Construção e Análise de Algoritmos

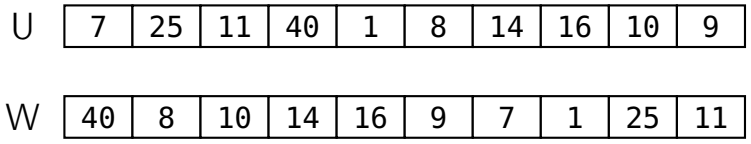
aula 05: Botando a ordenação para funcionar

1 Introdução

Imagine que nós temos duas listas U e W, ambas com tamanho n.

As duas listas contém os mesmos elementos, em ordens diferentes.

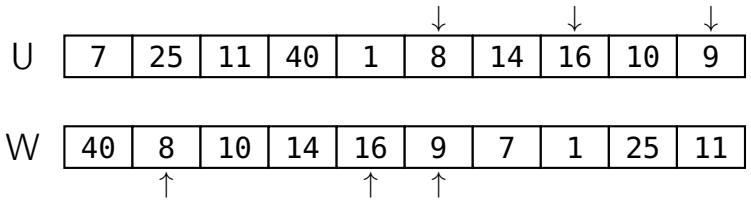
Por exemplo,



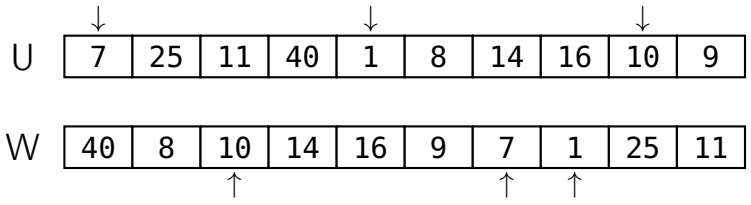
Mas daí, pode acontecer que para uma tripla qualquer (x, y, z), os três números aparecem na mesma ordem nas duas listas.

Quando isso acontece, nós dizemos que (x, y, z) é uma tripla engraçada.

No exemplo acima, $(8, 16, 9)$ é uma tripla engraçada



Mas a tripla $(7, 1, 10)$ não é



A nossa tarefa de hoje é

→ *Contar o número total de triplas engraçadas em* U *e* W

Mas, isso é uma coisa muito fácil.

Basta considerar toda tupla de elementos em U e verificar se eles aparecem na mesma ordem em V.

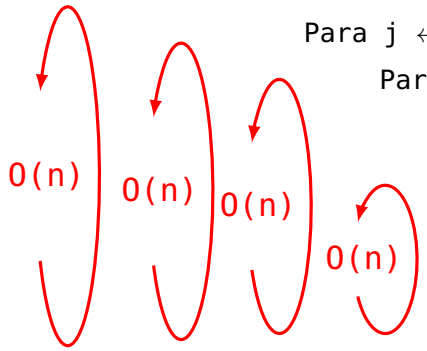
A coisa fica assim

```
func contaTE (U,W)

    cont ← 0

    Para i ← 2 Até N-2
        Para j ← i+1 Até N-1
            Para k ← j+1 Até N
                aux ← 1
                Para ℓ ← 1 Até N
                    Se ( aux = 1  E  W[ℓ] = U[i] )    aux++
                    Sse ( aux = 2  E  W[ℓ] = U[j] )   aux++
                    Sse ( aux = 3  E  W[ℓ] = U[k] )
                        cont++;   Quebra

    Retorna (cont)
```



O problema é que esse algoritmo é uma grande porcaria: $O(n^4)$.

2 Uma pequena esperteza

Mas pelo menos nós aprendemos alguma coisa.

Quer dizer, não é uma boa ideia contar as triplas engraçadas uma por uma.

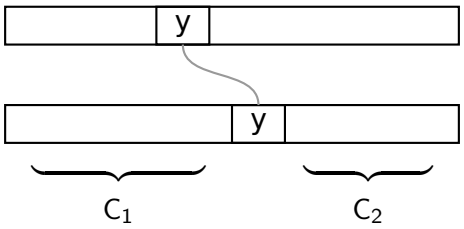
Então agora, nós vamos contar várias triplas engraçadas de uma só vez.

A ideia é considerar o elemento central da tripla como um pivô

$$\begin{array}{c}(\,x, y, z\,)\\ \uparrow \\ \text{pivô}\end{array}$$

Daí, a gente localiza o pivô nas duas listas.

E calcula o número de elementos comuns em U e W, à esquerda e à direita do pivô



Logo, o número de triplas engraçadas que tem y como pivô é igual a $C_1 \times C_2$.

