**Construção e Análise de Algoritmos**

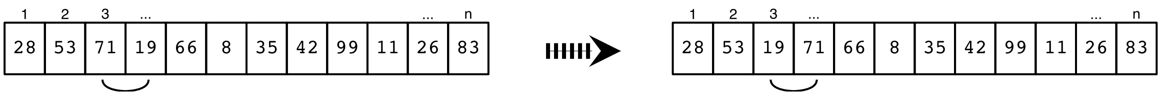
**aula 01: O algoritmo de ordenação *Shellsort***

# 1 Introdução

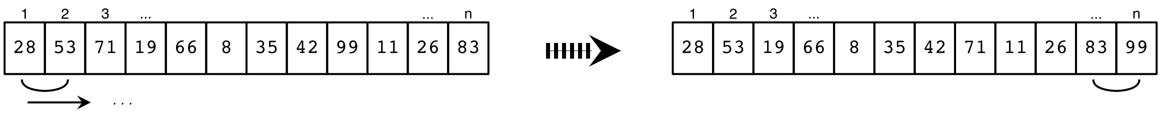
Considere o problema de ordenar uma lista de números.

| 1 | 2 | 3 | ... | | | | | | | ... | n |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 28 | 53 | 71 | 19 | 66 | 8 | 35 | 42 | 99 | 11 | 26 | 83 |

Uma maneira natural de resolver esse problema consiste em identificar pares de elementos que estão na ordem errada e inverte-los de posição. Por exemplo,




Daí, nós podemos imaginar esse procedimento sendo aplicado em todos os pares de posições consecutivas da lista, trocando os elementos de posição sempre que necessário.



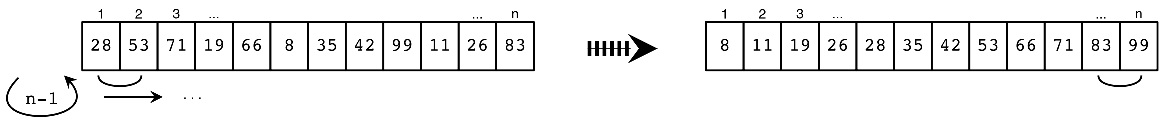

Abaixo nós temos o pseudo-código dessa operação de varredura:

```
Para i = 1 Até n-1
    Se ( V[i] > V[i+1] )
    {
        aux <-- V[i]; V[i] <-- V[i+1]; V[i+1] <-- aux;
    }
```

Examinando com atenção o resultado desse procedimento, nós observamos que o maior elemento da lista foi levado para a última posição.

E, se o procedimento for executado outra vez, o segundo maior elemento será levado para a penúltima posicão.

Agora é fácil ver que basta repetir esse procedimento `n-1` vezes para ordenar a lista inteira.




```
Repita n-1 vezes
    Para i = 1 Até n-1
        Se ( V[i] > V[i+1] )
        {
            aux <-- V[i]; V[i] <-- V[i+1]; V[i+1] <-- aux;
        }
```

Esse algoritmo é conhecido como o *algoritmo de ordenação da bolha*.

Agora que nós já temos um algoritmo de ordenação, não é difícil melhorá-lo um pouquinho.

A observação chave é que, depois que o maior elemento foi colocado na última posição, ele não vai mais sair de lá.

Isto é, não é preciso realizar a varredura até a última posição depois da primeira iteração.

Analogamente, não é preciso realizar a varredura até a penúltima posição depois da segunda iteração.

E assim por diante.

Essa observação nos permite modificar o algoritmo da seguinte maneira

```
1.     Para k = 1 Até n-1
2.         Para i = 1 Até n-k
3.             Se ( V[i] > V[i+1] )
4.             {
5.                 aux <-- V[i]; V[i] <-- V[i+1]; V[i+1] <-- aux;
6.             }
```

reduzindo o seu tempo de execução essencialmente à metade. (Porque?)

Legal!

Agora, nós queremos saber qual é o tempo de execução desse algoritmo.

Assuma que o bloco formado pelas linhas `3-6` executa em uma unidade de tempo.

Então, é fácil ver que a primeira execução do laço mais interno (linha `2`) leva tempo $n-1$.

A segunda execução desse laço leva tempo $n-2$.

A terceira execução leva tempo $n-3$.

