Algoritmos Recursivos

Algoritmos Recursivos

Resolvendo um problema com recursão

- Instâncias pequenas: resolvemos diretamente

- Instâncias grandes:
  - construímos uma instância menor do mesmo problema.
  - Resolvemos essas subinstâncias recursivamente.
  - Usamos as soluções dos subproblemas para produzir uma solução para o problema original.

Projeto de Algoritmos Recursivos

- Encontrar e definir um subproblema adequado.
- Supor que temos um algoritmo A que resolve instâncias menores do problema.
- Concluir que A tbm resolve a instância original

O subproblema escolhido:
- não precisa ser o problema original
- costuma ser uma variante mais forte

Recursão e Indução

É fácil obter um algoritmo recursivo da prova indutiva.

- caso base: Instâncias pequenas
- caso geral: Instâncias grandes
  - a chamada recursiva corresponde ao uso da hipótese de indução.
  - a construção da solução corresponde as ações feitas no passo da indução.

Calculando o valor de um polinômio

Problema do Valor do Polinômio

Entrada: Um vetor $A[0..n]$ de $n+1$ números reais e um real x.

Saída: O valor de $P_A(x)$

$$P_A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0,$$

onde $a_i = A[i]$ para todo $i = 0, \ldots, n$

Ex:

$A =$

0	1	2	3
4	0	-3	2

e $x = 4$

$$P_3(4) = 2 \cdot (4)^3 - 3(4)^2 + 0 \cdot (4)^1 + 4$$
$$= 96 - 48 + 0 + 4 = 52$$

Notação

- Sejam $A[0..n]$ e $k \in \mathbb{N} \cup \{0\}$ tal que $0 \leq k \leq n$

$$P_A(x) = A[n] \times x^n + A[n-1] \times x^{n-1} + \cdots + A[1] \times x + A[0]$$

- Quando A for claro do contexto, tbm usamos

$$P_k(x) = A[k] \times x^k + A[k-1] \times x^{k-1} + \cdots + A[1]x + A[0]$$

$$P_A(x) = 5x^4 + \overbrace{6x^3 + 1x^2 + 7x + 10}^{P_3(x)}$$

	10	7	1	6	5
A =	0	1	2	3	4

Projeto de Algoritmos por Indução

- Você sabe fazer o caso pequeno (Base)

- $n = 0 \Rightarrow P_A(x) = a_0$
- Basta devolver $A[0]$

- Suponha que você tem uma instância de tamanho n

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

Se você souber calcular o valor de um polinômio de tamanho $n-1$, você consegue resolver o problema?

Assuma que você sabe calcular $P_{n-1}(x)$

→ por mágica

Projeto de Algoritmos por Indução

Hipotese de indução

Dado um vetor $A'[0..n-1]$ de n números reais e um número real x, sabemos calcular o valor de

$$P_{A'}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

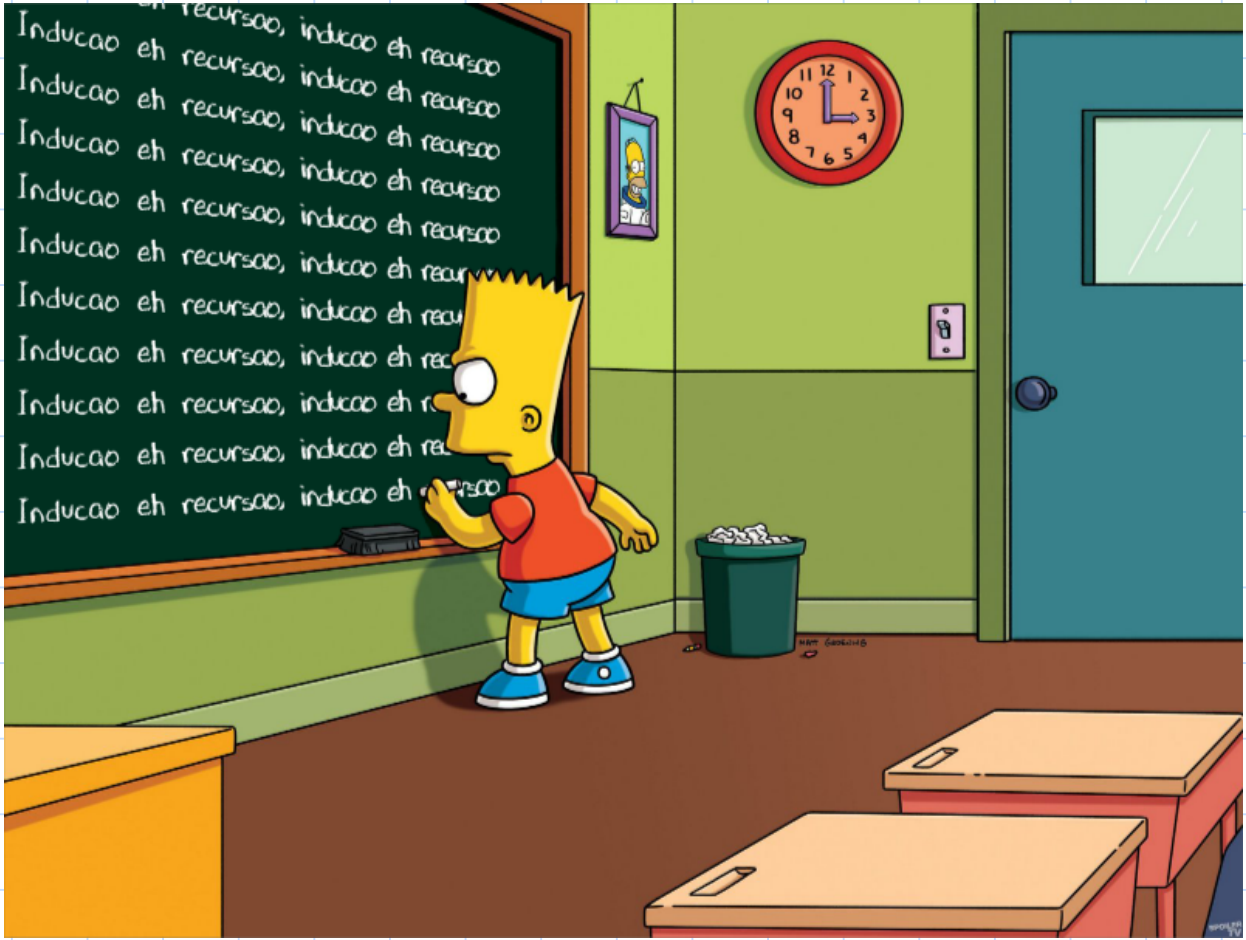
→ no código o valor dado pela hipótese de indução será dado pela chamada recursive.