Para implementar essa ideia, nós vamos percorrer a lista U tomando cada um dos seus elementos como pivô.

Daí, a gente localiza o pivô em W e faz a contagem dos elementos comuns.

A coisa fica assim

```
func contaTE-2.0 (U,W)

    cont ← 0

    Para i ← 2 Até N-1
        pivo ← U[i]
        j ← Localiza (pivo,W[1..N])
        C1 ← contaComuns (U[1..i-1],W[1..j-1])
        C2 ← contaComuns (U[i+1..N],W[j+1..N])
        cont ← C1*C2

    Retorna (cont)
```

O(n)

Abaixo nós temos as funções utilizadas no código

```
func Localiza (x,W[a..b])

    Para i ← a Até b
        Se ( W[i] = x )  Retorna(i)
    Retorna(-1)

func contaComuns (U[a..b],W[c..d])

    cont ← 0
    Para i ← a Até b
O(n) ←  Se ( Localiza (U[i],W[c..d]) != -1 )
            cont++
    Retorna(cont)
```

O(n)

O(n)

Agora já não é difícil ver que a complexidade do nosso segundo algoritmo é

$$O(n) \quad \times \quad \Big[O(n) \quad + \quad O(n^2) \Big] \qquad = \qquad O(n^3)$$

Mas isso ainda é muito porcaria.

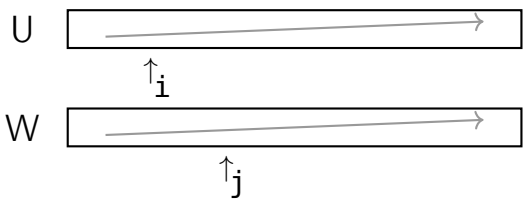
3 Botando a ordenação para funcionar

Então agora, a gente começa a usar aquilo que a gente aprendeu.

Quer dizer, a gente ja aprendeu a ordenar uma lista de maneira eficiente.

E a boa notícia é que é bem rápido contar os elementos comuns quando as duas listas estão ordenadas.

Basta percorrer as listas simultaneamente com dois dedos



A coisa fica assim

```
func contaComunsOrd (U[a..b],W[c..d])

    cont ← 0
    i ← a    j ← c
    Enquanto ( i ≤ b  E  j ≤ d )
        Se ( U[i] = W[j] )
            cont++;   i++;   j++
        Se ( U[i] < W[j] )    i++
        Senão                 j++

    Retorna(cont)
```

O(n)

Daí, antes de fazer a contagem dos elementos comuns, nós precisamos ordenar as faixas de U e W — (*antes e depois do pivô*)

Mas, nós não podemos mudar a ordem dos elementos de U e W para não atrapalhar a contagem.

Então, nós vamos utilizar duas listas auxiliares G, H para isso.

A coisa fica assim

```
func contaTE-3.0 (U,W)

    cont ← 0

    Para i ← 2 Até N-1
        pivo ← U[i]
        j ← Localiza (pivo,W[1..N])
        G[1..i-1] ← Ordena (U[1..i-1])
        G[i+1..N] ← Ordena (U[i+1..N])
        H[1..j-1] ← Ordena (W[1..j-1])
        H[j+1..N] ← Ordena (W[j+1..N])
        C1 ← contaComunsOrd (U[1..i-1],W[1..j-1])
        C2 ← contaComunsOrd (U[i+1..N],W[j+1..N])
        cont ← C1*C2

    Retorna (cont)
```

O(n) O(nlogn) O(n)

E a complexidade do nosso terceiro algoritmo é

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

Bom, as coisas estão melhorando.

Mas isso ainda é uma porcaria.

4 Uma segunda esperteza

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

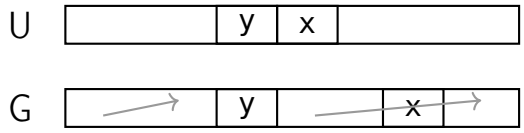
Quer dizer, nós estamos fazendo um monte de ordenações.

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.

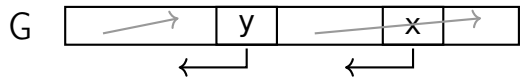
É bem fácil ver como isso funciona para a lista U.

