Construção e Análise de Algoritmos

aula 02: Acelerando o algoritmo da bolha

1 Introdução

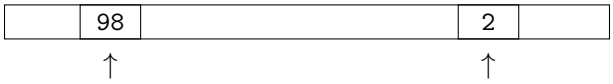
Porque os algoritmos porcaria de ordenação são uma porcaria ?

Bom, é fácil ver porque a ordenação por seleção é uma porcaria: ela faz n vezes uma operação que leva tempo $O(n)$ — (*i.e., encontrar o maior elemento em uma faixa da lista*)

Logo, o seu tempo de execução é $O(n^2)$.

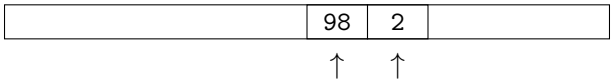
Mas e o algoritmo da bolha, porque ele é uma porcaria?

Bom, imagine que tem um elemento grandão no lado esquerdo da lista, e um elemento pequenininho no lado direito

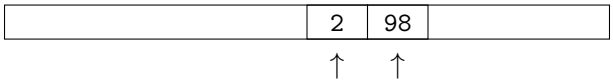


Então, a medida que as varreduras vão acontecendo, o elemento grandão vai andando para a direita, e o elemento pequenininho vai andando para a esquerda

Daí, chega uma hora em que os dois elementos estão lado a lado

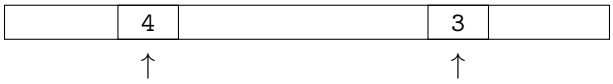


E nesse ponto, eles podem ser comparados e trocados entre si



Certo.

A seguir, a gente nota que isso acontece com toda inversão — (*i.e., quando o elemento da esquerda é maior do que o elemento da direita*)



Porque os elementos só se movem trocando de lugar com o vizinho — (*logo o 4 não pode pular por cima do 3 ...*)

Por outro lado, toda troca desfaz uma inversão.

Logo, o número de trocas realizadas pelo algoritmo da bolha é exatamente igual ao número de inversões na lista.

Agora nós podemos perguntar: *quantas inversões existem em uma lista desordenada?*

Bom, isso depende da lista, claro.

Mas, em uma lista aleatória a probabilidade de que dois elementos estejam invertidos é igual a 1/2.

Logo, metade dos pares de elementos vai formar uma inversão.

E a quantidade total de pares de elementos é

$$\binom{n}{2} \quad = \quad \frac{n(n-1)}{2} \quad = \quad O(n^2)$$

Pronto, aqui a gente já pode ver porque o algoritmo da bolha é uma porcaria.

Mas não é só isso.

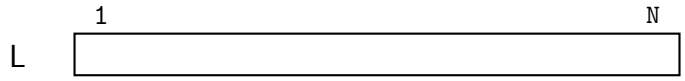
Esse argumento vale para todo algoritmo que troca apenas elementos consecutivos entre si.

E agora a gente já vê porque a ordenação por seleção é uma porcaria.

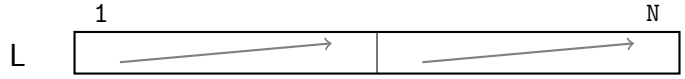
2 Acelerando o algoritmo da bolha

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

Suponha que L é uma lista não ordenada



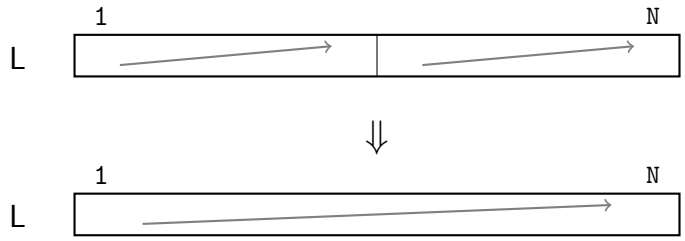
E agora imagine que, ao invés de ordenar a lista inteira, nós ordenamos primeiro uma metade e depois a outra metade



