Estruturas de Dados

aula 03: Trabalhando com listas ordenadas

1 Introdução

Na aula passada, nós vimos vários exemplos onde trabalhar com listas ordenadas é mais rápido do que trabalhar com listas desordenadas.

Na aula de hoje, nós vamos ver outros exemplos onde isso é verdade.

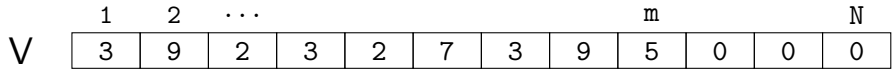
E vamos ver alguns exemplos onde isso não é.

2 Trabalhando com listas ordenadas

Imagine que nós temos uma lista que poder ter elementos repetidos.

E imagine que essa lista tem algumas posições vazias no final — (*preechidas com 0's*)

Por exemplo,

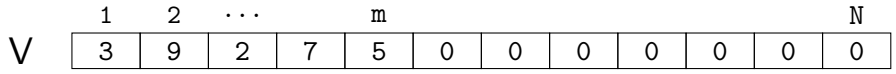


A tarefa consiste em

→ *Remover os elementos duplicados da lista*

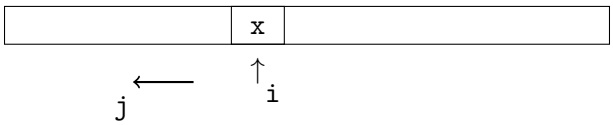
Quer dizer, se um elemento já apareceu antes, ele deve ser removido da lista.

No exemplo acima, isso nos daria



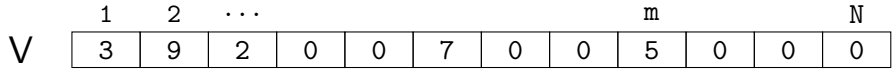
Certo.

Para encontrar as duplicatas, a gente percorre a parte anterior da lista para ver se o elemento já apareceu ali



E daí, se ele for uma duplicata, a gente marca ele com 0.

No exemplo acima, a coisa fica assim

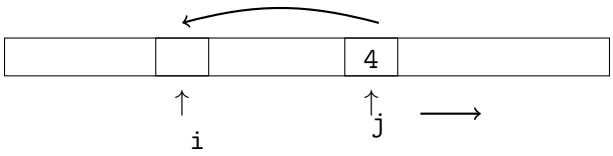


O problema ainda não está resolvido, claro.

E agora é preciso mover os 0's para o fim.

Nós vamos fazer isso utilizando dois dedos

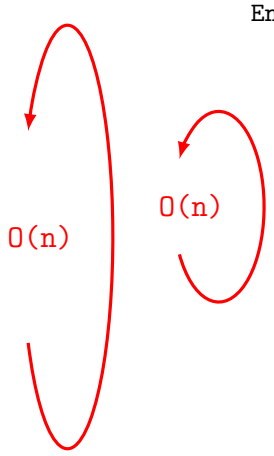
- o dedo `j` vai na frente, procurando elementos diferentes de 0
- e o dedo `i` coloca o elemento na posição correta



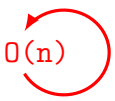
Para a coisa funcionar, o dedo `i` é posicionado no primeiro 0.

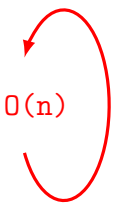
E o dedo `j` começa na posição seguinte.

Abaixo nós temos o pseudo-código completo (já anotado)




```
// 1a Etapa:  marcar as duplicatas
i = 2
Enquanto ( i ≤ m )                    // Para cada elemento
    j = i - 1;    achei  = 0
    Enquanto ( j > 0 )                // Procura lá atrás
        Se ( V[j] == V[i] )           // Você é uma duplicata?
            achei = 1
            Quebra
        j = j + 1
    Se ( achei == 1 )                 // Se for uma duplicata
        V[i] = 0                      // apaga o elemento
    i = i + 1
```






```
// 2a Etapa:  mover os 0's para o fim
i = 2
Enquanto ( V[i] ≠ 0 )                 // encontra o 1o zero
    i = i + 1
j = i + 1
Enquanto ( j ≤ m )                    // percorre o restante da lista
    Se ( V[j] ≠ 0 )
        V[i] = V[j];    i = i + 1
        V[j] = 0
    j = j + 1
m = i - 1
```

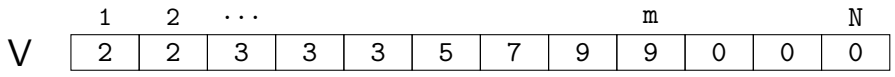
— (*as anotações $O(1)$ foram omitidas para simplificar a visualização*)

Agora, basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot \Big[O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \Big]$$
$$+ \quad O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1)$$
$$= \quad O(n^2)$$

Legal.