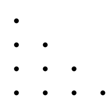
E assim por diante.

Logo, o algoritmo da bolha executa em tempo

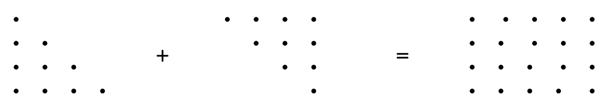
$$(n-1) \; + \; (n-2) \; + \; \ldots \; + \; 3 \; + \; 2 \; + \; 1$$

Mas, o quanto é isso?

Bom, essa soma corresponde basicamente a um triângulo



e dois triângulos formam um retângulo



Um retângulo de altura $n-1$ e comprimento $n$ vale $n(n-1)$.

Logo, o algoritmo da bolha executa em tempo $\frac{n(n-1)}{2}$.

Na área de análise de algoritmos, no entanto, nós não estamos interessados em expressões exatas.

Então, nós vamos dizer simplesmente:

- *O tempo de execução do algoritmo da bolha é* $O(n^2)$ *(i.e., da ordem de* $n^2$*)*

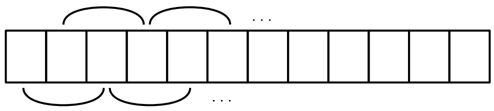
## 2 Acelerando o algoritmo da bolha

E agora?

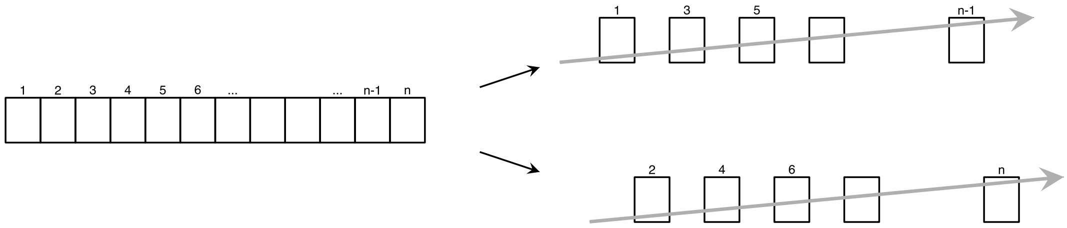
O que nós podemos fazer para acelerar o algoritmo da bolha ainda mais?

Bom, o algoritmo da bolha é lento porque ele move os elementos uma posição de cada vez.

Então, nós podemos mover os elementos duas posições de cada vez!




Mas, isso é a mesma coisa que executar o algoritmo da bolha uma vez nas posições pares e uma vez nas posições ímpares:




É fácil ver a ordenação de cada parte vai levar tempo $n^2/4$, o que dá um total de $n^2/2$ — a metade do algoritmo original!.

Mas, também é fácil ver que esse procedimento nem sempre coloca a lista em ordem ...

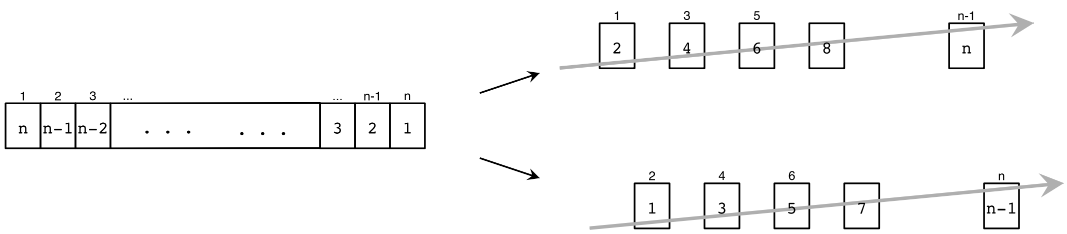
Por exemplo, se o menor elemento da lista estiver em uma posição par no início, então após a execução do procedimento ele estará na posição 2, que não é o seu lugar correto.

Por outro lado, nós podemos esperar que os elementos sempre terminam próximos à sua posição correta.

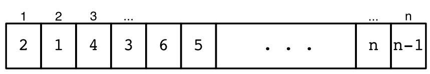
E, nesse caso, não seria muito trabalhoso terminar de ordenar a lista.

Vejamos um exemplo concreto.

