Algoritmos de Enumeração

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.

Dizemos que estamos enumerando objetos, ou gerando uma lista de objetos com uma determinada característica.

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.

```
item objetos[m];
```
Onde `objetos[i]` é o objeto associado ao algarismo `i`.

A função `imp_seq_n_base_m (int seq[], int n, int m)` abaixo, imprime todas as sequências, ou todos os números com n dígitos na base m.

```
int proxima(int a[], int N, int M) {
     int t = N-1;
     /* soma 1 ao vetor */
     while (t >= 0) {
          a[t] = (a[t] +1) % M;
          if (a[t] == 0) t--;
```

```
          else return 0;
      }
      return -1;
}

void imp_seq_n_base_m (int seq[], int n, int m) {
      int i;
      for (i = 0; i < n; i++) seq[i] = 0;
      do {
        /* imprime sequência atual */
        for (i = 0; i < n; i++) printf("%2d", seq[i]);
        printf("\n");
        /* gera a próxima */
      } while (proxima(seq, n, m) == 0);
}
```

Outra forma de resolver este problema é gerar todos os números de 0 até $n^m - 1$ e escrevê-los na base **m**. Ou seja, colocar cada dígito em um elemento do vetor **seq[0..n-1]**.

```
for(i = 0;i < n^m - 1; i++) {
   /* transforme i para a base m colocando cada um dos n dígitos
      em seq[0..n-1] */
       ...
}
```

Fica como exercício completar a solução desta forma.

Enumeração de subconjuntos

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

Queremos enumerar, ou listar todos os subconjuntos de A.
Já sabemos que são 2^n elementos, considerando também o conjunto vazio.
Em qual ordem vamos listá-los?
Existem várias ordens possíveis. Por exemplo, todos de 1 elemento, todos de 2 elementos,...

Esse problema é equivalente a enumerar todas as subsequências de `1 2 ... n`.

Exemplo para `n=3`.

```
1
2
3
1 2
```

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

A ordem em que os elementos aparecem na sequência não é importante, mas vamos colocá-los em ordem crescente.

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.`

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
ab
abc
ac
b
bc
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 também 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`;

Novamente vamos usar o vetor `s` a partir do índice `1`.

```
int prox(int s[],int k, int n) {
     /* caso particular - o primeiro elemento */
     if (k == 0) {
          s[1] = 1;
          return 1;
     }
     if (s[1] == n) return 0; /* final */
     if (s[k] < n) {
          s[k+1] = s[k] + 1;
          return k + 1;
     }
     s[k-1]++;
```

```
     return k - 1;
}
```

O programa abaixo imprime todas as subsequências na ordem lexicográfica usando a função `prox`.

```
#include <stdio.h>
#include <stdlib.h>

/* Dada a sequência s[1..k] gera a próxima em ordem lexicográfica.
   Retorna o comprimento da sequência (k-1 ou k+1).
   Retorna 0 se chegou ao fim. */
int prox(int s[],int k, int n) {
     /* caso particular - o primeiro elemento */
     if (k == 0) {
          s[1] = 1;
          return 1;
     }
     /* caso particular - o último elemento */
     if (s[1] == n) return 0;
     if (s[k] < n) {
          s[k+1] = s[k] + 1;
          return k + 1;
     }
     s[k-1]++;
     return k - 1;
}

/* imprime a sequência s[1..k] */
void imprima(int s[], int k) {
     int i;
     printf("\n");
     for (i = 1; i <= k; i++) printf("%4d", s[i]);
}

/* imprime todas as subsequências de 1..n */
int main() {
     int * s;
     int n, k;
     int cc=1;
     printf("\nentre com n:");
     scanf("%d",&n);
     s = malloc((n+1)*sizeof(int));
     k=0;
     while (1) {
          k = prox(s, k, n);
          if (k == 0) break;
```

```
            imprima(s,k);
            cc++;
        }
        printf("\n\n***%5d elementos\n\n", cc);
}
```

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:
$1+2+4+8+...+2^{n-1} = 2^n-1$

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 = 1;i <= 123456789; i++) {
    /* Separar dígitos de i e testar se são todos diferentes
       e estão em ordem crescente */
}
```

Exercício: Baseado nas sugestões anteriores ou em outras que você achar melhor, encontre outro algoritmo para gerar as várias subsequências. Na ordem lexicográfica ou não.

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
```

A função `Permuta` abaixo é recursiva e gera as permutações a partir da primeira `1 2 ...n`, na ordem lexicográfica.

```
/* permutações de 1 a N na ordem lexicográfica */
#include <stdio.h>
#include <stdlib.h>

/* Troca */
void Troca(int v[],int i,int j)
{
    int t;
    t = v[i];
    v[i] = v[j];
    v[j] = t;
}

/* Gira_Esquerda */
void Gira_Esquerda(int v[],int go,int n)
{
    int tmp = v[go];
    for (int i=go; i<n; i++)
    {
        v[i] = v[i+1];
    }
    v[n] = tmp;
}

void Imprima(int s[], int k) {
     int i;
     printf("\n");
     for (i=1; i<=k; i++) printf("%4d", s[i]);
}
```

```
/* função Permuta */
void Permuta(int v[],int inicio, int n) {
    Imprima(v,n);
    if (inicio<n) {
        int i,j;
        for(i=n-1; i>=inicio; i--) {
            for(j=i+1; j<=n; j++) {
                Troca(v,i,j);
                Permuta(v,i+1,n);
            }
            Gira_Esquerda(v,i,n);
        }
    }
}

int main() {
     int * s;
     int N, i;
     printf("\nentre com n:");
     scanf("%d",&N);
     s = malloc((N+1)*sizeof(int));
     /* inicia o vetor */
     for (i=1; i<=N; i++) s[i] = i;
     Permuta (s, 1, N);
}
```

Exercícios:

1) Verifique o funcionamento da função `Permuta` acima.
2) Tente achar outro algoritmo que gere as permutações de `1..n` na ordem lexicográfica ou não, recursivo ou não.
3) 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. 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`.
4) 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

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
321

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:

```
#include <stdio.h>

void imprima(int X[], int NN) {
     int ii;
     printf("\n");
     for (ii=1; ii<=NN; ii++) printf("%3d", X[ii]);
}

void perm(int S[], int K, int N) {
     int i, j;
     int saux[100];
     if (K>N) imprima(S,N);
     else /* coloque K em todas as K posições possíveis e chama perm com K+1 */
          for (i=K; i>=1; i--) {
             for (j=K-1; j>=i; j--) saux[j+1]=S[j];
             saux[i]=K;
             for (j=i-1; j>=1; j--) saux[j]=S[j];
             perm(saux, K+1, N);
          }
}

int main() {
    int s[100];
    int n;
    printf("\nEntre com N:");
    scanf("%d",&n);
    do {
       s[1]=1;
       perm(s, 2, n);
       printf("\nEntre com N:");
       scanf("%d",&n);
    } while (n>0);
}
```

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.