Agora, a gente imagina que a nossa lista já está ordenada



E daí, a gente nota que é bem fácil verificar se um elemento é uma duplicata: basta olhar para o elemento anterior.

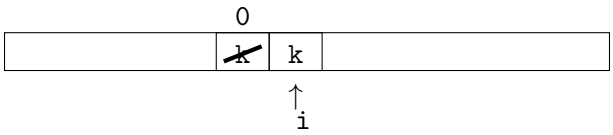
Daí, é só zerar as duplicatas.

E depois mover os zeros para o fim.

Certo.

Mas, aqui nós vamos fazer uma pequena esperteza.

Quer dizer, quando a gente encontrar a duplicata, nós vamos zerar o elemento anterior



Porque daí, a duplicata serve como o elemento anterior para o próximo elemento.

A coisa fica assim

```
                // 1a Etapa:  marcar as duplicatas
O(1)  |         i = 2
                Enquanto ( i ≤ m )
O(n)
O(1)  |         Se ( V[i] == V[i-1] )          // Você é uma duplicata?
                    V[i-1] = 0                 // Apaga elemento anterior

                // 2a Etapa:  mover os 0's para o fim
O(n)  |         (...  a mesma coisa ...  )
```

Agora, basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(n) \quad = \quad O(n)$$

Quer dizer, o algoritmo da versão ordenada é muito mais rápido do que o outro.

A seguir nós vamos ver outros exemplos.

Exemplos

a. Os elementos comuns

Agora imagine que nós temos duas listas

U	5	3	9	7	8	10	13	1	2	14
V	4	8	2	11	6	5	15	3	19	1

A tarefa consiste em

→ *Imprimir todos os números que aparecem nas duas listas*

— *(*i.e., a operação de interseção*)*

Bom, as duas listas estão desordenadas.

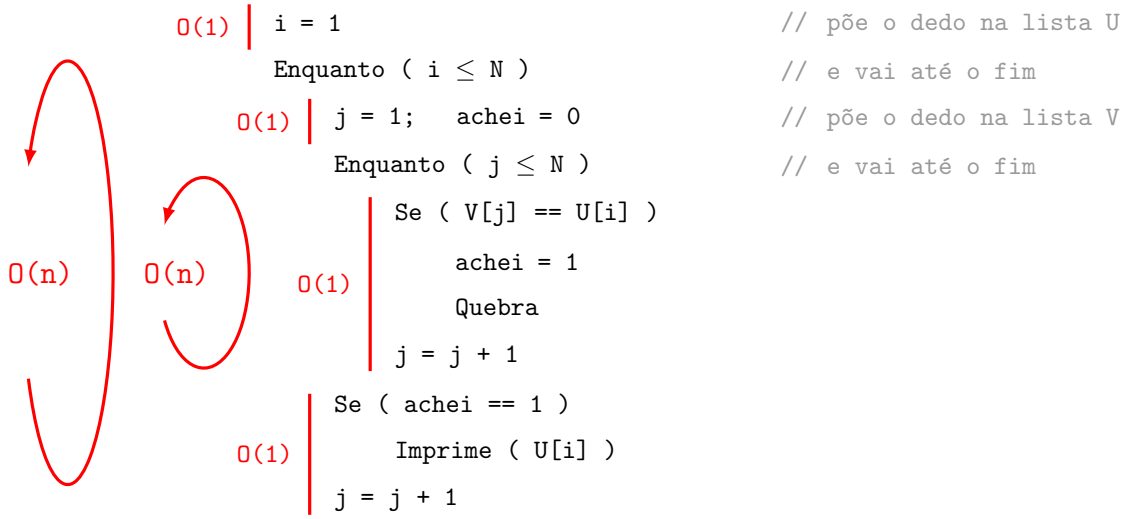
Então, não há outra coisa a fazer senão

Verificar, para cada elemento de U, se ele aparece em V

Se aparecer imprime

Se não aparecer não imprime

A coisa fica assim



Agora, basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot \Big[O(1) \quad + \quad O(n) \cdot O(1) \quad O(1) \Big] \quad = \quad O(n^2)$$

