Recursão

→ Uma função é recursiva se chama a si mesma.

→ Estamos projetando um algoritmo A para resolver um problema X; temos uma instância I de tamanho n que queremos resolver; percebemos que há uma instância I' contida em I de tamanho menor que n cuja solução nos ajuda a resolver I.
  ↳ como resolver I'?

→ Recursão é uma técnica muito poderosa de solução de problemas: a ideia é reduzir uma instância de um problema em instâncias menores cuja solução ajude a resolver a instância original.

→ Problema: busca de um elemento com chave k em um vetor.

	1	2	3	4	5	6	7	8	9
A	3	6	2	12	4	9	10	1	5

Note que uma instância desse problema contém várias instâncias menores: qualquer subvetor é um vetor e, portanto uma instância válida.

Recursão

→ Dado vetor $A[1..n]$ e chave de busca k, queremos resolver o problema "existe $x \in A[1..n]$ com chave k?"
  ↳ se soubermos resolver "existe $x \in A[1..n-1]$ com chave k?"
  ↳ e soubermos se $A[n].chave = k$
  então combinando essas informações resolvemos o problema original.

Ideia: k está em $A[n]$, ou em $A[1..n-1]$, ou não está em A

→ mas note que essa estratégia só faz sentido se $n \geq 1$.

Busca linear recursiva

```
BUSCALINEARRECURSIVA(A, n, k)
1 se n == 0 então
2     devolve −1
3 se A[n].chave == k então
4     devolve n
5 devolve BUSCALINEARRECURSIVA(A, n − 1, k)
```

→ Esse algoritmo funciona?

BuscaLinearRecursiva(A, 0, k)

BuscaLinearRecursiva(A, 1, k)

BuscaLinearRecursiva(A, 2, k)

BuscaLinearRecursiva(A, 3, k)

Corretude de algoritmos recursivos

```
BUSCALINEARRECURSIVA(A, n, k)
1 se n == 0 então
2     devolve −1
3 se A[n].chave == k então
4     devolve n
5 devolve BUSCALINEARRECURSIVA(A, n − 1, k)
```

Queremos provar, para qualquer n

$P(n)$ = "BuscaLinearRecursiva(A,n,k) devolve a posição do elemento em A com chave k se ele existir em $A[1..n]$, ou devolve -1."

→ Isso é exatamente o tipo de afirmação que a indução permite provar.

PASSO 1: Prove $P(0)$

PASSO 2: Para provar $P(n)$ para um $n \geq 1$, primeiro suponha que $P(t)$ vale, para $0 \leq t < n$, e então use isso para provar $P(n)$

Corretude da busca linear recursiva

```
BUSCALINEARRECURSIVA(A, n, k)
1 se n == 0 então
2     devolve −1
3 se A[n].chave == k então
4     devolve n
5 devolve BUSCALINEARRECURSIVA(A, n − 1, k)
```

P(n) = "BuscaLinearRecursiva(A,n,k) devolve a posição do elemento em A com chave k se ele existir em A[1..n], ou devolve −1."

CASO BASE: n = 0. O algoritmo diretamente devolve −1, que é a resposta esperada, pois A[1..0] é vazio. Logo, P(0) vale.

Considere algum n > 0.

HIPÓTESE: P(t) vale, para 0 ≤ t < n.

Quando o algoritmo recebe n > 0, ele verifica se A[n].chave = k (linha 3). Se forem iguais, devolve n e, portanto, P(n) vale nesse caso.

Se A[n].chave ≠ k, ele devolve o mesmo que BuscaLinearRecursiva(A,n-1,k). Mas, por hipótese, essa chamada corretamente detecta se "x ∈ A[1..n-1]". Como já sabíamos que A[n].chave ≠ k, concluímos que a chamada atual funciona.

Tempo de execução

```
BUSCALINEARRECURSIVA(A, n, k)
1 se n == 0 então                          Θ(1)
2     devolve −1
3 se A[n].chave == k então                 Θ(1)
4     devolve n
5 devolve BUSCALINEARRECURSIVA(A, n − 1, k)
```

→ só há uma chamada recursiva

→ são no máx. n chamadas recursivas, cada uma levando tempo constante

∴ O(n)

Busca em vetores ordenados

→ nossa melhor estratégia para resolver esse problema é a busca binária.