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

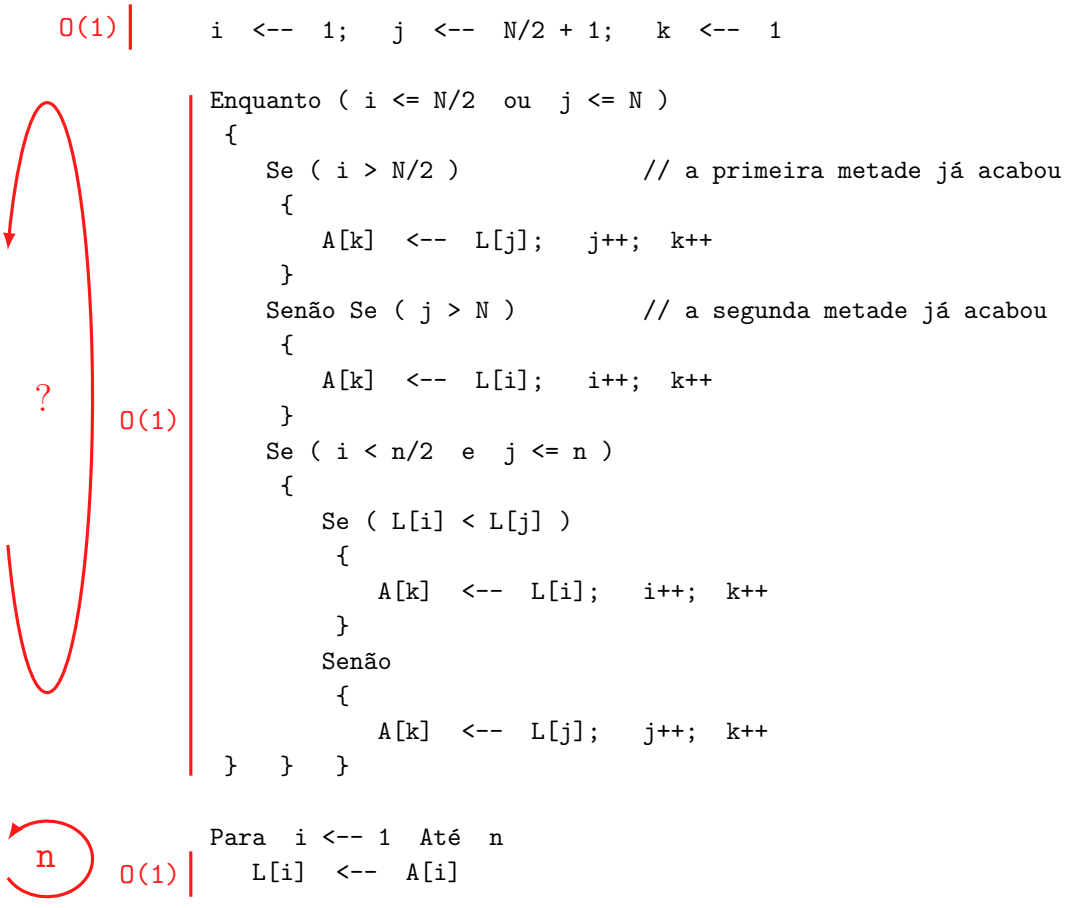
```
ordenação-Bolha ( L[1..N/2] )

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 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(n²)
ordenação por metades:  O(n²/2)
```

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:   .  .  .
```

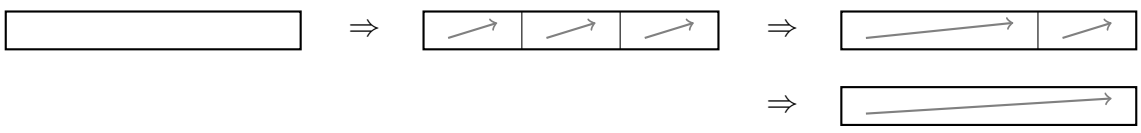
Legal, não é?

3 Ordenação por terços

Certo.

Agora a gente vai levar essa ideia um pouquinho mais longe.

Quer dizer, agora a gente vai quebrar a lista em 3 partes



Abaixo, nós temos o algoritmo que implementa essa ideia

```
ordenação-Bolha ( L[1..N/3] )
```

```
ordenação-Bolha ( L[N/3+1..2N/3] )

ordenação-Bolha ( L[2N/3+1..N] )

Intercalação ( L[1..N/3] , L[N/3+1..2N/3] )