Suponha que no início a lista está em ordem decrescente (e que $n$ é par)




Então, após as duas ordenações nós obtemos o seguinte resultado:




E agora é fácil ver que uma simples varredura na lista, trocando os elementos adjacentes de posição quando necessário, coloca a lista em ordem.

Como a varredura leva tempo $n$ (aproximadamente), o tempo total da ordenação é

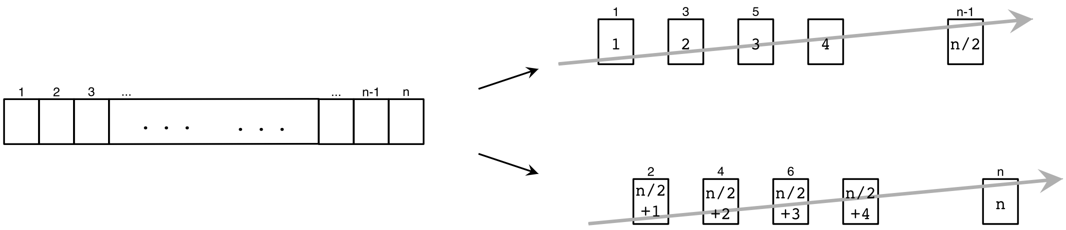
$$\frac{n^2}{4} + \frac{n^2}{4} + n = \frac{n^2}{2} + n$$

o que é aproximadamente a metade do tempo original!

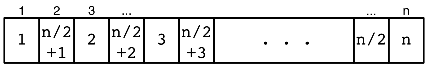
Mas, infelizmente, as coisas nem sempre são tão boas assim ...

Considere uma configuração inicial da lista onde as posições ímpares contém os $n/2$ menores elementos, e as posições pares contém os $n/2$ maiores elementos (não importa exatamente como).

Então, as duas ordenações




produzem o seguinte resultado




*Quantas varreduras são necessárias para colocar essa lista em ordem?*

A cada varredura, os elementos grandes se movem uma posição para frente, e os elementos pequenos se movem uma posição para trás.

Como o elemento $n/2 + 1$ está a $n/2$ posições de distância do seu lugar correto (e o elemento $n/2$ também), serão necessárias $n/2$ varreduras para ordenar a lista.

Logo, o tempo total da ordenação é dado por:

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

Ou seja, dessa vez nós não ganhamos nada ...

Mas, nós aprendemos alguma coisa!

Nós aprendemos que se o números grandes e pequenos estão distribuídos de maneira aproximadamente igual entre as posições pares e ímpares da lista, então as duas ordenações iniciais colocam a lista em uma configuração quase ordenada.

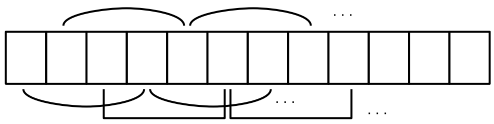
E daí bastam algumas poucas varreduras para completar o trabalho.

A questão é: *como é que nós podemos garantir que os números grandes e pequenos estejam distribuídos de maneira uniforme entre as posições pares e ímpares da lista?*

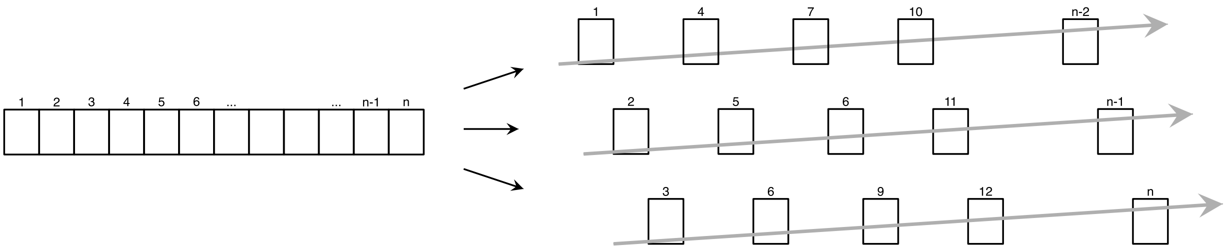
Uma primeira ideia seria embaralhar todos os elementos aleatoriamente antes de começar o trabalho — nós vamos examinar essa ideia algumas aulas adiante.

A seguir nós vamos ver uma outra ideia.