Projeto de Algoritmos por Indução

Caso Geral ($n > 0$)

$$P_n(x) = a_n x^n + \overbrace{a_{n-1} x^{n-1} + \cdots + a_1 x^1 + a_0}^{P_{n-1}(x)}$$
$$= a_n x^n + P_{n-1}(x)$$

- Calculamos $P_{n-1}(x)$ recursivamente (por H.I. sabemos fazer)
- Somamos $a_n x^n$ ao resultado obtido.


recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao
Inducao eh recursao, inducao eh recursao

Projeto de Algoritmos por Indução

Vamos considerar apenas os coeficientes do subvetor A[0..n]

```
Calculo-Polinômio(A[0..p], n, x)
1   Se n == 0
2      y = A[0]
3   Senão
4      y' = Cálculo-Polinômio(A, n-1, x)
5      x n = 1
6      para  i = 1 até n
7         xn = xn . x
8      y = y' + A[n] . xn
9   devolva y
```

Correção de Algoritmos Recursivos

Correção de Algoritmos Recursivos

Como fazer?

Provemos por indução que o algoritmo está correto.

Teo. Cálculo-Polinômio (A, n, x) resolve corretamente o Problema do Valor do Polinômio.

Demonstração

- A prova segue por indução em n

Base $n = 0$

- Quando $n = 0$, $P_A(x) = a_0$
- Como $n = 0$, temos que o teste da linha 1 da verdadeiro. O algoritmo faz

$$y = A[0]$$

na linha 2

- Então o algoritmo corretamente retorna $y = A[0] = a_0$ na linha 9.

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2    y = A[0]
3 Senão
4    y' = Cálculo-Polinômio(A, n-1, x)
5    xn = 1
6    para i=1 até n
7       xn = xn · x
8    y = y' + A[n] · xn
9 devolva y
```

Passo ($n \geq 1$)

- Agora suponha que $n \geq 1$.
- Então o teste da linha 1 falha e o algoritmo executa a linha 4, fazendo

$$y' = \text{Cálculo-Polinômio}(A, n-1, x)$$

- Nessa chamada A tem $n-1$ elemento, então, por Hipótese de indução, Cálculo-Polinômio $(A, n-1, x)$ resolve corretamente o problema e, assim

$$y' = P_{n-1}(x)$$

- Note que $P_n(x) = a_n x^n + P_{n-1}(x)$
$$= A[n] \times x^n + y'$$

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2    y = A[0]
3 Senão
4    y' = Cálculo-Polinômio(A, n-1, x)
5    xn = 1
6    para i = 1 até n
7       xn = xn . x
8    y = y' + A[n] . xn
9 devolva y
```

- Então o algoritmo executa a linha 5, fazendo

$$xn = 1$$

- Na sequência, o código atinge a linha 6. A invariante desse laço diz que

$Q(t)$ = " antes da t-ésima iteração começar, vale que

Exercício

ⓐ $i = t$

ⓑ $xn = x^{t-1}$ "