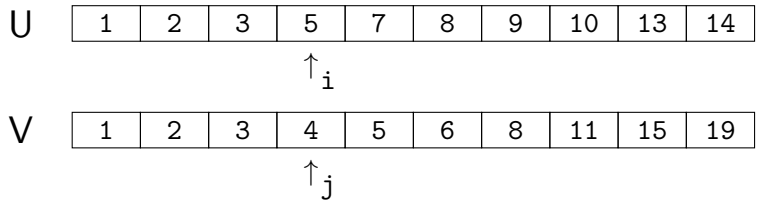
Legal.

Agora imagine que as listas estão ordenadas

U	1	2	3	5	7	8	9	10	13	14
V	1	2	3	4	5	6	8	11	15	19

Bom, a ideia aqui é colocar um dedo em cada lista, e verificar se os elementos são iguais.

Fazendo isso, a gente imprime 1, 2, 3 e chega na seguinte situação



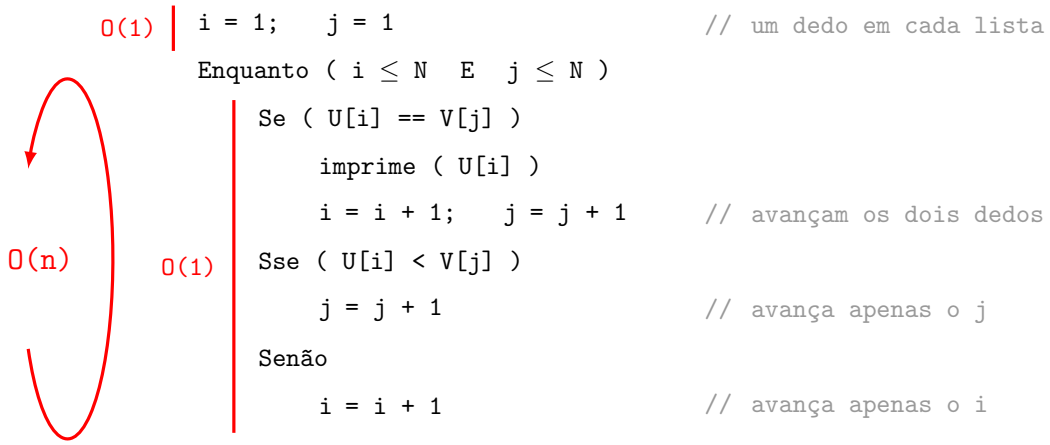
Aqui a gente vê que o 4 é menor que o 5.

E daí, como as listas estão ordenadas, a gente já sabe que o 4 não está na lista U.

Então, a gente avança o dedo `j`.

A coisa continua assim até o fim.

E abaixo nós temos o algoritmo (já anotado)



Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad = \quad O(n)$$

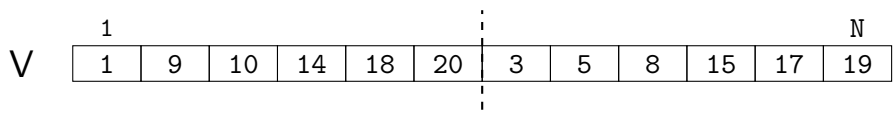
E aqui, mais uma vez, a gente vê que o algoritmo da versão ordenada é muito mais rápido.

◇

b. O procedimento de intercalação

Agora imagine que nós temos uma lista onde a primeira metade e a segunda metade já estão ordenadas.