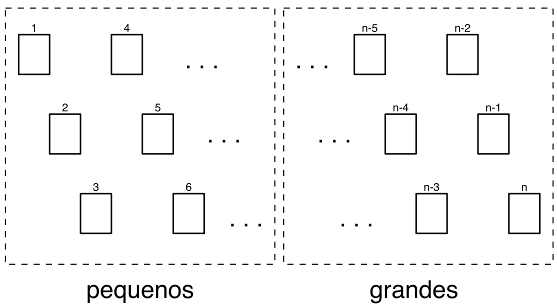
Vamos ver o que acontece quando nós executamos o algoritmo da bolha dando saltos de tamanho 3.



Como nós já sabemos, isso corresponde a separar os elementos que se encontram em posições da forma $3k$, $3k+1$, $3k+2$, e ordenar essas sublistas uma de cada vez.




A observação interessante, a seguir, é que após as ordenações das três sublistas, os números grandes de cada uma delas ficarão do lado direito, e os números pequenos ficaram do lado esquerdo




E, mais interessante ainda, é a observação de que metade das posições no bloco da esquerda são pares e metade são ímpares (e o mesmo vale para o bloco da direita)

Ou seja, esse procedimento produz exatamente a configuração que nós queríamos!

A seguir, nós aplicamos o algoritmo da bolha com saltos de tamanho 2.

E, finalmente, basta uma única varredura para completar a ordenação — esse resultado será discutido no apêndice dessa nota de aula.

Esse algoritmo executa em tempo

$$
\begin{aligned}
3 \cdot \frac{n^2}{9} \;+\; 2 \cdot \frac{n^2}{4} \;+\; n \;&=\; \frac{n^2}{3} \;+\; \frac{n^2}{2} \;+\; n \\
&=\; \frac{5n^2}{6} \;+ n
\end{aligned}
$$

e ao menos alguma coisa nós acabamos ganhando ...

## 3 O algoritmo Shellsort

Vejamos o que nós acabamos de descobrir.

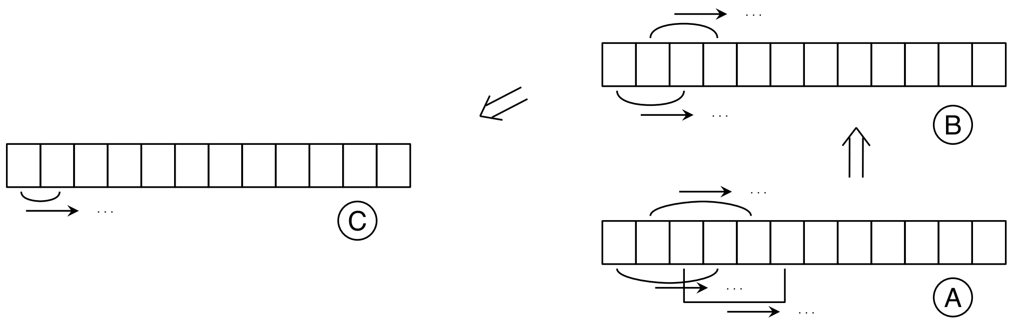
Para ordenar uma lista de tamanho $n$, ao invés de aplicar diretamente o algoritmo da bolha, nós podemos

A. executar o algoritmo com saltos de tamanho 3

B. executar o algoritmo com saltos de tamanho 2

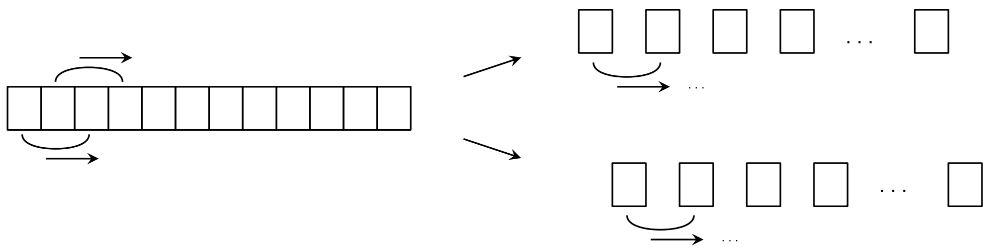
C. realizar uma única varredura com saltos de tamanho 1

A figura abaixo ilustra essas três etapas.