- Por ⓐ, sabemos que o laço termina após n - iterações
- Por $Q(n+1)$, temos que

$$xn = x^n$$

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2     y = A[0]
3 Senão
4     y' = Cálculo-Polinômio(A, n-1, x)
5     xn = 1
6     para i = 1 até n
7         xn = xn · x
8     y = y' + A[n] · xn
9 devolva y
```

- Na sequência o algoritmo executa a linha 8 fazendo

$$y = y' + A[n] \cdot x^n$$

$$= P_{n-1}(x) + A[n] \cdot x^n$$

$$= P_n(x)$$

- Por fim, o algoritmo retorna

$$y = P_n(x),$$

que é o valor correto

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2    y = A[0]
3 Senão
4    y' = Cálculo-Polinômio(A, n-1, x)
5    xn = 1
6    para i = 1 até n
7       xn = xn · x
8    y = y' + A[n] · xn
9 devolva y
```

Análise de Tempo de Execução

```
Calculo-Polinômio(A[0..p], n, x)
1    Se n == 0
2       y = A[0]
3    Senão
4       y' = Cálculo-Polinômio(A, n-1, x)
5       xn = 1
6       para i=1 até n
7          xn = xn . x
8       y = y' + A[n] . xn
9    devolva y
```

Análise de Tempo de Execução

```
Calculo-Polinômio(A[0..p], n, x)
1    Se n==0
2       y = A[0]                    } A = Θ(1)
3    Senão
4       y' = Cálculo-Polinômio(A, n-1, x)
5       xn = 1
6       para i=1 até n
7          xn = xn.x
8       y = y' + A[n] . xn
9    devolva y                      } B = Θ(1)
```

Caso: $n = 0$

$T(n) = A + B$

$= \Theta(1) + \Theta(1)$

$= \Theta(1)$

Análise de Tempo de Execução

Caso: $n > 0$

```
Calculo-Polinômio(A[0..p], n, x)
1    Se n == 0                          ] A = Θ(1)
2       y = A[0]
3    Senão
4       y' = Cálculo-Polinômio(A, n-1, x)   ] B = T(n-1)
5       xn = 1                          ] C = Θ(1)
6       para i = 1 até n                ] D = Θ(n)
7          xn = xn . x
8       y = y' + A[n] . xn              ] E = Θ(1)
9    devolva y
```

$$T(n) = A + B + C + D + E$$
$$= \Theta(1) + T(n-1) + \Theta(1) + \Theta(n) + \Theta(1)$$
$$= T(n-1) + \Theta(1)$$

Análise de Tempo de Execução

```
Calculo-Polinômio(A[0..p], n, x)
1    Se n == 0
2        y = A[0]
3    Senão
4        y' = Cálculo-Polinômio(A, n-1, x)
5        xn = 1
6        para i = 1 até n
7            xn = xn . x
8        y = y' + A[n] . xn
9    devolva y
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{se } n \geq 1 \end{cases}$$

Análise de Tempo de Execução

Tempo de Execução de Cálculo-Polinômio

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{se } n \geq 1 \end{cases}$$

Resolvendo $T(n)$, temos que

$$T(n) = \Theta(n^2).$$

Análise de Tempo de Execução

- Quando $n = 0$, temos que apenas as linhas 1-2, e 9 são executadas. Todas essas linhas tem tempo constante e são executadas apenas uma vez. Portanto gastamos

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

- Quando $n > 0$, temos que as linhas 1, 4-9 são executadas.
- As linhas 1, 5, 8, 9 executam em tempo constante e são executadas apenas uma vez. Portanto, o algoritmo gasta $\Theta(1)$ com esse trecho.

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2     y = A[0]
3 Senão
4     y' = Cálculo-Polinômio(A, n-1, x)
5     xn = 1
6     para i = 1 até n
7         xn = xn · x
8     y = y' + A[n] · xn
9 devolva y
```

Análise de Tempo de Execução

- A linha 4 leva tempo $\Theta(1)$ para invocar a chamada recursiva a Cálculo-Polinômio $(A, n-1, x)$. Além disso, gastamos $T(n-1)$ dentro da chamada recursiva.

- O custo p/ executar as instruções das linhas 6-7 uma única vez é constante. Essas instruções são executadas $\Theta(n)$ vezes. Logo o custo total com esse trecho é $\Theta(n)$

- Assim, temos que $T(n) = T(n-1) + \Theta(n)$ para $n \geq 1$.

```
Calculo-Polinômio(A[0..p], n, x)
1 Se n == 0
2    y = A[0]
3 Senão
4    y' = Cálculo-Polinômio(A, n-1, x)
5    xn = 1
6    para i = 1 até n
7       xn = xn . x
8    y = y' + A[n] . xn
9 devolva y
```

Análise de Tempo de Execução

- Por consequinte, temos

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(n) & \text{se } n > 0 \end{cases}$$

- Agora, vemos prova que $T(n)$ e $O(n^2)$.
- Para isso vamos usar o método da substituição

$$T(n) \leq c\,n^2$$

$$\vdots$$ Exercício.