Intercalação ( L[1..2N/3] , L[2N/3+1..N] )
```

E agora é preciso ver se isso torna o algoritmo ainda mais rápido.

Análise

Bom, a gente já sabe que cada ordenação da bolha vai levar tempo

$$O\Big([n/3]^2 \Big) \quad = \quad O\Big(\frac{n^2}{9} \Big)$$

E como são 3 ordenações, isso nos dá

$$3 \ \times \ O\Big(\frac{n^2}{9} \Big) \quad = \quad O\Big(\frac{n^2}{3} \Big)$$

Daí, não é difícil ver que as duas intercalações levam tempo

$$O\Big(\frac{2n}{3} \Big) \quad + \quad O(n) \quad < \quad 2 \cdot O(n)$$

E juntando as duas coisas, nós obtemos a seguinte estimativa

$$O\Big(\frac{n^2}{3} \Big) \quad + \quad 2 \cdot O(n)$$

Comparando com o algoritmo anterior, nós temos o seguinte

$$\begin{array}{rl} \texttt{ordenação via metades:} & O(n^2/2) \ + \ O(n) \\ \texttt{ordenação por terços:} & O(n^2/3) \ + \ 2 \cdot O(n) \end{array}$$

Quer dizer, o termo quadrático diminuiu e o termo linear aumentou.

Logo, quando n é grande, isso é uma boa ideia.

4 Ordenação por blocos

Certo.

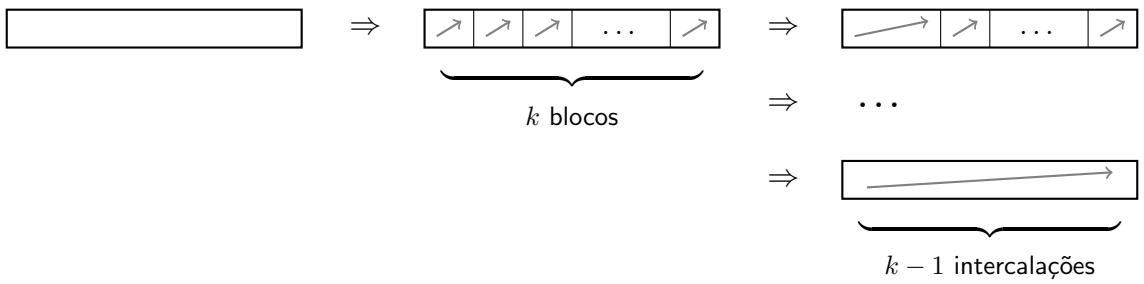
Agora já dá pra ver o que está acontecendo.

Quer dizer, quando a gente divide a lista em mais partes

- o tempo da ordenação diminui — (*o termo quadrático*)
- e o tempo da intercalação aumenta — (*o termo linear*)

Daí que, como o tempo da ordenação é maior, vale a pena continuar dividindo a lista em partes.

Então agora, nós consideramos o algoritmo que divide a lista em k blocos



A coisa fica assim

```
ordenação-Bolha ( L[1..n/k] )
ordenação-Bolha ( L[n/k+1..2n/k] )
. . .
ordenação-Bolha ( L[(k-1)n/k+1..n] )

