

## AED1 - Aula 26

### Algoritmos de enumeração

Often it appears that there is no better way to solve a problem than to try all possible solutions. This approach, called exhaustive search, is almost always slow, but sometimes it is better than nothing.  
— Ian Parberry, Problems on Algorithms

Hoje vamos falar de alguns problemas de enumeração.

Para resolver certos problemas combinatórios,

- é necessário enumerar (i.e., fazer uma lista com)
  - todos os objetos de um determinado tipo.

O número de objetos a enumerar é tipicamente muito grande.

- Por isso, os algoritmos enumerativos costumam consumir muito tempo.
  - Mas, certas vezes é o melhor que podemos fazer, ou
    - ao menos é um ponto de partida.

#### Enumeração de subconjuntos por tamanho

Talvez, o mais comum desses problemas seja

- apresentar todos os subconjuntos de um conjunto  $S$  dado.

Como exemplo, dado  $S = \{1, 2, 3\}$ , seus subconjuntos são:

$\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$

Qual o número de subconjuntos de um conjunto com  $n$  elementos?

- Resp.:  $2^n$ , pois cada elemento pode ou não estar num subconjunto,
  - sendo assim responsável por dobrar o número de subconjuntos.

Nosso algoritmo vai gerar todos os subconjuntos do conjunto  $S$ ,

- indo dos menores até os maiores. Note que,
  - o 1º será o conjunto vazio e o último será o próprio conjunto  $S$ .

Os subconjuntos terão seus elementos exibidos em ordem crescente.

- Isso é importante para não gerar subconjuntos repetidos,

- como  $\{1, 2\}$  e  $\{2, 1\}$ .

Os subconjuntos de mesmo tamanho serão exibidos em ordem lexicográfica,

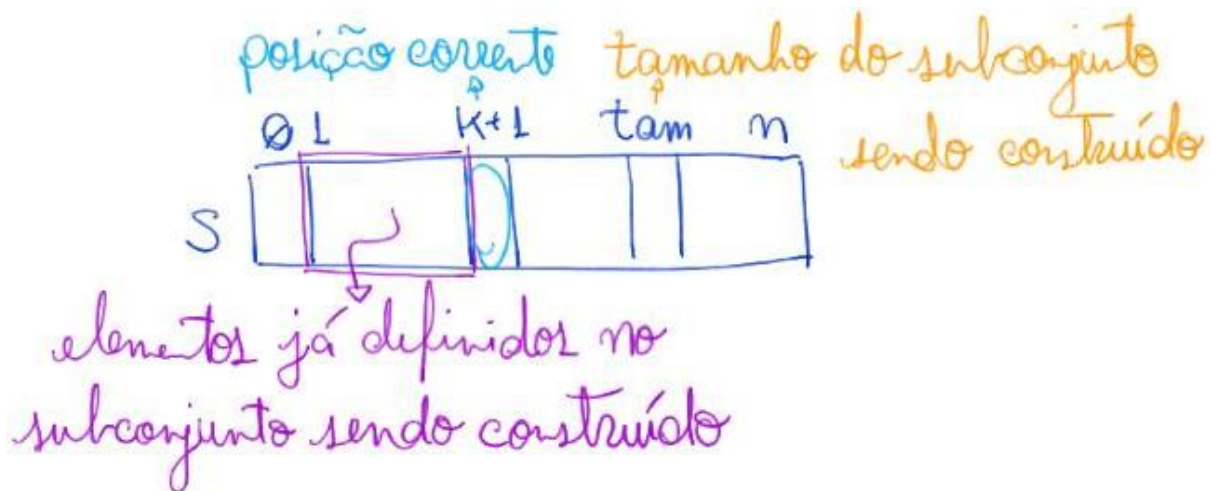
- i.e., ordem alfabética utilizada em dicionários.
- Isso significa que os subconjuntos serão ordenados
  - considerando cada elemento da esquerda para a direita,
- e um subconjunto  $S$  aparecerá antes de outro subconjunto  $S'$ ,
  - se o primeiro elemento distinto entre eles for menor em  $S$  que em  $S'$ .