- Portanto Cálcula-Polinômio é $O(n^2)$ □

Melhorando Nosso Código

```
Calculo-Polinômio(A[0..p], n, x)
1   Se n==0
2      y = A[0]
3   Senão
4      y' = Cálculo-Polinômio(A, n-1, x)
5      xn = 1
6      para i=1 até n
7         xn = xn · x
8      y = y' + A[n] · xn
9   devolva y
```

- É possível fazer algo melhor do que esse código
- Note que para calcular $a_{n-1} \times x^{n-1}$ precisamos saber quem é x^{n-1}. Saber esse valor facilita o trabalho de calcular x^n

$$x^n = x^{n-1} \cdot x$$

Melhorando Nosso Código

- Nosso algoritmo anterior recalcula potências de x
- Podemos reaproveitar o cálculo de x^{n-1}
- Para isso, fortalecemos a hipótese de indução

Hipótese Marombada 

Dado um vetor $A'[0..n-1]$ de n números reais e um número real x, sabemos calcular o valor de

$$P_{A'}(x) = a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \quad \underline{\text{e } x^{n-1}}$$

⇒ Podemos fazer hipóteses mais fortes sobre a recursão

⇒ mas agora tbm precisamos concluir que sabemos calcular x^{n-1}

```
Calculo-Polinômio2(A[0..p], n, x)
1    Se n==0
2       y = A[0]
3       xn = 1
4    Senão
5       y', x' = Cálculo-Polinômio2(A, n-1, x)
6       xn = x' . x
7       y = y' + A[n] . xn
8    devolva y, xn
```

Correção do Algoritmo Cálculo-Polinômio2

Teo. Cálculo-Polinômio2 recebe um vetor $A[0..p]$ com n reais e um real x e retorna

$$P_A(x) \quad e \quad x^n$$

Exercício

Análise do Algoritmo Cálculo-Polinômio2

```
Calculo-Polinômio2(A[0..p], n, x)
1    Se n==0
2        y = A[0]
3        xn = 1
4    Senão
5        y', x' = Cálculo-Polinômio2(A, n-1, x)
6        xn = x' . x
7        y = y' + A[n] . xn
8    devolva y, xn
```

Análise do Algoritmo Cálculo-Polinômio2

Caso: $n = 0$

```
Calculo-Polinômio2(A[0..p], n, x)
1    Se n==0
2        y = A[0]
3        xn = 1
4    Senão
5        y', x' = Cálculo-Polinômio2(A, n-1, x)
6        xn = x' · x
7        y = y' + A[n] · xn
8    devolva y, xn
```

Linhas 1–3: $A = \Theta(1)$

Linhas 8: $B = \Theta(1)$

$$T(n) = A + B$$
$$= \Theta(1) + \Theta(1)$$
$$= \Theta(1)$$

Análise do Algoritmo Cálculo-Polinômio2

Caso: $n > 1$

```
Calculo-Polinômio2(A[0..p], n, x)
1   Se n == 0                                   ] A = Θ(1)
2       y = A[0]
3       xn = 1
4   Senão
5       y', x' = Cálculo-Polinômio2(A, n-1, x)  ] C = T(n-1)
6       xn = x' · x
7       y = y' + A[n] · xn                      ] B = Θ(1)
8   devolva y, xn
```

$T(n) = A + B + C$

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

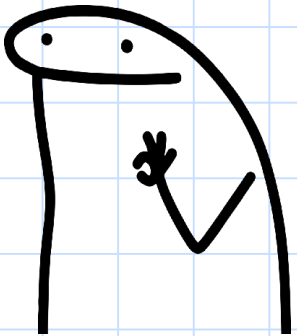
$= T(n-1) + \Theta(1)$

Análise do Algoritmo Cálculo-Polinômio2

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(n-1) + \Theta(1) & \text{se } n \geq 1 \end{cases}$$

Resolvendo essa recorrência, temos que

$$T(n) = \Theta(n)$$



- Talvez você tenha percebido que n nesse problema não é o tamanho da entrada, mas sim o grau do polinômio. Note que o tamanho da entrada é exatamente uma unidade maior do que o grau do polinômio

$P(x) = a_0 \Rightarrow$ Vetor A tem 1 elemento

$P(x) = a_2 x^2 + a_1 x + a_0 \Rightarrow$ Vetor A tem 3 elemento

(2) → Valor de n

- Sempre escrevemos o tempo do algoritmo em função do tamanho da entrada. No entanto temos, neste caso, a função do tempo em relação ao grau do polinômio. Então, vamos corrigir isso

- Fiz isso porque o resultado final é o mesmo (neste caso) e a análise fica menos confusa de seguir (porque ñ precisamos ficar somando 1 ao grau). Agora vou argumentar que eu não enganei vocês... Para isso, vou exemplificar usando o Calcula-Polinômio.

- Seja A um vetor que representa um polinômio de grau n
- Seja m o tamanho dessa entrada, ou seja, o tamanho do vetor A. Assim, temos

$$m = n + 1$$

- Da nossa análise concluímos que Cálculo-Polinômio roda em $\Theta(m^2)$. Se eu não menti para vocês, ñ tendo diferença fazer a análise com grau neste caso,

deveríamos ser capazes de concluir que

Cálculo-Polinômio é $\Theta(m^2)$.