Intercalação ( L[1..n/k] , L[n/k+1..2n/k] )
. . .
Intercalação ( L[1..(k-1)n/k] , L[(k-1)n/k+1..n] )
```

— (*na prática, a gente faz isso com laços*)

E agora, é preciso ver quanto tempo isso leva.

Análise

Bom, a gente já sabe que cada ordenação pelo algoritmo da bolha vai levar tempo

$$O\Big([\, n/k \,]^2 \Big) \quad = \quad O\Big(\frac{n^2}{k^2} \Big)$$

E como são k ordenações, isso nos dá um total de

$$k \;\times\; O\Big(\frac{n^2}{k^2} \Big) \quad = \quad O\Big(\frac{n^2}{k} \Big)$$

Daí, a gente pode imaginar que cada intercalação leva tempo $O(n)$.

E como são $k - 1$ intercalações, isso nos dá um total de

$$(k-1) \;\cdot\; O(n)$$

Juntando as duas coisas, nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

$$O\Big(\frac{n^2}{k} \Big) \quad + \quad (k-1) \cdot O(n)$$

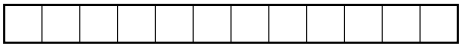
5 Acelerando ao máximo

Legal.

Agora nós estamos prontos para acelerar o nosso algoritmo ao máximo.

Quer dizer, a gente já viu que aumentar o número de blocos é uma boa ideia.

Daí que, aumentando esse número ao máximo, a gente divide a lista em n blocos de tamanho 1



Mas isso não é uma boa ideia ...

Quer dizer, basta olhar o que acontece com o tempo de execução.

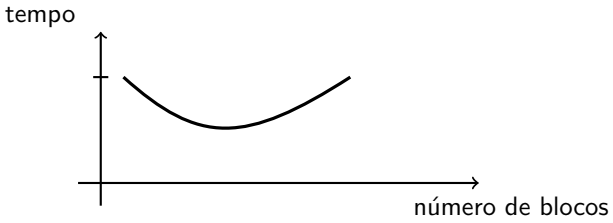
Substituindo o k por n na expressão acima, nós obtemos

$$\left(\frac{n^2}{n}\right) \quad + \quad (n-1) \cdot O(n) \quad = \quad O(n^2)$$

`Hmm ... :/`

Quer dizer, o que está acontecendo é o seguinte

→ *Quando a gente aumenta o número de blocos,*
o tempo de execução primeiro diminui e depois aumenta



Então, nós precisamos descobrir o ponto onde o tempo de execução é o menor possível.

E para isso, nós precisamos entender melhor o que está acontecendo.

Olhando para os dois casos extremos, nós vemos o seguinte

- 2 blocos: $O\big(n^2/2\big) \quad + \quad O(n)$
- n blocos: $O(n) \quad + \quad (n-1) \cdot O(n)$

Quer dizer, no primeiro caso o tempo da ordenação é grande, e o tempo da intercalação é pequeno.

E no segundo caso, o tempo da ordenação é pequeno e o tempo da intercalação é grande.

Daí que, o tempo de execução mínimo acontece quando esses dois tempos são iguais.

Usando os termos da expressão acima, nós escrevemos

$$O\Big(\frac{n^2}{k}\Big) \quad = \quad (k-1) \cdot O(n)$$

o que a gente reescreve de maneira simplificada assim

$$\frac{n^2}{k} \quad = \quad k \cdot n$$

o que pode ser reescrito assim

$$n \quad = \quad k^2$$

o que nos dá

$$k \quad = \quad \sqrt{n}$$

Quer dizer, a situação ideal acontece quando nós dividimos a lista em $\sqrt{n}$ blocos
— (*com tamanho* $\sqrt{n}$ *cada um*)

Nessa situação, o tempo de execução do algoritmo é

$$O\Big(\frac{n^2}{\sqrt{n}}\Big) \quad + \quad (\sqrt{n}-1) \cdot O(n) \quad = \quad O\big(n \cdot \sqrt{n}\big)$$

E agora, nós temos um algoritmo que é bem mais rápido do que os algoritmos que nós vimos na aula passada.

Não é legal?

6 Acelerando o algoritmo da bolha mais ainda

Agora nós vamos utilizar outra ideia.

Quer dizer, a gente começa perguntando

- *Qual é a ineficiência básica do algoritmo da bolha ?*

Porque, se a gente consegue remover essa ineficiência, então o algoritmo pode ficar mais rápido.

Bom, a ineficiência básica do algoritmo da bolha é a seguinte

→ *os elementos andam apenas uma posição de cada vez*