Por exemplo,



A tarefa é

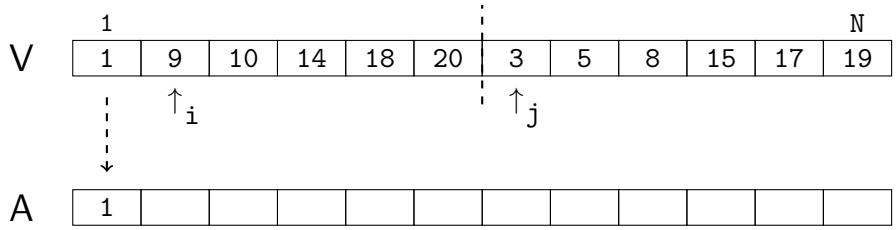
→ *Ordenar a lista completamente*

Bom, aqui a gente pode usar uma ideia semelhante à do exemplo anterior

Quer dizer, a gente coloca um dedo em cada metade.

E verifica qual elemento é o menor.

Daí, o menor deles é copiado para uma lista auxiliar



A coisa continua assim até que todos os elementos tenham sido copiados para a lista A.

E daí, é só copiar os elementos de volta para a lista V.

Abaixo nós temos o algoritmo (já anotado)

```
O(1)  i = 1;    j = N/2 + 1                  // um dedo em cada metade
      k = 1                                  // um dedo na lista auxiliar
      Enquanto ( i ≤ N/2   OU   j ≤ N )
O(n)      Se ( i > N/2 )                     // a 1a metade já acabou
O(1)          A[k] = V[j];    k++;    j++
          Sse ( j > N )                      // a 2a metade já acabou
              A[k] = V[i];    k++;    i++
          Sse ( U[i] < V[j] )                // quem é o menor?
              A[k] = V[i];    k++;    i++
          Senão
              A[k] = V[j];    k++;    j++

O(1)  i = 1                                  //
      Enquanto ( i ≤ N )                     // copia tudo de volta
O(n)      V[i] = A[i];    i++                //
```

Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad + \quad O(n) \cdot O(1) \quad = \quad O(n)$$

◇

c. A operação de inserção

Imagine outra vez que nós temos uma lista com posições vazias no fim

A tarefa é

→ *Inserir o número* `k` *na lista*

Bom, se a lista não está ordenada, isso é a coisa mais fácil do mundo

- basta colocar o `k` na 1a posição vazia, e depois aumentar o valor de m

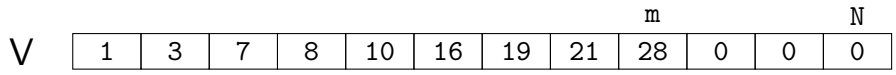
A coisa fica assim

```
V[m+1] = k
m++
```

E o tempo de execução é $O(1)$.

Legal.

Agora imagine que a lista está ordenada



E imagine que `k` = 17.

Então, o `k` precisa ser colocado entre o 16 e o 19.

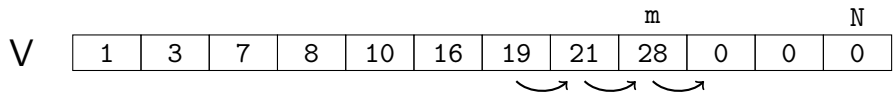
Só que não tem lugar ali.

Quer dizer, as posições vazias estão todas no final da lista.

E agora?

Bom, agora a gente desloca uma posição vazia até ali.

Ou melhor, a gente move todos os elementos a partir do 19 uma posição para a direita



E daí, a gente pode colocar o `k` no lugar em que o 19 estava.

Certo.

Mas antes de tudo, é preciso descobrir o lugar onde o `k` vai entrar.

Abaixo nós temos o código completo (já anotado)

```
// 1a Etapa:  encontrar o lugar do k

O(1)  i = 1                                  // coloca o dedo no inicio
      Enquanto ( i ≤ N )                     // e vai adiante ...
O(n)      Se ( V[i] > k   OU   V[i] = 0 )
O(1)          Quebra
          i++
```

```
                // 2a Etapa:  deslocar os elementos

O(1)  |  j = m                                  // coloca o dedo no fim
      Enquanto ( j ≥ N )                       // e vai em direção ao i
O(n)  |  O(1)  |  V[j+1] = V[j]
               |  j--

                // 3a Etapa:  inserir o k na lista
O(1)  |  V[i] = k
      |  m++
```

Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad = \quad O(n)$$

E aqui nós temos o primeiro exemplo onde a versão desordenada do problema é mais rápida.