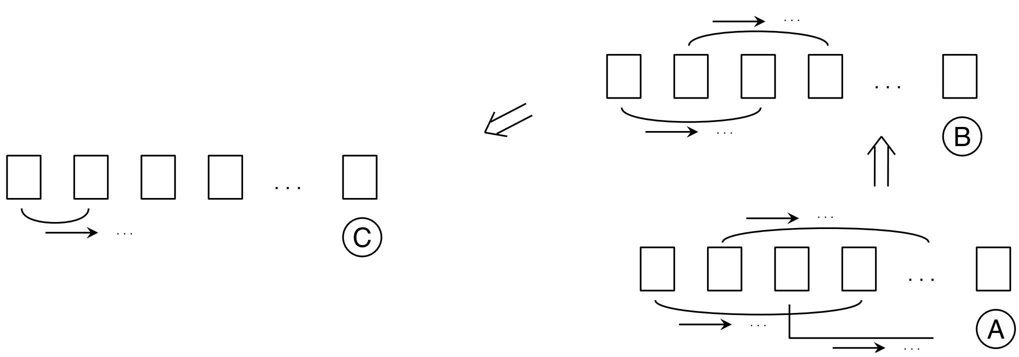



Apesar de ser um pouquinho mais complicado, esse procedimento é mais rápido do que a versão original do algoritmo da bolha.

Além disso, nós também vimos que a execução do algoritmo da bolha com saltos de tamanho 2 pode ser vista como duas execuções independentes do algoritmo da bolha (uma sobre as posições pares e outra sobre as posições ímpares).



Ora, mas se a execução do algoritmo da bolha pode ser acelerada utilizando o esquema acima, então nós também podemos acelerar as execuções do algoritmo sobre as posições pares e ímpares.



Fazendo isso, cada execução com salto 2, que levava tempo $n^2/4$, passa a executar em tempo

$$\frac{5}{6} \cdot \left(\frac{n}{2}\right)^2 \ + \ \frac{n}{2} \quad = \quad \frac{5n^2}{24} \ + \ \frac{n}{2}$$

de modo que o tempo total do algoritmo é reduzido para

$$\begin{aligned} 3 \cdot \tfrac{n^2}{9} \ + \ 2 \cdot \left(\tfrac{5n^2}{24} \ + \ \tfrac{n}{2}\right) \ + \ n \quad &= \quad \tfrac{n^2}{3} \ + \ \tfrac{10n^2}{24} \ + \ n \ + \ n \\ &= \quad \tfrac{3n^2}{4} \ + 2n \end{aligned}$$

A mesma coisa, é claro, também pode ser feita com as execução com salto 3.

Mas, aqui acontece uma coisa ainda mais interessante.

Note que, ao aplicar o esquema acima sobre as execuções com salto 2, nós estamos efetivamente modificando o esquema para:

A. executar o algoritmo com saltos de tamanho 3

   B1. executar o algoritmo com saltos de tamanho 4

   B2. executar o algoritmo com saltos de tamanho 6

B. realizar uma única varredura com saltos de tamanho 2

C. realizar uma única varredura com saltos de tamanho 1

De fato, fazendo novamente as contas, nós vemos que o esquema corresponde a um algoritmo que executa em tempo

$$3 \cdot \frac{n^2}{9} \ + \ 4 \cdot \frac{n^2}{16} \ + \ 6 \cdot \frac{n^2}{36} \ + \ 2n \quad = \quad \frac{3n^2}{4} \ + 2n$$

A seguir, ao modificar o esquema para acelerar a execução com saltos de tamanho 3, nós obtemos

   A1. executar o algoritmo com saltos de tamanho 6

   A2. executar o algoritmo com saltos de tamanho 9

A. realizar uma única varredura com saltos de tamanho 3

   B1. executar o algoritmo com saltos de tamanho 4

   B2. executar o algoritmo com saltos de tamanho 6

B. realizar uma única varredura com saltos de tamanho 2

C. realizar uma única varredura com saltos de tamanho 1

E agora existem duas execuções do algoritmo com saltos de tamanho 6: A1 e B2.

Mas, isso não é necessário! (veja a discussão no Apêndice.)

Isto é, o passo B2 pode ser eliminado, e o novo algoritmo executa em tempo

$$9 \cdot \frac{n^2}{81} \ + \ 6 \cdot \frac{n^2}{36} \ + \ 4 \cdot \frac{n^2}{16} \ + \ 3n \quad = \quad \frac{19n^2}{36} \ + 3n \quad \simeq \quad \frac{1n^2}{2} \ + 3n$$

Agora o tempo está começando a diminuir bem mais rápido ...