Seja $F(m)$ a função que retorna o grau do polinômio de tamanho m, ou seja, $F(m) = m - 1$. Note que o tempo de execução em função do tamanho da entrada agora pode ser escrito por

$T(F(m))$ ← $T(n)$ retorna o tempo de execução em função do grau do polinômio.

- Como $T(n) \in O(n^2)$, temos que

$$T(F(m)) \leq C \cdot F(m)^2 \quad \forall\ m \geq m_0$$
$$= C \cdot (m-1)^2$$
$$= C\left[m^2 - 2m + 1\right]$$
$$= C\,m^2 - 2Cm + C$$
$$= \Theta(m^2)$$ ← protinho

Poderíamos definir $T^*(m) := T(F(m))$ pl representar que $T^*(m)$ é o tempo de execução do algoritmo CálculoPolinômio em função do tamanho da entrada

Problema de Ordenação

Entrada: Um vetor $A[1..n]$ de números

Saída: Uma permutação dos elementos de A tal que $A[1] \leq A[2] \leq \cdots \leq A[n]$

→ Vimos que o insertio-Sort resolve esse problema em $O(n^2)$.

→ Vamos tentar fazer um algoritmo recursivo para esse problema.

Hipótese de Indução

Sabemos resolver o problema da ordenação para um vetor com menos de n elementos.

Caso base (n=1)

- $A[1..1] = A[1]$ está trivialmente ordenado

Caso geral ($n \geq 2$)

- Recursivamente ordenamos $A[1..n-1]$
- Reorganizamos A para colocar $A[n]$ na posição correta

Caso geral ($n \geq 2$)

- Recursivamente ordenamos $A[1..n-1]$
- Reorganizamos A para colocar $A[n]$ na posição correta

```
Ordena(A[1..n], n)
    Se n == 1
        retorna
    Ordena(A, n-1)
    atual ← A[n]
    j ← n-1
    Enquanto j>0 e A[j] > atual
        A[j+1] = A[j]
        j = j-1
    A[j+1] = atual
```

Caso geral ($n \geq 2$)

- Recursivamente ordenamos $A[1..n-1]$
- Reorganizamos A para colocar $A[n]$ na posição correta

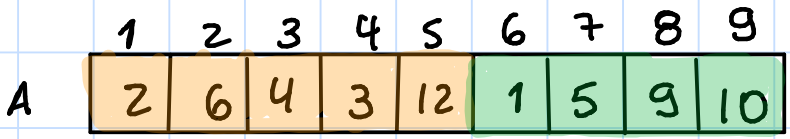
```
Ordena(A[1..n], n)
    Se n == 1
        retorna
    Ordena(A, n-1)
    atual ← A[n]
    j ← n-1
    Enquanto j>0 e A[j] > atual
        A[j+1] = A[j]
        j = j-1
    A[j+1] = atual
```

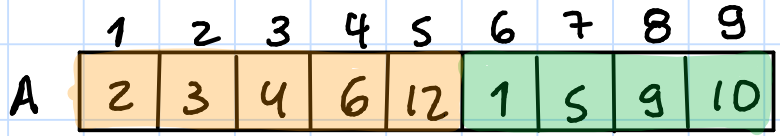
Esse algoritmo é um "InsertionSort" no qual o laço externo foi trocado por uma recursão.

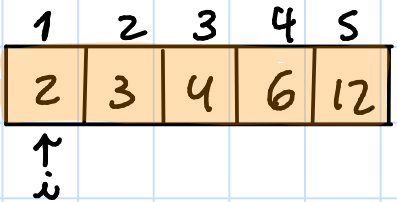
Caso geral ($n \geq 2$)

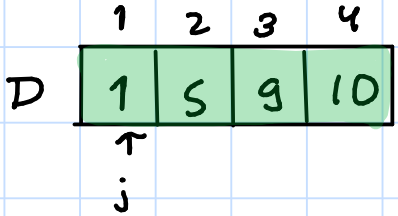
(Tente outra vez)

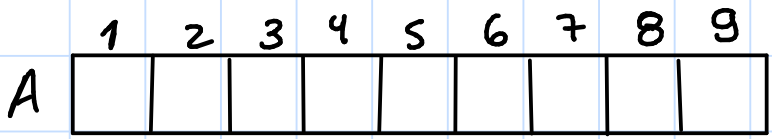
- Recursivamente ordenamos $A[1..\lceil \frac{n}{2} \rceil]$ e $A[\lceil \frac{n}{2} \rceil + 1..n]$.
- Combinamos os dois subvetores ordenados.