◇

d. A operação de remoção

Imagine mais uma vez que nós temos uma lista com posições vazias no final

A tarefa é

→ *Remover o número* `k` *da lista*

Bom, se a lista não está ordenada, a tarefa é fácil

- basta colocar o último elemento na posição do `k`, e depois diminuir o valor de m

Mas antes disso, é preciso encontrar o `k`.
Abaixo nós temos o algoritmo completo (já anotado)

```
                // 1a Etapa:  procurar o k

O(1)  |  i = 1;    achei = 0
      Enquanto ( i ≤ m )                       // percorre a lista
O(n)  |  Se ( V[i] == k )                       // Você é o k?
      |  O(1)  |  achei = 1
               |  Quebra
      |  i++
```

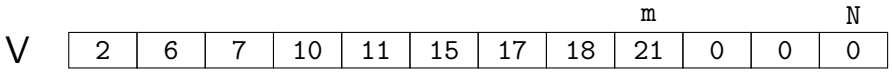
```
// 2a Etapa:  remover o k
Se ( achei == 1 )                    Se o k foi encontrado
    V[i] = V[m]
    m--
```
O(1)

Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad = \quad O(n)$$

Legal.

Agora imagine que a lista está ordenada

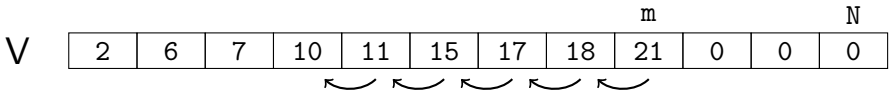


E imagine que `k` = 10.

Bom, dessa vez nós não podemos colocar o último elemento no lugar do `k` — (*porque a lista iria ficar desordenada*)

E nem podemos deixar a posição do `k` vazia — (*porque?*)

Então, a gente move todos os elementos depois do `k` uma posição para a esquerda



A coisa fica assim

```
// 1a Etapa:  procurar o k

(...  a mesma coisa ...)

// 2a Etapa:  deslocar elementos para a esquerda

Se ( achei == 1)
    j = i+1                          // primeiro elemento depois do k
    Enquanto ( j ≥ m )               // vai até o fim
        V[j-1] = V[j]                // deslocando para a esquerda
        j++

// 3a Etapa:  apagar o último elemento
V[m] = 0
m--
```
O(n) — O(1)

O(n) — O(1)

Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(n) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad = \quad O(n)$$

◇

e. A operação de atualização

Imagine mais uma vez que nós temos uma lista com posições vazias no final

A tarefa é

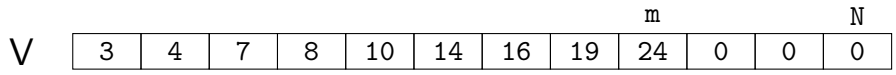
→ *Encontrar o número* `x` *e atualizar o seu valor para* `y`

Dessa vez, a versão desordenada não tem muita graça.

Quer dizer, primeiro a gente faz a busca, e depois atualiza o valor do elemento.

E isso leva tempo $O(n)$ — (*porque?*)

Então, a gente imagina direto que a lista está ordenada



E imagina que `x` = 24.

Daí nós temos dois casos.

Quer dizer,

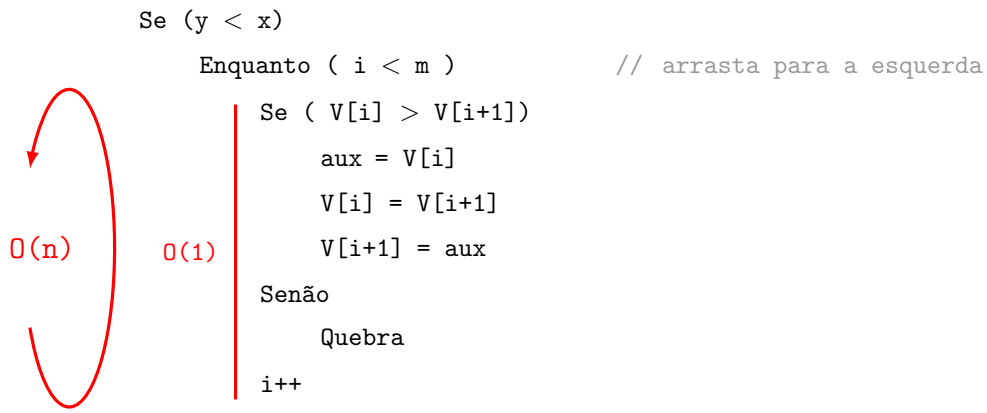
- Se `y` for menor que `x`, então após a atualização pode ser necessário arrastar o elemento para a esquerda
- Se `y` for maior que `x`, então após a atualização pode ser necessário arrastar o elemento para a direita

