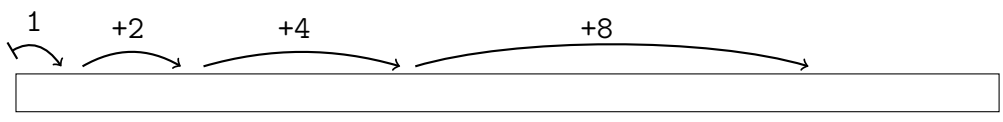
Mas, a gente ainda pode fazer melhor do que isso.

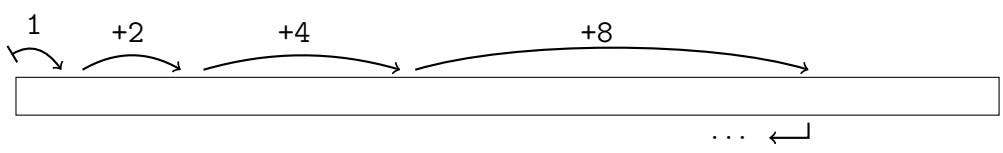
Quer dizer, a esperteza agora é a gente ir aumentando o tamanho do pulo



Dessa maneira, a gente avança bem mais rápido pela lista — (*examinando poucos elementos*)

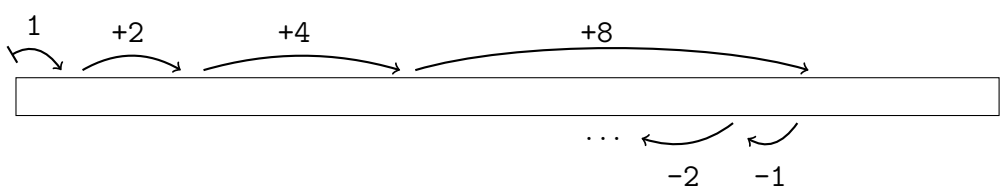
Mas o problema é que os pulos vão ficando cada vez maiores.

Daí que, na hora em que a gente passa do ponto, pode ter que voltar um montão



Mas, isso não é realmente um problema.

Porque a gente pode voltar pulando também



E se a gente passar do ponto outra vez, é só começar a pular outra vez.

Legal.

Agora é preciso implementar essa ideia.

E a melhor maneira de fazer isso é escrever uma função `Pula-pula()`.

Quer dizer, essa função vai implementar o procedimento padrão em uma faixa da lista.

E o programa principal vai utilizá-la para ficar indo e voltando.

Certo.

A função só tem dois resultados possíveis

- ou ela acha o elemento
- ou ela passa do ponto

E para indicar o que aconteceu, a gente vai utilizar 3 valores de retorno

`res, pos, pant` : resultado, posição onde a coisa parou, e posição anterior

Daí, a coisa fica assim

```
func Pula-pula ( x, L, inicio, fim, direcao )
    Se ( direção == 1 )                          // pulando para frente
        Se ( V[inicio] == x )
            Retorna  1, inicio, 0
        pant = inicio;   i = inicio+1;   pulo = 2
        Enquanto ( i ≤ fim   E   V[i] < k )
            pant = i;   i = i + pulo;   pulo = pulo * 2
        Se ( i > fim   OU   V[i] > k )
            Retorna  0, MIN(i,fim), pant         // não achou
        Senão
            Retorna  1, i, 0                     // achou
    Se ( direção == -1 )                         // pulando para trás
        (...  a mesma coisa ...  )
```

E o programa principal fica assim

```
inicio = 1;   fim = N;   direcao = 1
res = 0                                          // pulando para frente
Enquanto ( res == 0 )
    res,a,b  =  Pula-pula ( x, L, inicio, fim, direcao )
    Se ( direcao == 1 )
        fim = a;   inicio = b+1;   direção = direção * (-1)
        Se ( inicio > fim )     Quebra
    Se ( direcao == -1 )   (...  a mesma coisa ...)
```

Análise

Mas, e agora?

- *Qual é o tempo de execução desse algoritmo ?*

Bom, a função `Pula-pula()` é fácil.

Porque, a cada volta do laço, o pulo dobra de tamanho.

Logo, após $\log_2(n)$ voltas, o pulo é igual a n.

E ao dar um pulo desse tamanho, a gente cai fora da lista.

Logo, a função executa em tempo $O(\log\, n)$ — (*no pior caso*)

Certo.

Agora, a gente precisa estimar o número de voltas do laço no programa principal.

E aqui, a gente foca nas variáveis `inicio` e `fim`.

Quer dizer, quando o algoritmo começa, a distância entre elas é igual a n.

Mas daí, logo após a primeira chamada à função `Pula-pula()`, a distância é no máximo $n/2$

— (*porque?*)

E após a segunda chamada, a distância é no máximo $n/4$ — (*porque?*)

Quer dizer, a cada chamada a distância é reduzida à metade.

Logo, a coisa acaba em no máximo $\log_2(n)$ voltas.

Portanto, o algoritmo executa em tempo

$$\log_2(n) \quad \times \quad \log_2(n) \quad = \quad O(\log^2\, n)$$

Concretamente, em uma lista com 1.000.000 elementos, basta examinar $20^2 = 400$ deles (aprox.) para encontrar o número x — (*ou descobrir que ele não está lá*)

Não é legal?