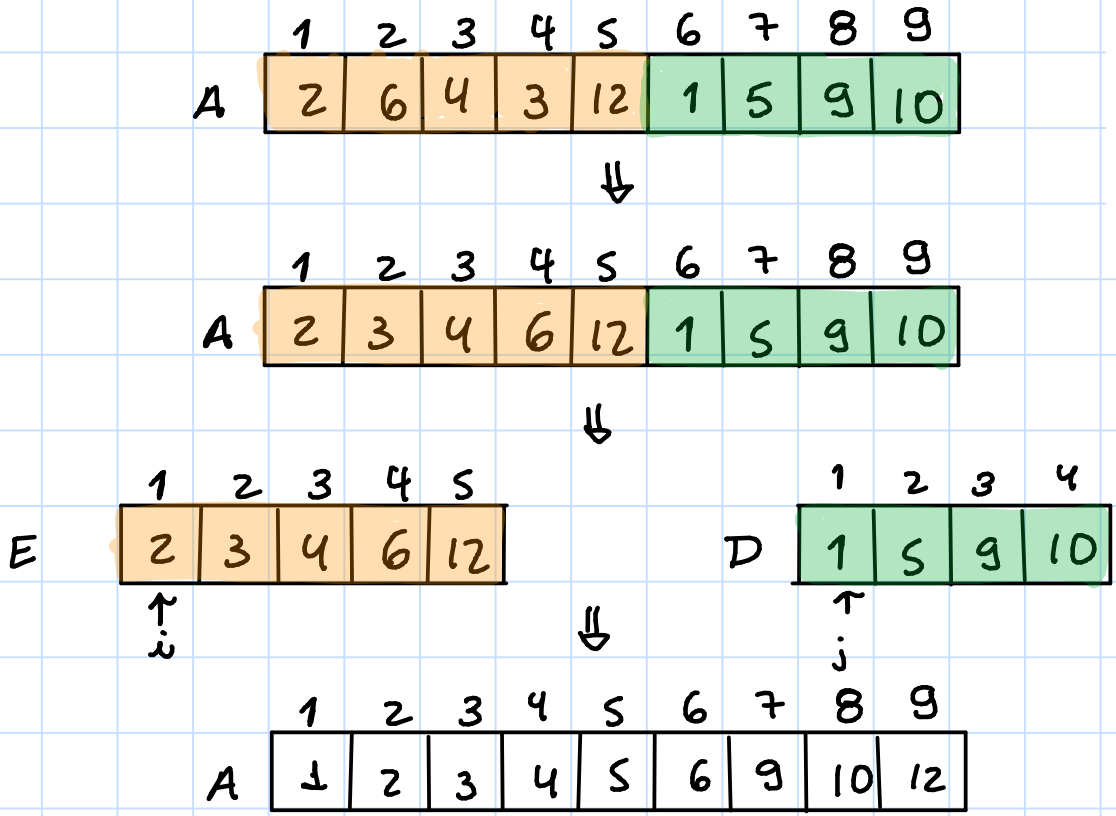




Caso geral ($n \geq 2$)

(Tente outra vez)

- Recursivamente ordenamos $A[1..\lceil \frac{n}{2} \rceil]$ e $A[\lceil \frac{n}{2} \rceil + 1..n]$.
- Combinamos os dois subvetores ordenados.



Caso geral ($n \geq 2$)

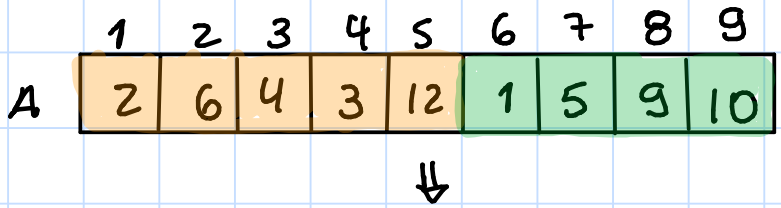
(Tente outra vez)

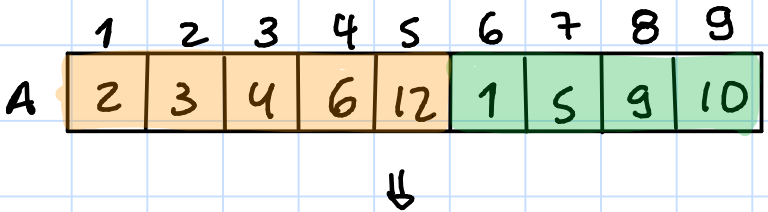
- Recursivamente ordenamos $A[1..\lceil \frac{n}{2} \rceil]$ e $A[\lceil \frac{n}{2} \rceil + 1..n]$.
- Combinamos os dois subvetores ordenados.