A coisa fica assim

```
        // 1a Etapa:  procurar o x
O(n)    (...  a mesma coisa ...)

        // 2a Etapa:  deslocar elementos para a esquerda

        Se ( achei == 1)
O(1)        V[i] = y                        // atualiza o elemento
            Se (y < x)
                Enquanto ( i > 1 )          // arrasta para a esquerda
O(n)                Se ( V[i] < V[i-1])
O(1)                    aux = V[i]
                        V[i] = V[i-1]
                        V[i-1] = aux
                    Senão
                        Quebra
                    i--
```



Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(n) \quad + \quad O(1) \quad + \quad O(n) \cdot O(1) \quad = \quad O(n)$$

◇

f. Um algoritmo de ordenação

Nós já vimos vários exemplos onde é uma onde é uma boa ideia trabalhar com a lista ordenada.

Mas, para que a lista esteja ordenada primeiro é preciso ordená-la, não é?

Bom, agora nós vamos ver um algoritmo que coloca a lista em ordem.

Provavelmente, a ideia mais natural para ordenar uma lista é

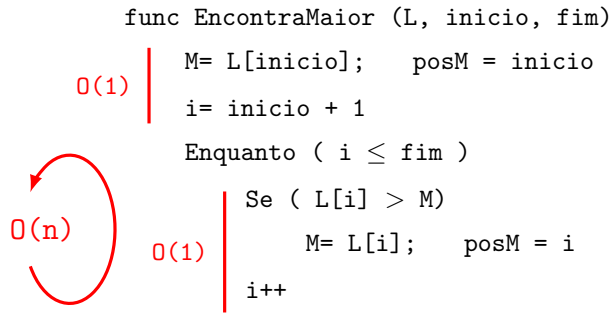
- colocar o maior elemento na última posição
- colocar o 2o maior elemento na penúltima posição
- colocar o 3o maior elemento na antepenúltima posição
- e por aí vai ...

Para fazer isso, é preciso encontrar o maior elemento.

E depois que o maior elemento está na última posição, o segundo maior elemento é o maior da porção $\texttt{V}[1..\texttt{N}-1]$ da lista.

Então aqui, é uma boa ideia escrever uma função que encontra o maior elemento em uma porção da lista — (*e retorna a sua posição*)

A coisa fica assim



E é fácil ver que a função executa em tempo $O(n)$ — (*no pior caso*)

Certo.

Já que a gente está aqui, não custa nada escrever a função que troca dois elementos de lugar

```
func Troca (L, i, j)
    aux = L[i]
    L[i] = L[j]
    L[j] = aux
```

O(1)

que executa em tempo $O(1)$.

E agora, é bem fácil escrever o algoritmo que ordena a lista

```
func OrdenaLista (L)
    Para j = N Até 2
        i = EncontraMaior (L,1,j)
        Troca(L,i,j)
```

O(n) O(n) ⟵ O(1) ⟵

Agora basta examinar as anotações para obter o tempo de execução do algoritmo

$$O(n) \cdot \Big[O(n) \;+\; O(1) \Big] \quad = \quad O(n^2)$$

Legal, não é?

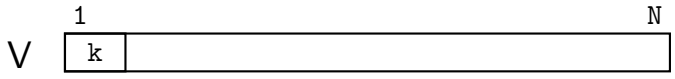
◇

g. Ordenação via partição

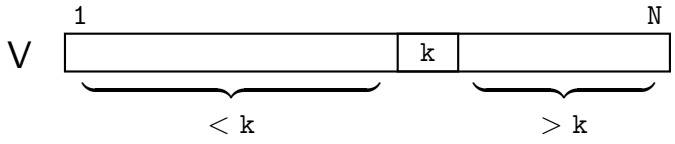
Sim, mas agora nós vamos ver uma coisa mais legal.

