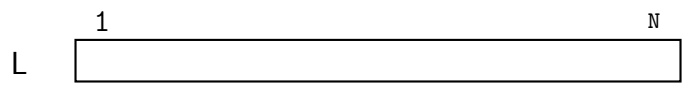
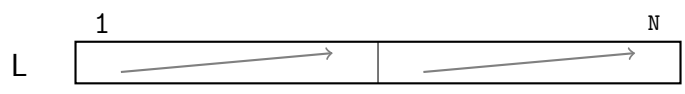
Agora a gente vai ver uma ideia diferente para acelerar o algoritmo da bolha.

Suponha que L é uma lista não ordenada



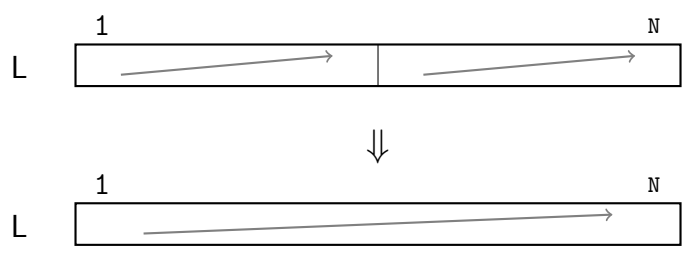
E agora imagine que, ao invés de ordenar a lista inteira, nós ordenamos primeiro uma metade e depois a outra metade



Quer dizer, isso ainda não resolve o problema.

Mas daí, a gente roda o procedimento de intercalação — (*que nós vimos na aula 00.2*)

Quer dizer, o procedimento de intercalação combina as duas metades ordenadas para produzir um lista completamente ordenada



Pronto.

Aqui nós temos mais um algoritmo: a ordenação por metades.

Quer dizer, a coisa funciona assim

```
ordenação-Bolha ( L[1..N/2] )

ordenação-Bolha ( L[N/2+1..N] )

Intercalação ( L[1..N/2] , L[N/2+1..N] )
```

À primeira vista, isso não parece uma boa ideia.

Porque fazer duas vezes a metada de uma coisa parece ser a mesma coisa que fazer a coisa inteira de uma vez só.

Mas, vamos ver o que dizem as contas ...

Para determinar o tempo de execução do algoritmo, nós examinamos a função `Intercalação()`

```
O(1)   i <-- 1;   j <-- N/2 + 1;   k <-- 1

       Enquanto ( i <= N/2  ou  j <= N )
        {
           Se ( i > N/2 )              // a primeira metade já acabou
            {
               A[k]  <--  L[j];   j++;  k++
            }
           Senão Se ( j > N )          // a segunda metade já acabou
            {
               A[k]  <--  L[i];   i++;  k++
            }
           Se ( i < n/2  e  j <= n )
            {
               Se ( L[i] < L[j] )
                {
                   A[k]  <--  L[i];   i++;  k++
                }
               Senão
                {
                   A[k]  <--  L[j];   j++;  k++
        }   }   }
?   O(1)

       Para  i <-- 1  Até  n
          L[i]  <--  A[i]
n   O(1)
```

O laço `Enquanto` dá n voltas, porque a cada volta do laço um elemento de L é copiada para a lista A.

Daí que, o tempo de execução da função `Intercalação()` é

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

Por outro lado, a gente já sabe que `ordenação-Bolha()` executa em tempo $O(n^2)$.

Só que nesse caso, a função está sendo chamada para ordenar metades de tamanho $n/2$.

Logo, cada chamada leva tempo $O(n^2/4)$, e a anotação do código principal fica assim

```
O(n²/4)  ←   ordenação-Bolha ( L[1..N/2] )
O(n²/4)  ←   ordenação-Bolha ( L[N/2+1..N] )
    O(n)  ←   Intercalação ( L[1..N/2] , L[N/2+1..N] )
```

E assim nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

$$O(n^2/4) \quad + \quad O(n^2/4) \quad + \quad O(n) \qquad = \qquad O(n^2/2)$$

Abaixo nós temos a comparação de desempenho entre esse algoritmo e a ordenação da bolha

`ordenação da bolha:`	$O(n^2)$
`ordenação por metades:`	$O(n^2/2)$

Por um lado as duas coisas são a mesma coisa — (*porque* $O(n^2/2)$ *é a mesma coisa que* $O(n^2)$)

Mas por outro lado, isso é apenas o ponto de vista da nossa análise aproximada

— (*onde fatores constantes não contam*)

Na prática, a estratégia realmente reduz o tempo de execução (aprox.) à metade

— (*quando a lista é bem grande ...*)

Isso pode ser comprovado escrevendo programas que implementam os dois algoritmos e realizando uma bateria de testes

`ordenação da bolha:`	. . .
`ordenação via metades:`	. . .

Legal, não é?