A ideia do algoritmo é usar uma função recursiva,

- que gera todos os subconjuntos de um determinado tamanho  $tam$ .
- Esta função recursiva será chamada por um laço externo,
  - que varia  $tam$  entre 0 e  $n$ .

Os conjuntos sendo construídos pela função recursiva

- são armazenados em um vetor  $s$ ,
  - que começa vazio e tem tamanho  $n + 1$ .



Cada chamada da função recursiva,

- considera que o subvetor  $s[1 .. k]$  tem a parte já definida
  - do subconjunto sendo construído.
- Assim, ela coloca cada elemento válido na posição corrente  $s[k + 1]$  e
  - faz uma chamada recursiva para preencher o restante do subconjunto.

Um elemento é válido para a posição  $s[k + 1]$

- se ele é maior que o elemento em  $s[k]$ ,
  - já que geramos os subconjuntos em ordem crescente,
- e se existem suficientes elementos maiores que ele
  - para fazer o subconjunto atingir tamanho  $tam$ .

Além disso, os elementos válidos são testados do menor para o maior,

- para gerar os subconjuntos em ordem lexicográfica.

Código do algoritmo recursivo para gerar subconjuntos

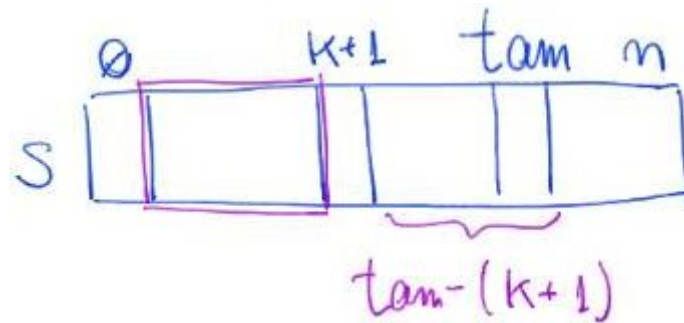
```
void subConj(int n) {
    int *s, tam;
    s = malloc((n + 1) * sizeof(int));
    s[0] = 0;
    for (tam = 0; tam <= n; tam++)
        subConjR(s, 0, tam, n); // chamada que gera todos os
// subconjuntos de {1, ..., n} de tamanho tam
    free(s);
}

// função que gera todos os subconjuntos de {1, ..., n}
// de tamanho tam que contém os elementos em s[1 .. k]
void subConjR(int *s, int k, int tam, int n) {
    int i;
    if (tam == k) {
        imprima(s, tam); // imprime s[1 .. tam]
        return;
    }
    // laço que testa todos os elementos válidos,
    // em ordem crescente, para a posição s[k + 1]
    for (i = s[k] + 1; i <= n - (tam - (k + 1)); i++)
    { // note que (n - i) >= tam - (k + 1)
        s[k + 1] = i;
        subConjR(s, k + 1, tam, n);
    }
}
```

Se todo valor entre 1 e n é um elemento válido

- para colocar no subconjunto sendo construído,
  - por que no laço temos  $i \leq n - (tam - (k + 1))$ ?
- Note que, isso significa  $(n - i) \geq tam - (k + 1)$ ,
  - ou seja, o número de elementos disponíveis é
    - $\geq$  que o número de posições por preencher.
- Observe que, como construímos nossos subconjuntos

- exibindo os elementos em ordem crescente,
- depois de colocar um elemento em  $s[k + 1]$  qualquer elemento  $i$ 
  - disponível para completar o subconjunto
- estará restrito entre  $s[k + 1] < i \leq n$ ,
  - ou seja, temos  $n - s[k + 1]$  opções para  $i$ .
- Por outro lado, como o subconjunto deve atingir tamanho  $\text{tam}$ 
  - e  $s$  já terá  $k + 1$  elementos,
- precisaremos de pelo menos  $\text{tam} - (k + 1)$  elementos para completá-lo.