→ Dados $A[1..n]$ e k, comparamos k com $A[\lfloor n/2 \rfloor]$.chave.
  ↳ se iguais, encontramos o elemento procurado e devolvemos $\lfloor n/2 \rfloor$
  ↳ se $k < A[\lfloor n/2 \rfloor]$.chave, a única chance de haver um tal elemento é ele estar em $A[1..\lfloor n/2 \rfloor - 1]$
  ↳ se $k > A[\lfloor n/2 \rfloor]$.chave, a chance é procurar em $[\lfloor n/2 \rfloor + 1..n]$

→ Ainda queremos procurar por x, mas agora em um vetor menor.

Ideia: K está em $A[n/2]$, ou em $A[1..n/2-1]$, ou em $A[n/2+1]$ ou não está em A.

Busca em vetores ordenados

```
BUSCABINARIARECURSIVA(A, esq, dir, k)
1 se esq > dir então
2     devolve −1
3 meio = ⌊(esq + dir)/2⌋
4 se A[meio].chave == k então
5     devolve meio
6 senão se k < A[meio].chave então
7     devolve BUSCABINARIARECURSIVA(A, esq, meio − 1, k)
8 senão
9     devolve BUSCABINARIARECURSIVA(A, meio + 1, dir, k)
```

P(n) = "Sendo $n = dir - esq + 1$, então BuscaBinariaRecursiva(A, esq, dir, K) devolve -1 se não há elemento com chave K em $A[esq..dir]$ e devolve i caso contrário, com $A[i].chave = K$."

CASO BASE: $n = 0$. Para $dir - esq + 1 = 0$, $dir + 1 = esq$, o que significa $esq > dir$. Nesse caso, o algoritmo devolve -1, o que é o esperado, pois $A[esq..dir]$ é vazio e certamente não contém x.

Busca binária recursiva

```
BUSCABINARIARECURSIVA(A, esq, dir, k)
1 se esq > dir então
2 |  devolve −1
3 meio = ⌊(esq + dir)/2⌋
4 se A[meio].chave == k então
5 |  devolve meio
6 senão se k < A[meio].chave então
7 |  devolve BUSCABINARIARECURSIVA(A, esq, meio − 1, k)
8 senão
9 |  devolve BUSCABINARIARECURSIVA(A, meio + 1, dir, k)
```

Queremos provar P(n), com $n > 0$.
HIPÓTESE: P(t) vale para todo $0 \leq t < n$.
Quando $n > 0$, $dir - esq + 1 > 0$, o que implica $dir + 1 > esq$, ou $dir \geq esq$.
Então o que o algoritmo faz é calcular meio e comparar a chave dessa posição com K.
Se A[meio].chave = K, ele devolve meio e, logo, P(n) vale nesse caso.
Se A[meio].chave > K, ele chama BuscaBinariaRec(A, esq, meio-1, k) e devolve o mesmo que ela. Note que

$$meio - 1 - esq + 1 = \left\lfloor \frac{esq+dir}{2} \right\rfloor - esq \leq \frac{esq+dir}{2} - esq = \frac{dir-esq}{2} = \frac{n-1}{2}$$

e que $(n-1)/2 < n$ sempre que $n > -1$. Então, por hipótese, P(meio - esq) vale e a chamada recursiva detecta se A[esq..meio-1] contém o elemento procurado. Como o vetor tem as chaves ordenadas e $K < A[meio].chave$, então concluímos que, nesse caso, a chamada atual corretamente detecta se A[esq..dir] contém o elemento procurado. Logo, P(n) vale nesse caso.
Se A[meio].chave < K, a análise é similar e concluímos que P(n) sempre vale.