Quer dizer, imagine que nós temos uma lista desordenada.

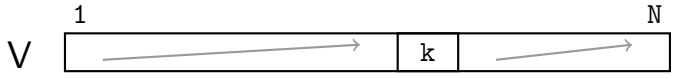
E imagine que o seu primeiro elemento é o `k`



Daí, a gente aplica o procedimento de partição: mover os elemento menores que `k` para a esquerda, e os elemento maiores que para para a direita



E daí, a gente ordena o lado esquerdo e o lado direito — (*usando o algoritmo do exemplo (f)*)



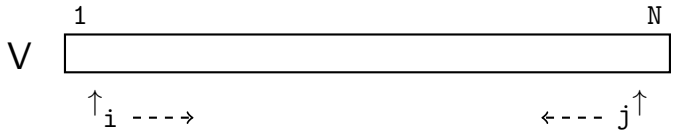
Quando a gente faz isso, a lista fica completamente ordenada.

Mas, será que isso é uma boa ideia?

Bom, para descobrir iss, nós vamos escrever e analisar o algoritmo.

E a gente começa com a função de partição.

A ideia é usar dois dedos que se deslocam simultaneamente



A ideia é que `i` procura elementos maiores do que `k`.

E `j` procura elementos menores do que `k`.

Quando ambos já encontraram o seu elemento, daí ocorre a troca.

E a coisa pára quando os dedos `i`, `j` se cruzam.

A coisa fica assim

```
                func Partição (L)
                    k = L[1]
O(1)                i = 1;    j = N
                    Enquanto (i ≤ j)
                       Se (L[i] > k  E  L[j] < k)
O(n)                       Troca(L,i,j)
O(1)                   Se (L[i] < k)    i++
                       Se (L[j] > k)    j--
O(1) ⟵              Troca(L,1,i-1
```

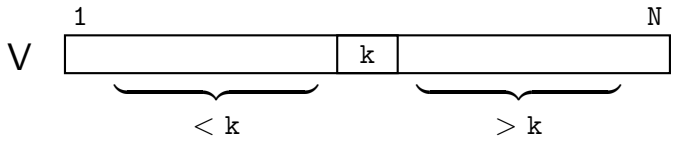
Examinando as anotações, a gente vê que a função executa em tempo

$$O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \quad = \quad O(n)$$

E agora, é preciso estimar o tempo da ordenação.

Mas aqui existem vários casos — (*dependendo do lugar em que o* `k` *foi parar ...*)

O melhor caso é a situação em que o `k` ficou no meio da lista



Daí, cada ordenação trabalha com a metade da lista.

E cada uma delas leva tempo

$$\left(\frac{n}{2}\right)^2 \quad = \quad \frac{n^2}{4}$$

— (*aqui nós não estamos trabalhando com a notação O para ver melhor o que está acontecendo*)

Logo, a coisa toda leva tempo

$$\frac{n^2}{4} \quad + \quad \frac{n^2}{4} \quad + \quad n \quad = \quad \frac{n^2}{2} \quad + \quad n$$

E aqui a gente vê que o tempo de execução foi reduzido (aprox.) à metade — (*quando n é grande*)

Legal, não é?

Sim, mas esse é apenas o melhor caso.

E ainda falta examinar o pior caso.

Bom, o pior caso acontece quando o `k` vai parar em um dos extremos da lista

Nesse caso, nós temos apenas uma ordenação, que trabalha com uma lista com $n - 1$ elementos.

Isso leva tempo

$$(n-1)^2 \quad = \quad n^2 \quad - \quad 2n \quad + \quad 1$$

Logo, a coisa toda leva tempo

$$n^2 \quad - \quad 2n \quad + \quad 1 \quad + \quad n \quad = \quad n^2 \quad - \quad n \quad + \quad 1$$

E dessa vez, a gente não ganhou quase nada.

Agora, a gente pode perguntar: *Essa ideia vale a pena na prática ou não?*

$\diamond$