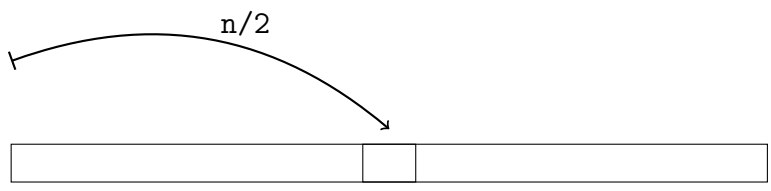
Mas, a gente ainda pode fazer melhor do que isso.

E a ideia é inverter as coisas.

Quer dizer, a busca pula-pula acelerada começa dando pulos pequenos, e daí vai aumentando o tamanho do pulo.

Agora, a gente vai começar dando pulos grandes, e daí vai diminuindo o tamanho do pulo.

A coisa começa com um pulo de tamanho $n/2$, que nos leva para o meio da lista



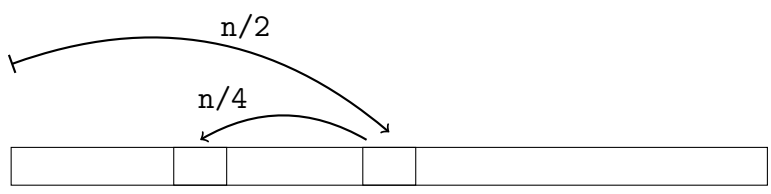
E daí, existem 3 possibilidades

1. Se esse é o elemento x que a gente procura, então a busca acaba aí
2. Se ele é maior do que x, então a gente pode descartar o lado direito
   — (*porque lá, todo mundo também é maior*)
3. Se ele é menor do que x, então a gente pode descartar o lado esquerdo
   — (*porque lá, todo mundo também é menor*)

E daí, a coisa continua assim.

Quer dizer, imagine que a gente está no caso 2 — (*onde o lado direito é descartado*)

Então agora, a gente dá um pulo para a esquerda de tamanho $n/4$



E faz a mesma coisa outra vez.

A coisa continua assim até que o elemento seja encontrado.

Ou até que a lista inteira seja descartada.

Abaixo nós temos o algoritmo da busca binária

```
inicio = 1;   fim = N;   achei = 0

Enquanto ( inicio ≤ fim )

     meio = (inicio + fim) / 2

     Se ( L[meio] == x )     achei = 1;   Quebra

     Se ( L[meio] > x )      fim = meio - 1

     Se ( L[meio] > x )      fim = meio - 1

Se ( achei == 1 )     Imprime ('Achei')

Senão                 Imprime ('Não está lá')
```

Análise

Dessa vez a análise não é difícil.

Porque a distância entre `inicio` e `fim` vai reduzindo à metade a cada volta do laço.

Logo, o laço dá no máximo $\log_2(n)$ voltas.

E o algoritmo executa em tempo

$$O(\log n)$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar 20 deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

Não é legal?

Implementação recursiva

A busca binária não é realmente uma esperteza.

Quer dizer, ela é a aplicação de uma ideia bastante geral: a técnica da divisão.

Como o nome já diz, a ideia é

→ *dividir o problema em duas metades, e daí resolver as duas metades*

Mas dessa vez, a situação é melhor ainda, porque a gente só precisa resolver uma metade

— (*ou fazer a busca em apenas uma metade*)

Quando a gente vê as coisas assim, é mais natural escrever o algoritmo de maneira recursiva

```
func Busca-bin ( x, L, inicio, fim )

     Se ( inicio > fim )   Retorna 0

     meio = (inicio + fim) / 2

     Se ( L[meio] == x )     Retorna 1

     Se ( L[meio] > x )      resp = Busca-bin ( x, L, inicio, meio-1 )

     Se ( L[meio] < x )      resp = Busca-bin ( x, L, meio+1, fim )

     Retorna resp
```

Análise

Aqui, a gente vê que o algoritmo executa em tempo $O(\log n)$, porque ele faz no máximo $\log_2(n)$ chamadas recursivas — (*porque?*)