Porque o que faz os elementos andarem é a troca.

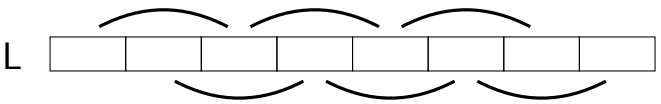
E a troca sempre troca dois elementos adjacentes.

Certo.

Então a solução é simples.

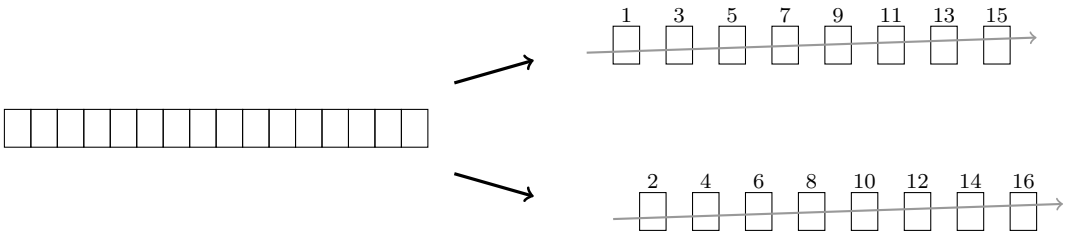
Quer dizer, a ideia é trocar elementos que estejam mais distantes um do outro.

A versão mais simples dessa ideia consiste em comparar e trocar elementos à distância 2



Na prática, isso corresponde a executar o algoritmo da bolha sobre elementos ímpares e elementos

pares, em separado



Em geral, isso ainda não resolve o problema completamente.

Mas daí, a gente executa varreduras até que a lista fique completamente ordenada

Análise

Mas será que isso é uma boa ideia?

Bom, a primeira observação é que a gente está aplicando a estratégia da divisão outra vez

→ *nós executamos o algoritmo da bolha uma vez em cada metade da lista*

E a gente já sabe que isso leva tempo

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

Quer dizer, até aqui a gente ganhou algum tempo.

Mas, é preciso ver quanto tempo leva para terminar o trabalho.

Para fazer isso, a gente imagina que a lista contém os números $1, 2, \ldots, n$.

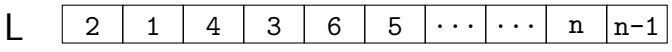
E daí, a gente considera duas situações

caso 1: Imagine que

- os números pares estão nas posições ímpares
- os números ímpares estão nas posições pares

— (*em uma ordem qualquer*)

Então, após a ordenação das posições ímpares e das posições pares, a lista fica assim



E aqui, basta uma varredura para colocar a lista em ordem.

O que leva tempo $O(n)$.

Logo, nesse caso, o tempo de execução do algoritmo é

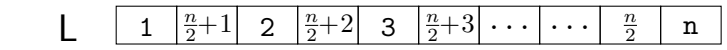
$$\frac{n^2}{2} \quad + \quad n$$

caso 2: Imagine que

- os números $1, 2, \ldots, \frac{n}{2}$ estão nas posições ímpares
- os números $\frac{n}{2} + 1, \ldots, n$ estão nas posições pares

— (*em uma ordem qualquer*)

Então, após a ordenação das posições ímpares e das posições pares, a lista fica assim



E agora, é preciso fazer varreduras para colocar a lista em ordem.

Quantas varreduras?

Bom, vamos concentrar a nossa atenção no elemento $n/2$.

Esse elemento está a $n/2$ posições de distância do seu lugar correto — (*aprox.*)

E a cada varredura ele anda uma posição nessa direção — (*porque?.*)

Logo, vão ser necessárias $n/2$ varreduras para colocar a lista em ordem — (*aprox.*)

E o tempo de execução do algoritmo nesse caso é

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

Quer dizer, a gente não ganhou nada.

6.1 O algoritmo bolha-32

Bom, a gente ainda não ganhou nada.

Mas, a gente ainda pode ganhar alguma coisa.

Quer dizer, a gente pode ganhar entendimento.

E o entendimento vem quando a gente faz as seguintes perguntas

- *O que é que torna o melhor caso bom ?*
  O que é que torna o pior caso ruim ?

Bom, é mais fácil começar com a segunda pergunta.

Quer dizer, o pior caso é ruim porque

1. Os números pequenos estão todos nas posições ímpares.

   Daí que, na hora em que a gente ordena as posições ímpares, nós vamos ter números pequenos no final da lista.

2. Os números grandes estão todos nas posições pares.

   Daí que, na hora em que a gente ordena as posições pares, nós vamos ter números grandes no início da lista.