Quer dizer, a ordenação anterior deixa a situação assim



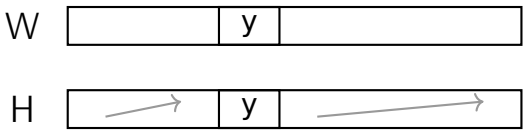
Daí, quando for a vez do x assumir o papel de pivô, basta

- 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



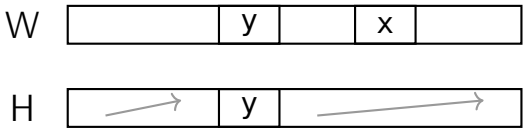
Porque, com a mudança do pivô, vários elementos vão mudar de lado.

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.

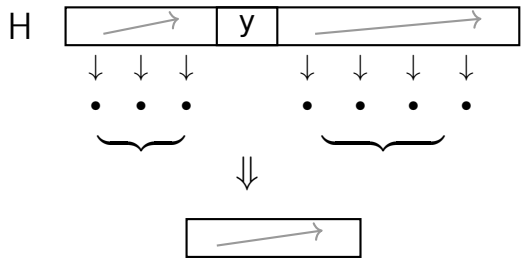
O primeiro passo é visualizar o novo pivô



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.

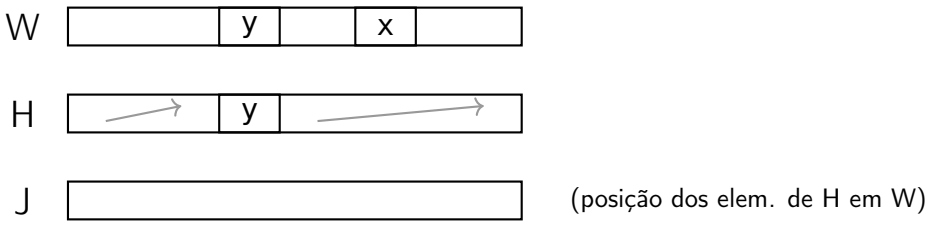
Quer dizer, os elementos que nós queremos localizar são aqueles que estão à esquerda de `x` na lista W.

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])
    G ← ordena2Lados (U,2)
O(nlogn)|
    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

