A gente começa com a seguinte pergunta

- *Será que no pior caso a busca tem que levar tempo $O(n)$?*

Bom, se a lista está desordenada a resposta é Sim.

Porque no pior caso, a gente tem que olhar todos os elementos para ter certeza se o elemento está lá ou não.

Mas, quando a lista está ordenada a gente não precisa olhar todos os elementos.

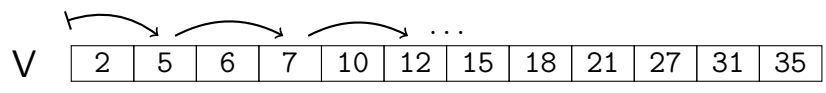
Porque?

Quer dizer, como?

Bom, existem várias maneiras de fazer isso.

E a gente vai começar com a mais simples delas.

Quer dizer, a ideia da busca pula-pula é ir pulando os elementos de 2 em 2, em busca do número x



E daí, podem acontecer várias coisas

1. Se o x está em uma posição par, então a gente cai em cima dele, e ele é encontrado.
2. Se o x está em uma posição ímpar, então a gente passa por cima dele — (*encontrando um elemento maior que x, ou caindo fora da lista*)

   Daí, a gente volta uma posição para encontrá-lo.
3. Se o x está não está na lista, então a gente passa do ponto — (*como no caso 2*)

   E daí, volta uma posição para descobrir que ele não está lá.

A coisa fica assim

```
i = 2
Enquanto ( i ≤ N   E   V[i] < x )                    // pula até não poder mais
     i = i + 2
                                                     // e dai, vê o que aconteceu:
Se ( i > N )                                         // se passou do fim
     Se ( V[N] == x )     Imprime ('Achei')
     Senão                Imprime ('Não está lá')
Senão Se ( V[i] == x )    Imprime ('Achei')         // se caiu em cima
Senão Se ( V[i-1] == x )  Imprime ('Achei')         // se passou por cima
Senão                     Imprime ('Não está lá')
```

Análise

Bom, do ponto de vista da notação O, o algoritmo executa em tempo $O(n)$.

Mas desse ponto de vista, a gente não vê o que está acontecendo.

Quer dizer, o laço da primeira etapa só passa por $n/2$ elementos.

E na segunda etapa, a gente só olha mais um ou dois.

Então, esse algoritmo já é duas vezes mais rápido do que a busca na lista desordenada — (*no pior caso*)

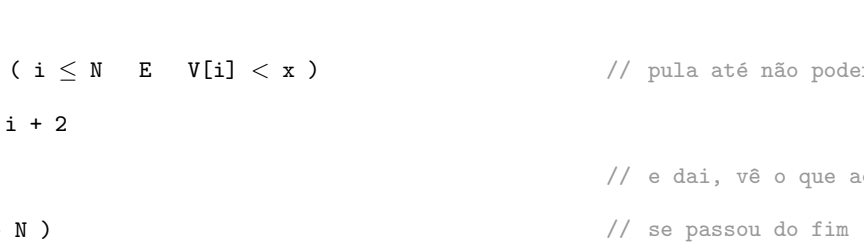
Legal.

Mas, a gente ainda pode fazer melhor.

Quer dizer, a ideia natural é aumentar o tamanho do pulo

- com pulos de tamanho 3, o tempo cai para $n/3$
- com pulos de tamanho 4, o tempo cai para $n/4$
- e por aí vai ...

Para entender o que está acontecendo, a gente considera o algoritmo que dá pulos de tamanho `k`.



A coisa fica assim

```
i = k
Enquanto ( i ≤ N   E   V[i] < x )          // pula até não poder mais
     i = i + k

Se ( i > N )   i = N                       // volta p/ ver o que aconteceu
achei = 0
Para j = i Até i - k
     Se ( V[i] == x )
          achei = 1;   Quebra
Se ( achei == 1 )     Imprime ('Achei')
Senão                 Imprime ('Não está lá')
```

Análise

Bom, é fácil ver que a primeira etapa examina no máximo

$$\frac{n}{k} \quad \text{elementos}$$

E que a segunda etapa examina no máximo

$$k \quad \text{elementos}$$

Logo, essa busca leva tempo

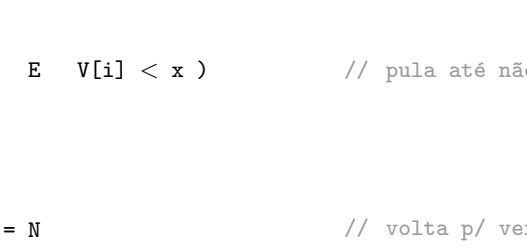
$$\frac{n}{k} \quad + \quad k$$

E agora, a gente pode perguntar

- *Qual é o valor de k que minimiza essa expressão ?*

Bom,

- quando o valor de k cresce, o primeiro termo diminui e o segundo termo aumenta
- quando o valor de k decresce, o primeiro termo aumenta e o segundo termo diminui



Logo, o valor ótimo de k é aquele que faz com que os dois termos fiquem iguais.

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

O que nos dá

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

Quer dizer, a busca pula-pula tem o seu melhor desempenho com pulos de tamanho $\sqrt{n}$.

Com essa configuração, a busca leva tempo

$$\frac{n}{\sqrt{n}} \quad + \quad \sqrt{n} \quad = \quad O(\sqrt{n})$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar 1000 deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

Não é legal?