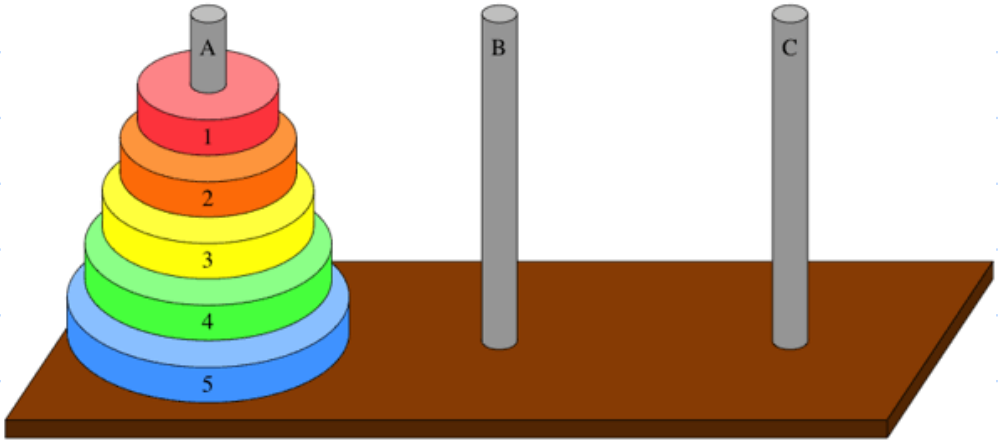
Tempo de execução

```
BUSCABINARIARECURSIVA(A, esq, dir, k)
1 se esq > dir então
2 |  devolve −1
3 meio = ⌊(esq + dir)/2⌋
4 se A[meio].chave == k então
5 |  devolve meio
6 senão se k < A[meio].chave então
7 |  devolve BUSCABINARIARECURSIVA(A, esq, meio − 1, k)
8 senão
9 |  devolve BUSCABINARIARECURSIVA(A, meio + 1, dir, k)
```

→ note que há apenas uma chamada recursiva

→ são no máximo $\lg n$ chamadas, cada uma com tempo $\Theta(1)$
$\therefore O(\lg n)$

Torres de Hanói



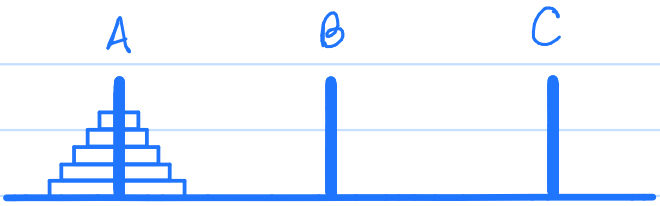
Objetivo: mover todos os discos da estaca A para a estaca C

↳ apenas um disco pode ser movido por vez

↳ um disco maior não pode ser colocado sobre um menor

Torres de Hanói

```
HANOi (n, orig, dest, aux)
    Se n > 0
        HANOi (n-1, orig, aux, dest)
        move n de orig para dest
        HANOi (n-1, aux, dest, orig)
```



Primeira chamada: HANOi (n, A, C, B)

Torres de Hanoi: corretude

$P(n)$ = "HANOI$(n, P1, P2, P3)$ move n discos de $P1$ para $P2$ sem que um disco maior fique sobre outro menor."

```
HANOI (n, orig, dest, aux)
    Se n > 0
        HANOI (n-1, orig, aux, dest)
        move n de orig para dest
        HANOI (n-1, aux, dest, orig)
```

$P(0)$ vale trivialmente.

Considere $n > 0$ e suponha que $P(k)$ vale para todo $0 \leq k < n$.

O algoritmo chama HANOI$(n-1, orig, aux, dest)$, e como $n-1 < n$, por hipótese sabemos que isso move $n-1$ discos de orig para aux sem que discos maiores fiquem sobre menores. Então movemos o disco n para dest, que também não quebra essa regra. Por fim, HANOI$(n-1, aux, dest, orig)$, por hipótese, corretamente move os $n-1$ discos de aux para dest sem quebrar a regra dos tamanhos.

CQD

Torres de Hanoi: tempo

```
HANOI (n, orig, dest, aux)
    Se n > 0
        HANOI (n-1, orig, aux, dest)
        move n de orig para dest
        HANOI (n-1, aux, dest, orig)
```

Recorrências

→ **Recorrências** são expressões que descrevem uma função em termos dela mesma.

Ex: $F_n = F_{n-1} + F_{n-2}$ se $n > 2$ $\quad (F_1 = F_2 = 1)$

→ A forma mais simples de descrever o tempo de execução de um algoritmo recursivo é usando recorrências.