E daí, é preciso fazer um monte de varreduras para colocar a lista em ordem.

Por outro lado, o melhor caso é bom porque

1. Os números pequenos estão bem distribuídos entre as posições pares e ímpares.

   Daí que, na hora em que a gente ordena as posições pares e ímpares, eles vão parar todos no início da lista.

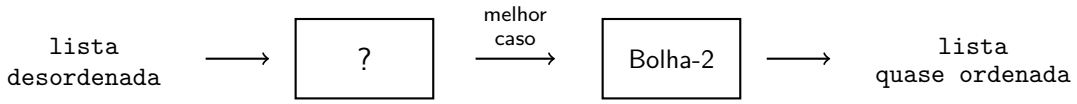
2. Os números grandes estão bem distribuídos entre as posições pares e ímpares.

   Daí que, na hora em que a gente ordena as posições pares e ímpares, eles vão parar todos no fim da lista.

Legal.

Agora que a gente entendeu, nós podemos tentar forçar que o melhor caso aconteça.

Quer dizer, a gente pode adicionar uma etapa ao algoritmo, que prepara o melhor caso para a fase de ordenação



E aqui é a hora para uma grande esperteza.

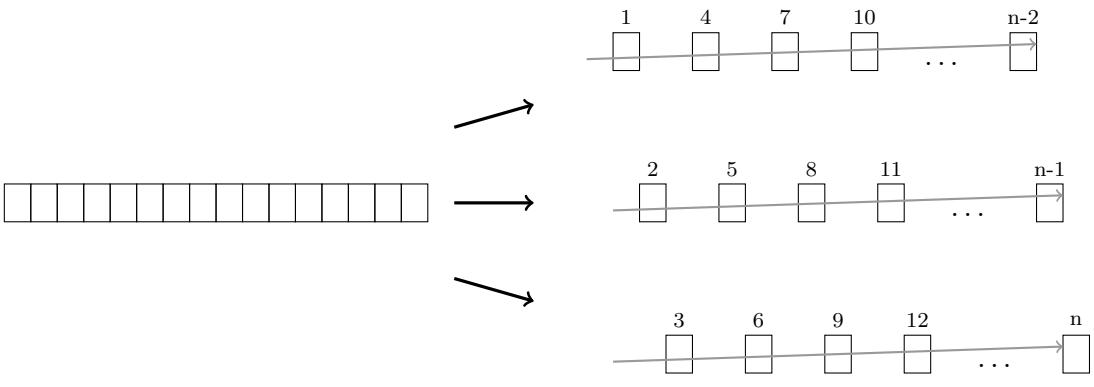
Quer dizer, a ideia é colocar uma ordenação da bolha com salto 3 na etapa preliminar

Porque?

Bom, aqui há mais coisa pra gente entender outra vez.

Quer dizer, na prática, o algoritmo da bolha com salto 3

- ordena as posições múltiplas de 3 — (*i.e., da forma* $3k$)
- ordena as posições da forma $3k + 1$
- ordena as posições da forma $3k + 2$



E isso faz com que, em cada lista

- os pequenos fiquem do lado esquerdo
- e os grandes fiquem do lado direito

Mas, agora veja quem está do lado esquerdo.

Sim! a metade das posições do lado esquerdo é par, e a outra metade é ímpar.

E a mesma coisa acontece do lado direito.

Não é legal?

Sim, mas ainda tem uma outr coisa mais legal.

Só que antes, é preciso fazer uma outra observação

→ *As ordenações feitas pela caixinha* Bolha-2

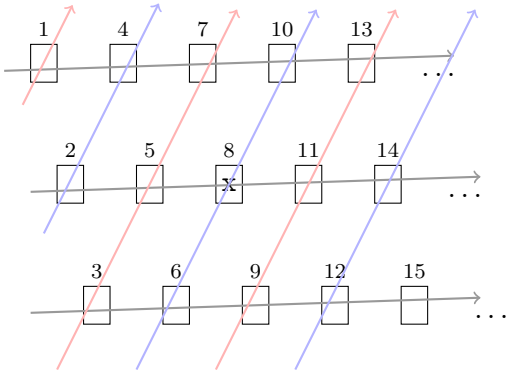
não desfazem as ordenações feitas pela caixinha Bolha-3

Quer dizer, as posições da forma $3k$, $3k+1$, $3k+2$ continuam ordenadas após a execução do Bolha-2.

Porque?

Bom, isso é um excelente exercício para o pensamento analítico — (*e será deixado pra você ...*)

O que é importante é que nesse ponto, nós temos um monte de relações de ordem



E agora, a gente concentra a atenção sobre o elemento `x` que está na posicão 8

Quer dizer,

- o `x` é menor do que o elemento da posicão 11, e todos que estão à direita dele
  — (*porque a linha* $3k+2$ *está ordenada*)
- o `x` é menor do que o elemento da posicão 10, e todos que estão à direita dele
  — (*porque as posições pares e a linha* $3k+1$ *estão ordenadas*)
- o `x` é menor do que o elemento da posicão 12, e todos que estão à direita dele
  — (*porque as posições pares e a linha* $3k$ *estão ordenadas*)

A gente também pode fazer um raciocínio semelhante para a esquerda.

E a partir daí, a gente conclui que

- ou o elemento `x` está na posição correta
- ou ele está ao lado da posição correta

A mesma coisa, claro, vale para todos os elementos da lista.

Logo, apenas uma varredura basta para deixar a lista completamente ordenada.

Não é legal?

Sim, e abaixo nós temos o esquema completo do algoritmo Bolha-32

O tempo de execução desse algoritmo é

$$3 \times \left(\frac{n}{3}\right)^2 \quad + \quad 2 \times \left(\frac{n}{2}\right)^2 \quad + \quad n \quad = \quad \frac{5n^2}{6} \quad + \quad n$$

E aqui a gente vê que ele é um pouquinho mais rápido do que o algoritmo

6.2 O algoritmo Shellsort

(. . .)