- Assim, se um elemento  $i > n - (\text{tam} - (k + 1))$ 
  - for colocado na posição  $s[k + 1]$ ,
- sobram  $n - s[k + 1] = n - i < n - (n - (\text{tam} - (k + 1))) = \text{tam} - (k + 1)$  elementos,
  - que é menos do que precisamos para chegar ao tamanho  $\text{tam}$ .

Quiz: Qual a importância de  $s[0]$  começar igual a 0?

- Dica: Note que, em  $\text{subConjR}$ , quando  $k = 0$  temos  $i = s[0] + 1$ .

Eficiência de tempo: Pelo menos da ordem de  $2^n$ ,

- uma vez que o algoritmo gera todos os subconjuntos.

## Enumeração de subsequências em ordem lexicográfica

Uma subsequência é o que sobra de uma sequência

- quando alguns de seus termos são apagados.

Mais precisamente, uma subsequência de  $s_1, s_2, \dots, s_n$  é

- qualquer sequência  $s_{i1}, s_{i2}, \dots, s_{ik}$ ,
  - com  $1 \leq i1 < i2 < \dots < ik \leq n$
- Note que, a ordem dos termos não é alterada.

Como exemplo, dada a sequência 1, 2, 3, 4, 5, 6, 7, 8, temos as subsequências

- 2, 3, 5, 8
- 1, 4, 5, 7, 8
- 2, 3, 5

Nosso problema é:

- Dado  $n$ , enumerar todas as subsequências de  $1, 2, \dots, n$ , ou seja,
  - fazer uma lista em que cada subsequência aparece uma única vez.

Exemplos:

- Para  $n = 3$

1  
1 2  
1 2 3  
1 3  
2  
2 3  
3

- Para  $n = 4$

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

Note que, entre as subsequências de  $1, 2, \dots, n$

- e os subconjuntos de  $\{1, 2, \dots, n\}$ ,
  - há uma correspondência 1 para 1,
    - exceto pelo fato de não listarmos a sequência vazia.
- Por isso, o número de subsequências de  $1, 2, \dots, n$ 
  - é  $2^n - 1$ .

Um adendo para o nosso problema,

- queremos que as subsequências sejam listadas em ordem lexicográfica,
  - que é a ordem do dicionário.

Mais precisamente, uma sequência  $r_1, r_2, \dots, r_j$  é

- lexicograficamente menor que  $s_1, s_2, \dots, s_k$  se
  - $j < k$  e  $r_1, \dots, r_j$  igual a  $s_1, \dots, s_j$  ou
  - existe  $i$  tal que  $r_1, \dots, r_{i-1}$  igual a  $s_1, \dots, s_{i-1}$  e  $r_i < s_i$ .
- Ou seja,  $r_1, \dots, r_j$  é um prefixo de  $s_1, \dots, s_k$  ou
  - as subsequências tem um prefixo comum
    - e o primeiro valor distinto é menor em  $r_1, \dots, r_j$ .

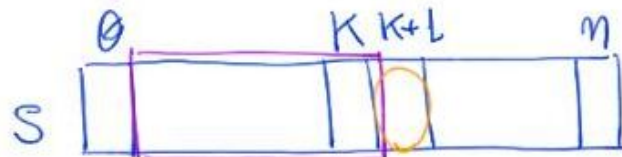
Destacamos que, em geral, a ordem das subsequências não é importante,

- mas pode nos ajudar a organizar nosso algoritmo.

