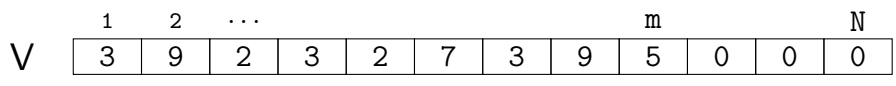
Imagine que nós temos uma lista que poder ter elementos repetidos.

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

Por exemplo,

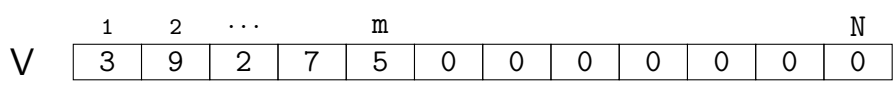



A tarefa consiste em

→ *Remover os elementos duplicados da lista*

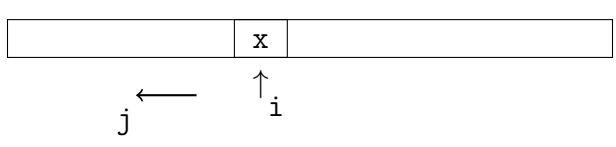
Quer dizer, se um elemento já apareceu antes, ele deve ser removido da lista.

No exemplo acima, isso nos daria



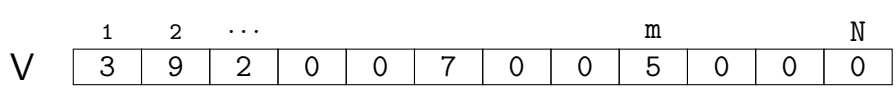

Certo.

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

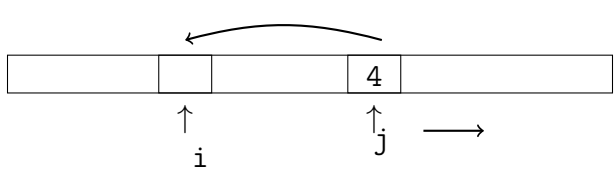



O problema ainda não está resolvido, claro.

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

Nós vamos fazer isso utilizando dois dedos

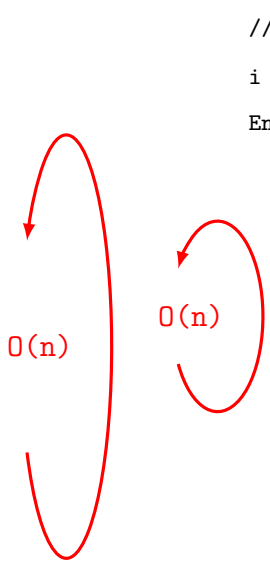
- o dedo `j` vai na frente, procurando elementos diferentes de 0
- e o dedo `i` coloca o elemento na posição correta




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

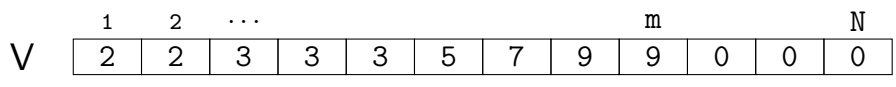
$$O(1) \quad + \quad O(n) \cdot \Big[ O(1) \quad + \quad O(n) \cdot O(1) \quad + \quad O(1) \Big]$$

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

$$= \quad O(n^2)$$

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.

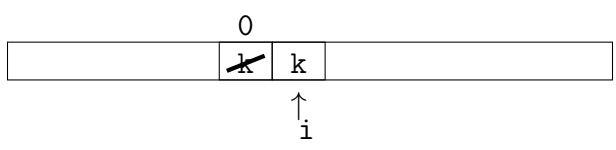
Daí, é só zerar as duplicatas.

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
i = 2                                       O(1)
Enquanto ( i ≤ m )
     Se ( V[i] == V[i-1] )                  // Você é uma duplicata?   O(n) / O(1)
          V[i-1] = 0                        // Apaga elemento anterior

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

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.