Recorrências descrevendo tempo

```
BUSCALINEARRECURSIVA(A, n, k)
1 se n == 0 então
2 |  devolve −1
3 se A[n].chave == k então
4 |  devolve n
5 devolve BUSCALINEARRECURSIVA(A, n − 1, k)
```

$T(0) = \Theta(1)$

$T(n) \leq T(n-1) + \Theta(1)$

```
BUSCABINARIARECURSIVA(A, esq, dir, k)
1 se esq > dir então
2 |  devolve −1
3 meio = ⌊(esq + dir)/2⌋
4 se A[meio].chave == k então
5 |  devolve meio
6 senão se k < A[meio].chave então
7 |  devolve BUSCABINARIARECURSIVA(A, esq, meio − 1, k)
8 senão
9 |  devolve BUSCABINARIARECURSIVA(A, meio + 1, dir, k)
```

$T(0) = \Theta(1)$

$T(n) \leq T(\lceil n/2 \rceil) + \Theta(1)$

```
HANOI (n, orig, dest, aux)
    se n > 0
        HANOI (n-1, orig, aux, dest)
        move n de orig para dest
        HANOI (n-1, aux, dest, orig)
```

$T(0) = \Theta(1)$

$T(n) = 2T(n-1) + \Theta(1)$

Recorrências × Tempo

→ Apesar de recorrências descreverem tempo de algoritmos recursivos de forma muito simples, as expressões

$$T(n) \leq T(n-1) + \Theta(1)$$
$$T(n) \leq T(\lceil \tfrac{n}{2} \rceil) + \Theta(1)$$
$$T(n) = 2T(n-1) + \Theta(1)$$

não nos dão informações diretas, como $O(n^2)$, $\Theta(n \log_2 n)$, $O(2^n)$...

→ Precisamos então resolver as recorrências.

↳ Substituição ↳ Árvore de recursão

↳ Iteração ↳ Mestre

→ Convenções:

i) $T(n) = \Theta(1) \quad \forall\, n \leq n_0$ para n_0 "conveniente" - caso base

ii) Pisos e tetos serão desconsiderados

$$T(n) \leq T(\tfrac{n}{2}) + \Theta(1)$$

iii) n será uma potência conveniente:

$\lceil n/a \rceil$ ou $\lfloor n/a \rfloor \Rightarrow n/a$ e $n = a^k$ para algum k

Exercício em sala

Calcular o tempo dos algoritmos a seguir

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
 1  n = A.rows
 2  let C be a new n × n matrix
 3  if n == 1
 4       c11 = a11 · b11
 5  else partition A, B, and C as in equations (4.9)
 6       C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
               + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
 7       C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
               + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
 8       C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
               + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
 9       C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
               + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Algoritmo 20.1: MULTIPLICAINTEIROS(x, y)

```
1 n = IGUALATAM(x, y)
2 se n ≤ 2 então
3 |  devolve xy
4 Seja x = 10^⌈n/2⌉ a + b e y = 10^⌈n/2⌉ c + d, onde a e c têm ⌊n/2⌋ dígitos cada e b
    e d têm ⌈n/2⌉ dígitos cada
5 p1 = MULTIPLICAINTEIROS(a, c)
6 p2 = MULTIPLICAINTEIROS(a, d)
7 p3 = MULTIPLICAINTEIROS(b, c)
8 p4 = MULTIPLICAINTEIROS(b, d)
9 devolve 10^(2⌈n/2⌉) p1 + 10^⌈n/2⌉ (p2 + p3) + p4
```

Algoritmo 20.2: KARATSUBA(x, y, n)

```
1 n = IGUALATAM(x, y)
2 se n ≤ 2 então
3 |  devolve xy
4 Seja x = 10^⌈n/2⌉ a + b e y = 10^⌈n/2⌉ c + d, onde a e c têm ⌊n/2⌋ dígitos cada e b
    e d têm ⌈n/2⌉ dígitos cada
5 p1 = KARATSUBA(a, c)
6 p2 = KARATSUBA(b, d)
7 p3 = KARATSUBA(a + b, c + d)
8 devolve 10^(2⌈n/2⌉) p1 + 10^⌈n/2⌉ (p3 − p1 − p2) + p2
```