Primeiro veremos um algoritmo iterativo para o problema,

- que constrói as subsequências em um vetor  $s$  e
  - cujas ideias podem ser ilustradas nas seguintes figuras

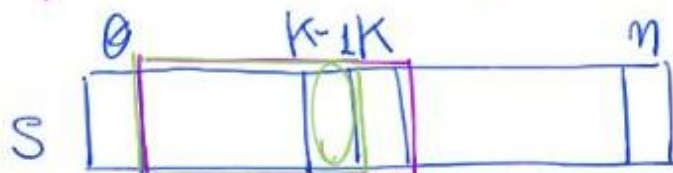
*se subseq  $s[1..k]$  pode ser estendida*



*coloca próximo valor permitido em  $s[k+1]$*

- Note que, uma sequência pode ser estendida
  - enquanto seu último valor não for  $n$ .
- Além disso, o próximo valor permitido para a posição  $k + 1$ ,
  - para gerar as sequências em ordem lexicográfica, é  $s[k] + 1$ .

*se subseq  $s[1..k]$  não pode ser estendida*



*reduz subseq. p/  $s[1..k-1]$  e avança  $s[k-1]$  para o próximo valor permitido*

- De modo semelhante, próximo valor permitido em  $s[k - 1]$ ,
  - que respeita a ordem lexicográfica, é  $s[k - 1] + 1$ .

Código do algoritmo iterativo para gerar subsequências em ordem lexicográficas

```
void subSeqLex(int n) {
    int *s, k;
    s = malloc((n + 1) * sizeof(int));
    s[0] = 0;
    k = 0;
    while (1) {
        if (s[k] < n)
        { // subseq pode ser extendida
            s[k + 1] = s[k] + 1;
            k += 1;
        }
        else // s[k] == n
        { // reduz subseq e avança valor do anterior
            s[k - 1] += 1;
            k -= 1;
        }
        if (k == 0)
            break;
        imprima(s, k);
    }
    free(s);
}
```

Eficiência de tempo: Da ordem de  $2^n$ , i.e.,  $O(2^n)$ ,

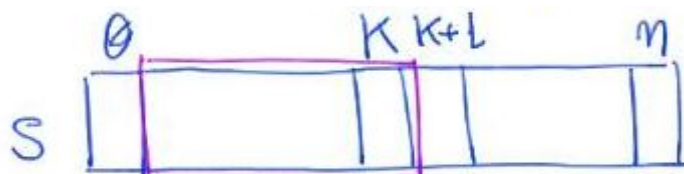
- uma vez que o algoritmo imprime uma subsequência por iteração.

Agora veremos um algoritmo recursivo para o problema,

- cujas ideias podem ser ilustradas nas seguintes figuras.

A função recursiva visa gerar todas as subsequências

- com prefixo  $s[1 .. k]$  em ordem lexicográfica.



- Para cada elemento  $m$  válido para a posição  $s[k + 1]$ ,

- sendo válido  $m$  entre  $s[k] < m \leq n$ ,
  - coloque  $m$  em  $s[k + 1]$  e imprima esta subsequência.

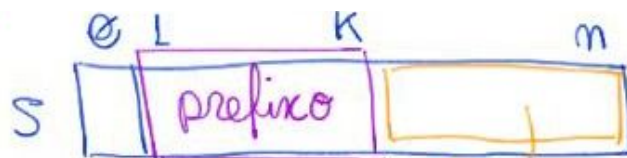


- Então, gere recursivamente todas as subsequências com prefixo  $s[1 .. k + 1]$ ,
  - com  $s[k + 1] = m$ .



sufixos da subsequência,  
com valores entre  $m+1$  e  $n$ ,  
gerados recursivamente

- E também, gere recursivamente todas as subsequências com prefixo  $s[1 .. k]$ ,
  - sem  $m$  no sufixo.



sufixos da subsequência,  
com valores entre  $m+1$  e  $n$ ,  
gerados recursivamente

Código do algoritmo recursivo para gerar subsequências em ordem lexicográfica

```
void subSeqLex2(int n) {
    int *s;
    s = malloc((n + 1) * sizeof(int));
    subSeqLexR(s, 0, 1, n);
    free(s);
}
```



```

// gera toda subsequência com prefixo s[1 .. k]
// e sufixos com valores em {m, ..., n}
void subSeqLexR(int *s, int k, int m, int n) {
    if (m <= n) {
        s[k + 1] = m;
        imprima(s, k + 1);
        subSeqLexR(s, k + 1, m + 1, n); // inclui m
        subSeqLexR(s, k, m + 1, n);    // não inclui m
    }
}

```

- Note que, a ordem lexicográfica é garantida
  - pela posição da impressão e ordem das chamadas recursivas.