    Retorna (cont)
```

O(n)

Agora, nós já temos um algoritmo que executa em tempo

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

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

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

5 Mudando de ponto de vista

Bom, a gente pode tentar.

Só que para isso, é preciso modificar o nosso ponto de vista — (*porque nós já chegamos ao limite do ponto de vista anterior*)

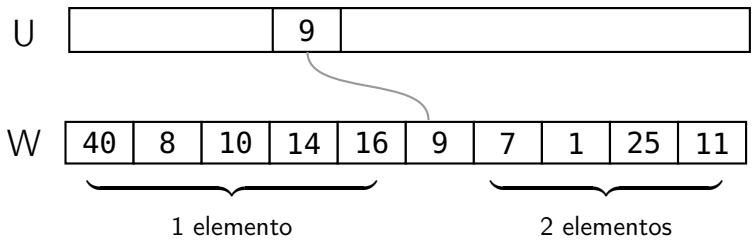
Quer dizer, imagine agora que os elementos da lista U estão em ordem crescente

U	1	7	8	9	10	11	14	16	25	40

W	40	8	10	14	16	9	7	1	25	11

Bom, isso facilita bastante as coisas.

Porque agora, para contar os elementos comuns basta procurar os elementos menores que o pivô do lado esquerdo, e os elementos maiores que o pivô do lado direito.

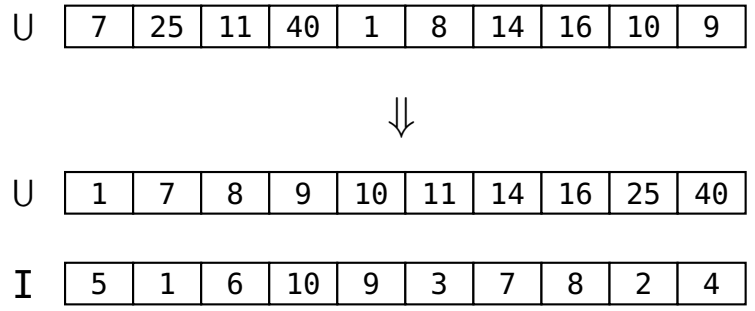


E isso nos dá um algoritmo bem mais simples de tempo $O(n^2)$.

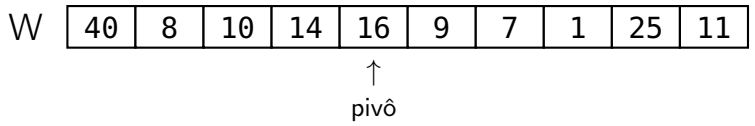
O problema é que no nosso problema a lista U não está ordenada.

Ora, mas se esse é o problema, então a gente pode ordenar a lista U.

E para não perder informação, a gente mantém os índices da configuração original em uma lista auxiliar `I`



Agora, nós percorremos a lista W utilizando cada um dos seus elementos como pivô



E para cada um deles nós contamos

- número de elementos na primeira parte de W que estão à esquerda do pivô em U
- número de elementos na segunda parte de W que estão à direita do pivô em U

Para fazer isso, nós

- fazemos uma busca binária na versão ordenada de U
- obtemos a posição do elemento na versão original de U, consultando a lista `I`
- e depois comparamos com a posição do pivô na versão original de U

Isso leva tempo $O(\log n)$ para cada elemento.

O que dá tempo $O(n \log n)$ para processar cada pivô.

O que dá um algoritmo de tempo $O(n^2 \log n)$.

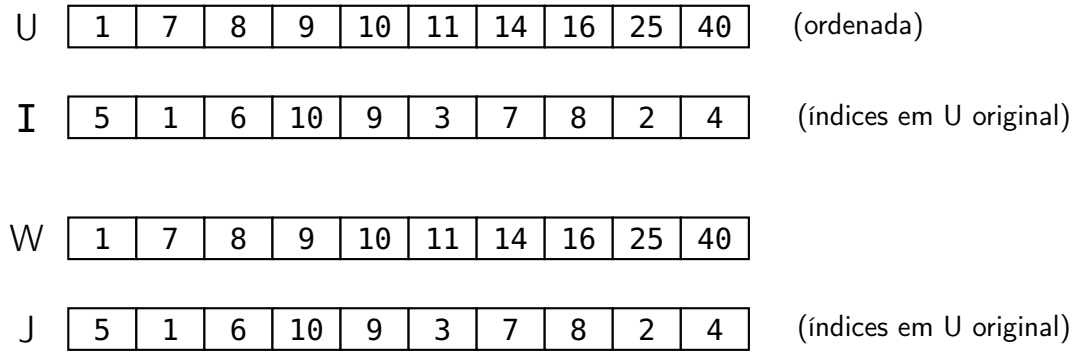
Isso não é muito legal.

Mas aqui a gente pode ser um pouquinho mais esperto.

Quer dizer, nós estamos fazendo várias vezes a busca binária com cada elemento de W.

Então, a ideia é fazer uma vez só.

E guardar os elementos em uma lista auxiliar J



Essas informações podem ser computadas em tempo $O(n \log n)$.

E agora, olhando para a lista J, nós estamos na situação em que U é uma lista ordenada.

E nessa situação, nós sabemos resolver o problema em tempo $O(n^2)$.

Não é legal?

6 A esperteza final

Sim, mas nós ainda não conseguimos um algoritmo mais rápido do que $O(n^2)$.

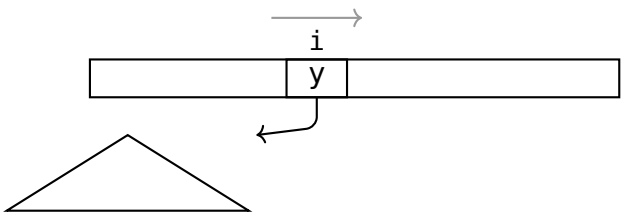
Então agora, nós vamos ver a nossa esperteza final.

Quer dizer, agora o nosso problema é o seguinte

→ *Contar, para cada elemento da lista* J
quantos elementos à sua esquerda são menores do que ele
quantos elementos à sua direita são maiores do que ele

E para fazer isso de maneira eficiente, nós vamos utilizar árvores binárias balanceadas.

A ideia consiste em percorrer a lista J, e inserir cada um deles na árvore



Daí, como a árvore está balanceada, cada inserção leva tempo $O(\log n)$.

E daí, a esperteza consiste em contar a quantidade de elementos menores do que `y` na árvore durante o próprio procedimento de inserção.

Para isso, basta que cada nós da árvore armazene a quantidade de elementos nas suas subárvores esquerda e direita.

Daí, o número de elementos menores do que `y` na árvore é o número de elementos menores do que `y` à esquerda de `y` na lista `J`.

E com esse número, nós conseguimos determinar o número de elementos maiores do que `y` à direita de `y` na lista `J`.

Portanto, em tempo $O(\log n)$ nós conseguimos determinar o número de triplas engraçadas pivoteadas por cada elemento.

E isso nos dá um algoritmo de tempo $O(n \log n)$ para o nosso problema.

Legal, não é?