para cada uma delas gerar todas de `1..4`

```
1234
1243
1423
4123

1324
1342
1432
4132

3214
3241
3421
4321

2134
2143
2413
4213

2314
2341
2431
4231

3214
3241
3421
4321
```

A função `perm` abaixo, também recursiva, imprime todas as permutações de `1..N` nesta ordem:

```
# k = elemento a ser inserido em permutações de ordem k - 1
# s = lista com a permutação atual sendo gerada
# n = ordem da permutação

def GeraPermutacao(s, k, n):
    # Verifica se está completa
    if k > n:
        ImprimePerm(n, s)
        return
    # inserir k em todas as k posições possíveis
```

```
    for i in range(k, 0, -1):
        # insere k na posição i
        saux = s[:]
        saux.insert(i, k)
        # Gera as permutações
        GeraPermutacao(saux, k + 1, n)

# Imprime uma permutação
def ImprimePerm(nn, pp):
    print("\npermutação = ", end = '')
    for i in range(1, nn + 1):
        print(pp[i], end = '')
    print()

# Entra com um número n > 0 e gera as permutações de 1..n
while True:
    num_elem = int(input("Entre com o número de elementos da permutação:"))
    if num_elem <= 0: break
    print("Serão geradas permutações de:", end = '')
    for i in range(1, num_elem + 1):
        print(i, end = '  ')
    print()
    # Construir a lista s
    # s[0] não é usado
    # a permutação fica em s[1..n]
    s = [0] * (num_elem + 1)
    # s inicia com a permutação de 1 elemento
    s[1] = 1
    GeraPermutacao(s, 2, num_elem)
```

Exercício

Escreva uma função **`int VerificaPermutacao (int s[], int n)`** que devolve **`1`** se **`s[1..n]`** é uma permutação de **`1..n`** e **`0`** caso contrário. Faça isso de três maneiras:

a) Com um algoritmo O(n)

b) Com um algoritmo O(n^2)

c) Com um algoritmo O(n.log n)

Combinações

Considere a sequência `1..n`.
O problema agora é gerar todas as combinações de `m` elementos desta sequência.
A quantidade é `n!/(m!.(n-m)!)`.
Vamos considerar também a ordem lexicográfica.

Exemplo: todas as combinações de 1..5 com 3 elementos.

```
1  2  3
1  2  4
1  2  5
1  3  4
1  3  5
1  4  5
2  3  4
2  3  5
2  4  5
3  4  5
```

Existe uma solução imediata deste problema a partir da solução de enumerar todos os subconjuntos acima.
Basta mostrar só as subsequências com m elementos.

Exercícios:

1) Tente achar um algoritmo que dados n e m, gere todas as combinações de 1..n com m elementos. Uma sugestão é usar o algoritmo que gera os subconjuntos com uma pequena variação. Veja os comentários abaixo para as combinações de 1..5 com 3 elementos:

1	2	3	Soma 1 no último elemento
1	2	4	Soma 1 no último elemento
1	2	5	5 é o maior nesta posição, soma 1 no anterior e este mais 1 no seguinte
1	3	4	Soma 1 no último elemento
1	3	5	5 é o maior nesta posição, soma 1 no anterior e este mais 1 no seguinte
1	4	5	5 é o maior, deveria somar 1 no anterior, mas 4 é o maior para esta posição. Então soma 1 no anterior, mais 1 no seguinte e mais 1 no seguinte
2	3	4	Soma 1 no último elemento
2	3	5	5 é o maior nesta posição, soma 1 no anterior e este mais 1 no seguinte
2	4	5	Como 5 é o maior, deveria somar 1 no anterior, mas 4 é o maior para esta posição. Então soma 1 no anterior, mais 1 no seguinte e mais 1 no seguinte
3	4	5	É o último porque 5 4 e 3 são os últimos em suas posições.

2) Existe também uma solução imediata baseada no primeiro algoritmo acima. Trata-se de gerar todos os números de m dígitos na base n e verificar quais deles são combinações. Neste caso combinações repetidas podem aparecer e é necessário verificar se os algarismos estão em ordem crescente.

Arranjos

Considere a sequência `1..n`.

O problema agora é gerar todos os arranjos de m elementos desta sequência.
A quantidade é `n!/(n-m)!`.

Exemplo: Todos os arranjos de `1..4` com 2 elementos.

```
1  2
2  1
1  3
3  1
1  4
4  1
2  3
3  2
2  4
4  2
3  4
4  3
```

Exercício:

1) Tente achar a solução imediata a partir dos algoritmos anteriores.
2) Tente achar outro algoritmo para gerar todos os arranjos.
3) Existe uma solução a partir dos algoritmos de permutação e combinações combinados. Basta gerar as m! permutações de cada uma das combinações de n elementos m a m.