Eficiência de tempo: Da ordem de  $2^n$ , i.e.,  $O(2^n)$ ,

- uma vez que o algoritmo imprime uma subsequência por chamada recursiva.

## Enumeração de permutações

Agora veremos um problema de enumeração um pouco diferente,

- no qual o comprimento dos objetos enumerados não muda,
  - mas a ordem dos seus elementos é alterada.

Dada uma sequência, uma permutação da mesma é qualquer sequência

- em que cada elemento da sequência original
  - apareça uma e apenas uma vez.

Nosso problema é, dado um inteiro positivo  $n$ ,

- gerar todas as permutações da sequência identidade  $1, 2, \dots, n$ .

Como exemplo, para  $n = 3$ , temos

```

1, 2, 3
1, 3, 2
2, 1, 3
2, 3, 1
3, 1, 2
3, 2, 1

```

Note que, o número de permutações de uma sequência de comprimento  $n$ ,

- que não tem elementos repetidos,
  - é  $n * (n - 1) * \dots * 2 * 1 = n!$
- Isso porque, qualquer dos  $n$  elementos pode aparecer na primeira posição,
  - qualquer dos  $n - 1$  restantes pode aparecer na segunda,
    - e assim por diante.
- Exercício: fazer a demonstração formal da intuição anterior,
  - usando indução matemática.

Novamente, vamos gerar nossas permutações em ordem lexicográfica.

- Exercício: analise, no algoritmo, que decisão de projeto gera esse resultado.

A seguir, apresentamos um algoritmo recursivo para esse problema,

- cuja ideia central é construir incrementalmente as permutações.

Assim, dado um prefixo da permutação no subvetor  $s[1 .. k]$ ,

- o algoritmo coloca cada um dos elementos válidos na posição  $s[k + 1]$ ,
  - sendo válidos elementos que não aparecem em  $s[1 .. k]$ ,
- e gera recursivamente as permutações
  - que irão preencher o sufixo  $s[k + 2 .. n]$  do subvetor.

gera toda permutação com prefixo  $s[l..k]$



colocando cada elemento ainda não usado em  $(k+1)$  e preenchendo o restante recursivamente

Código do algoritmo recursivo para gerar permutações

```
void perm(int n) {
    int *s;
    s = malloc((n + 1) * sizeof(int));
    permR(s, 0, n); // gera todas as permutações de 1, 2, ..., n
    free(s);
}

// gera todas as permutações de 1, 2, ..., n com prefixo s[1 .. k]
void permR(int *s, int k, int n) {
```

```

int i;
if (k == n) {
    imprima(s, n);
    return;
}
for (i = 1; i <= n; i++) {
    if (!presente(s, k, i)) {
        s[k + 1] = i;
        permR(s, k + 1, n);
    }
}
}

// verifica se x está presente em v[1 .. n]
int presente(int *v, int n, int x) {
    int i;
    for (i = 1; i <= n; i++)
        if (v[i] == x)
            return 1;
    return 0;
}

```

Eficiência de tempo:

- O algoritmo leva pelo menos tempo da ordem de  $n!$ ,
  - uma vez que o algoritmo gera todas as permutações,
- mas ao considerarmos o tempo que ele gasta
  - para preencher cada posição de uma permutação,
- o tempo total é ainda maior.

Bônus:

- Podemos usar implementações eficientes de Tabelas de Símbolos
  - como hash tables, para substituir a função presente
    - por uma busca mais eficiente.