Para entender o que está acontecendo, nós reproduzimos abaixo os tempo de execução dos quatro algoritmos que nós já vimos na aula de hoje:

$$n^2$$

$$3 \cdot \frac{n^2}{9} \;+\; 2 \cdot \frac{n^2}{4} \;+\; n$$

$$3 \cdot \frac{n^2}{9} \;+\; 4 \cdot \frac{n^2}{16} \;+\; 6 \cdot \frac{n^2}{36} \;+\; n \;+\; n$$

$$9 \cdot \frac{n^2}{81} \;+\; 6 \cdot \frac{n^2}{36} \;+\; 4 \cdot \frac{n^2}{16} \;+\; n \;+\; n \;+\; n$$

A ideia aqui é que os termos quadráticos estão sendo substituído por $n$, em troca da introdução de novos termos quadráticos divididos por produtos de potências de 2 e 3.

Esses novos termos quadráticos, é claro, também podem ser substituídos por $n$, acarretando a introdução de termos quadráticos menores ainda (da forma $n^2/2^p3^q$).

No final, após todas as substituições terem sido realizadas, nós teremos uma quantidade de termos $n$ que corresponde à quantidade de números da forma $2^p3^q$ menores que $n^2$.

Não é difícil verificar que a quantidade de números dessa forma é (da ordem de) $\log^2 n$.

E isso nos dá um tempo de execução (da ordem de):

$$\underbrace{n \;+\; n \;+\; n \;+\; \ldots \;+\; n}_{\log^2 n} \quad = \quad O(n \log^2 n)$$

O algoritmo que corresponde a essa ideia consiste em executar o algoritmo da bolha sucessivamente com saltos de tamanhos:

$$1 \quad 2 \quad 3 \quad 4 \quad 6 \quad 9 \quad 8 \quad 12 \quad 18 \quad 27 \quad 16 \quad \ldots$$

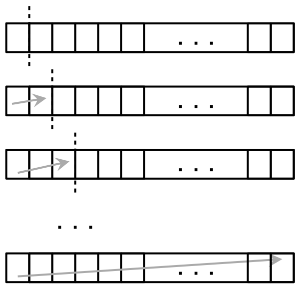
Esse algoritmo é conhecido como *Shellsort*.

( . . . )

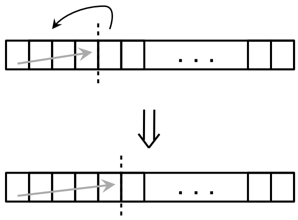
## Exercícios

1. **Ordenação por inserção**

   O algoritmo de ordenação por inserção coloca gradualmente os elementos em ordem, trabalhando da esquerda para a direita.

   A ideia é que, a cada passo, o primeiro elemento da porção desordenada é inserido na sua posição correta na porção ordenada (deslocando outros elementos para a direita)

   a) Apresente o pseudo-código do algoritmo de ordenação por inserção.

   b) Estime (a ordem de magnitude d') o tempo de execução do seu algoritmo.

2. **O pior algoritmo de ordenação do mundo** (DESAFIO)

   Esse algoritmo consiste em apenas um laço:

```
Enquanto ( a lista ainda não está ordenada )
 {
     Embaralhe os elementos da lista aleatoriamente
 }
```

   Na próxima aula, nós vamos ver que é possível embaralhar aleatoriamente os elementos de uma lista de tamanho em tempo $O(n \log n)$.

   Determine o tempo médio de execução desse algoritmo

3. **O algoritmo de ordenação do preguiçoso** (OPCIONAL)

   Esse algoritmo ordena uma lista com $n$ números utilizando $n$ despertadores.

   Para cada número da lista, você ajusta o alarme de um despertador para uma certa hora/minutos, de modo que se $x < y$ então o despertador de $x$ toca antes que o despertador de $y$.

   (Você também escreve o número correspondente em cada despertador.)

   E daí, você vai dormir ...

   Quando um desepertador toca, você acorda lê o número dele e o adiciona a uma lista.

   No dia seguinte, a lista estará completamente ordenada!

   Estime o tempo de execução desse algoritmo.