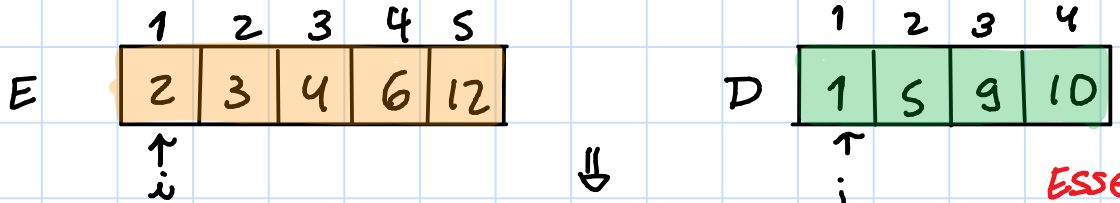
Caso geral ($n \geq 2$)

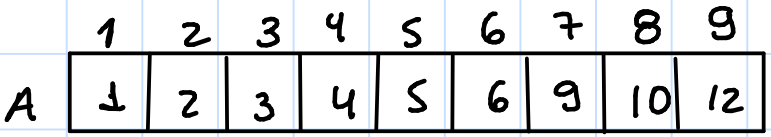
(Tente outra vez)

- Recursivamente ordenamos $A[1..\lceil \frac{n}{2} \rceil]$ e $A[\lceil \frac{n}{2} \rceil + 1..n]$.
- Combinamos os dois subvetores ordenados.









Esse algoritmo é chamado de Merge-Sort

```
Merge-Sort (A[1..p], ini, fim)
1    Se   ini < fim
2         meio = ⌊(ini + fim) / 2⌋
3         Merge-Sort (A, ini, meio)
4         Merge-Sort (A, meio+1, fim)
5         Combina (A, ini, meio, fim)
```

```
Combina ( A[1..p], ini, meio, fim)
1    n1 = meio - ini + 1
2    n2 = fim - meio
3    Sejam E[1.. n1+1] e D[1.. n2+1] dois vetores de números
4    Para i = 1 até n1
5        E[i] = A[ini + i - 1]
6    Para i = 1 até n2
7        D[i] = A[meio + i]
8    E[n1 + 1] = ∞
9    D[n2 + 1] = ∞
10   i = 1
11   j = 1
12   Para k = 1 até n
13       se E[i] < D[j]
14           A[k] = E[i]
15           i = i + 1
16       Senão
17           A[k] = D[j]
18           j = j + 1
```

Seja $n = fim - ini + 1$

```
Combina ( A[1..p], ini, meio, fim)
1    n1 = meio - ini + 1
2    n2 = fim - meio
3    Sejam E[1..n1+1] e D[1..n2+1] dois vetores de números
4    Para i = 1 até n1
5        E[i] = A[ini + i - 1]
6    Para i = 1 até n2
7        D[i] = A[meio + i]
8    E[n1 + 1] = ∞
9    D[n2 + 1] = ∞
10   i = 1
11   j = 1
12   Para k = 1 até n
13       se E[i] ≤ D[j]
14           A[k] = E[i]
15           i = i + 1
         Senão
16           A[k] = D[j]
17           j = j + 1
```

não contabilizamos alocação de memória. A memória simplesmente existe

$A = \Theta(1)$

$B = \Theta(n)$

$C = \Theta(1)$

$D = \Theta(n)$

Assim o tempo de combina é:

$$T(n) = A + B + C + D$$
$$= \Theta(1) + \Theta(n) + \Theta(1) + \Theta(n)$$
$$= \Theta(n)$$

Análise de Tempo do Alg. Combina

- Seja $n = fim - ini + 1$
- As linhas 1-2 e 10-11 levam tempo constante para executar e são executadas uma única vez. Logo o algoritmo gasta $\Theta(1)$ nesse trecho.

- As linhas 4-5 levam tempo constante para executar uma única vez e são executadas $\Theta(n_1)$ vezes. Já as linhas 6-7 levam tempo constante para executar uma única vez, mas são executadas $\Theta(n_2)$ vezes. Assim, o tempo total gasto pelo algoritmo com as linhas 4-7 é

$$\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2)$$
$$= \Theta(meio - ini + 1 + fim - meio) = \Theta(fim - ini + 1) = \Theta(n)$$

Análise de Tempo do Alg. Combina

- O laço da linha 12 (linhas 12-17) leva tempo constante para executar uma única vez. Como esse laço é executado $\Theta(fim - ini) = \Theta(n)$, temos que o custo com esse trecho é $\Theta(n)$

- Portanto, o custo total de execução desse algoritmo é $\Theta(n)$.

```
Merge-Sort (A[1..p], ini, fim)
1   Se   ini < fim                                  ] A = Θ(1)
2        meio = ⌊(ini + fim) / 2⌋                   ]
3        Merge-Sort (A, ini, meio)                  ] B = T(⌈n/2⌉)
4        Merge-Sort (A, meio+1, fim)                ] C = T(⌊n/2⌋)
5        Combina (A, ini, meio, fim)                ] D = Θ(n)
```

$$n = fim - ini + 1 \iff n + 2ini = fim - ini + 1 + 2ini$$
$$\iff n + 2ini - 1 = fim + 1 + ini - 1$$
$$\iff n + 2ini - 1 = fim + ini$$

$$meio = \left\lfloor \frac{ini + fim}{2} \right\rfloor = \left\lfloor \frac{n + 2ini - 1}{2} \right\rfloor = ini + \left\lfloor \frac{n-1}{2} \right\rfloor$$

```
Merge-Sort (A[1..p], ini, tim)
1    Se   ini < tim                                  ] A = Θ(1)
2         meio = ⌊(ini + tim) / 2⌋                   ]
3         Merge-Sort (A, ini, meio)                  ] B = T(⌈n/2⌉)
4         Merge-Sort (A, meio+1, tim)                ] C = T(⌊n/2⌋)
5         Combina (A, ini, meio, tim)                ] D = Θ(n)
```

