Só que, dessa vez é fácil ver como aumentar a eficiência do algoritmo.

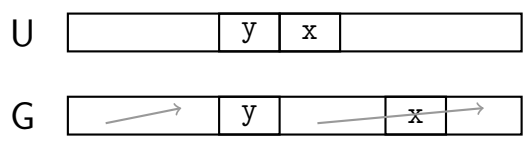
Quer dizer, nós estamos fazendo um monte de ordenações.

E cada ordenação começa o seu trabalho do zero.

Então a ideia é tentar reaproveitar o trabalho da ordenação anterior para realizar a próxima ordenação de maneira mais eficiente.

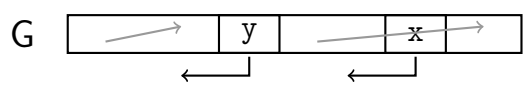
É bem fácil ver como isso funciona para a lista U.

Quer dizer, a ordenação anterior deixa a situação assim



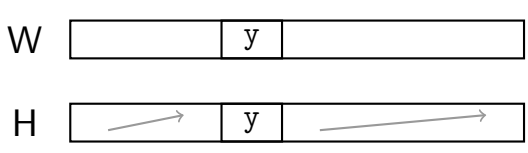
Daí, quando for a vez do x assumir o papel de pivô, basta

- deslocar o y para o lugar correto na ordem do lado esquerdo de G
- localizar o x no lado direito e deslocá-lo até a posição do pivô em G



Dessa vez, nós não vamos escrever o código da função, mas é bem fácil ver que essa computação pode ser realizada em tempo $O(n)$.

Mas, a situação nas listas W e H é um pouco mais complicada



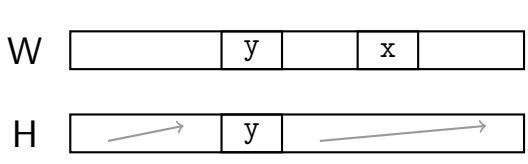
Porque, com a mudança do pivô, vários elementos vão mudar de lado.

E nós precisamos encontrar uma maneira de fazer isso sem ter que reordenar tudo.

A ideia, claro, é aproveitar a ordenação que nós já temos.

Vamos ver como a coisa funciona.

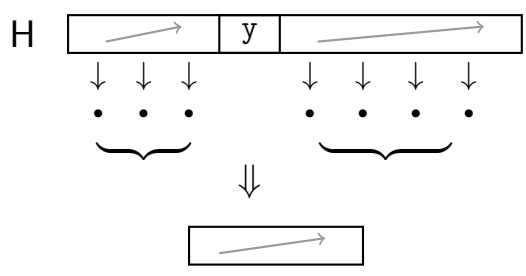
O primeiro passo é visualizar o novo pivô



Os elementos que estão à esquerda de x em W vão formar a nova primeira porção ordenada em H.

Só que nesse momento, os elementos estão espalhados nos dois lados de H.

A ideia, então, é localizar esses elementos e depois fazer a intercalação



— (*porque os dois lados de* H *estão ordenados*)

Mas, como é que a gente localiza esses elementos rapidamente?

Bom, aqui é a hora para mais uma esperteza.

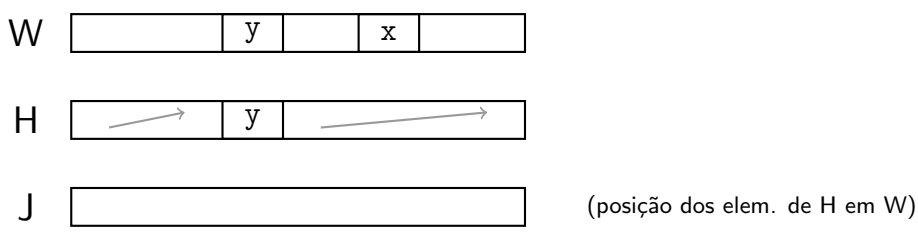
Quer dizer, os elementos que nós queremos localizar são aqueles que estão à esquerda de x na lista W.

Então, basta percorrer a lista H e verificar, para cada elemento, se ele está à esquerda de x em W.

O problema dessa ideia é que nós não sabemos a posição de um elemento de H na lista W.

Mas, quem disse que nós não sabemos?

Quer dizer, se isso é uma informação útil, então nós podemos manter ela guardada em uma lista auxiliar J



Agora, o teste que nós precisamos fazer para cada elemento de H leva tempo $O(1)$,

Logo, os elementos podem ser localizados em tempo $O(n)$.

A intercalação também leva tempo $O(n)$.

E nós fazemos a mesma coisa para os elementos do outro lado de x.

Logo, a reordenação da lista H também leva tempo $O(n)$.

— (*e mais uma vez, nós vamos omitir o código dessa função*)

Abaixo, nós temos a nova versão do nosso algoritmo

```
func contaTE-4.0 (U,W)

    J ← {1,2,3,...,n}                          // primeira iteração
    pivo ← U[2]
O(n)| j ← Localiza (pivo,W[1..N])
O(nlogn)| G ← ordena2Lados (U,2)
        | H,J ← ordena2Lados (W,j,J)
O(n)| C1,C2 ← contaComunsOrd (G,2,H,j)
    cont ← C1*C2

    Para i ← 3 Até N-1                         // demais iterações
O(n)|   pivo ← U[i]
    |   j ← Localiza (pivo,W[1..N])
    |   G ← Reordena-1 (G,i)
    |   H,J ← Reordena-2 (H,j,J)
    |   C1,C2 ← contaComunsOrd (G,2,H,j)
    |   cont ← C1*C2
O(n)

    Retorna (cont)
```

Agora, nós já temos um algoritmo que executa em tempo

$$O(n) \quad + \quad O(n \log n) \quad + \quad O(n) \times O(n) \qquad = \qquad O(n^2)$$

o que já não é mais uma porcaria — (*para esse problema ...*)

Mas, será que a gente consegue fazer ainda mais rápido?