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

$$\#\text{ elementos em } A[ini..meio] = meio - ini + 1$$

$$= \left(ini + \left\lfloor \frac{n-1}{2} \right\rfloor \right) - ini + 1$$

$$= \left\lfloor \frac{n-1}{2} \right\rfloor + 1 = \left\lfloor \frac{n+1}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$$

```
Merge-Sort (A[1..p], ini, fim)
1   Se   ini < fim                                  ] A = Θ(1)
2        meio = ⌊(ini + fim) / 2⌋                   ]
3        Merge-Sort (A, ini, meio)                  ] B = T(⌈n/2⌉)
4        Merge-Sort (A, meio+1, fim)                ] C = T(⌊n/2⌋)
5        Combina (A, ini, meio, fim)                ] D = Θ(n)
```

$n = fim - ini + 1 \Leftrightarrow fim = n + ini - 1$

\# elementos em $A[meio+1..fim] = fim - meio$

$= n + \cancel{ini} - 1 - \left(\cancel{ini} + \left\lfloor \frac{n-1}{2} \right\rfloor\right)$

$= n - \left\lceil \frac{n}{2} \right\rceil$

$= \left\lfloor \frac{n}{2} \right\rfloor$

```
Merge-Sort (A[1..p], ini, tim)
1   Se   ini < tim                                  ] A = Θ(1)
2        meio = ⌊(ini + tim) / 2⌋                   ]
3        Merge-Sort (A, ini, meio)                  ] B = T(⌈n/2⌉)
4        Merge-Sort (A, meio+1, tim)                ] C = T(⌊n/2⌋)
5        Combina (A, ini, meio, tim)                ] D = Θ(n)
```

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) \end{cases}$$

→ Provemos na aula de recorrência que $T(n) = \Theta(n \log n)$.

Correção MergeSort

→ Antes de provarmos a correção de MergeSort, precisamos provar a correção de Combina.

Teo 1.
Sejam $A[1..p]$ um vetor de números e $ini, meio, fim \in \mathbb{N}$ tais que $1 \leq ini \leq meio \leq fim \leq p$. Se os subvetores $A[ini..meio]$ e $A[meio+1..fim]$ estão ordenados, então após a execução de Combina$(A, ini, meio, fim)$ temos que o subvetor $A[ini..fim]$ está ordenado.

Correção MergeSort

→ Antes de provarmos a correção de MergeSort, precisamos provar a correção de Combina.

Teo 1.

Sejam $A[1..p]$ um vetor de números e $ini, meio, fim \in \mathbb{N}$ tais que $1 \leq ini \leq meio \leq fim \leq p$. Se os subvetores $A[ini..meio]$ e $A[meio+1..fim]$ estão ordenados, então após a execução de Combina$(A, ini, meio, fim)$ temos que o subvetor $A[ini..fim]$ está ordenado.

→ Combina é um algoritmo iterativo, portanto sua prova é análoga às vistas anteriormente (exercício)

Teo 2.

Seja $A[1..p]$ um vetor e sejam $ini, fim \in \mathbb{N}$ tais que $ini \leq fim$. Então MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$.

Teo 2.

Seja $A[1..p]$ um vetor e sejam $ini, fim \in \mathbb{N}$ tais que $ini \leq fim$. Então MergeSort(A, ini, fim) ordena o subvetor $A[ini..fim]$.

Demo.

- A prova segue por indução em $n = fim - ini + 1$ (# de elementos a serem ordenados)

- **BASE $n = 1$**

- Se $n = 1$, então $fim = ini$
- Assim, $A[ini..fim] = A[ini..ini] = A[ini]$, e portanto está trivialmente ordenado.

Passo ($n \geq 2$)

- Agora suponha que $n \geq 2$ e que o MergeSort (A, ini^*, fim^*) ordena corretamente o subvetor $A[ini^* .. fim^*]$ qndo $k < n$, onde $k = fim^* - ini^* + 1$.
- Como $n \geq 2$, temos que $fim \geq ini + 1$. Neste caso o teste da linha 1 dá verdadeiro e o corpo (linhas 2-5) é executado.
- Na linha 2 o algoritmo faz meio = $\lfloor (ini + fim)/2 \rfloor$
- Na linha 3 fazemos a chamada recursiva MergeSort(A, ini, meio)

```
Merge-Sort (A[1..p], ini, fim)
1 Se   ini < fim
2      meio = ⌊(ini + fim) / 2⌋
3      Merge-Sort (A, ini, meio)
4      Merge-Sort (A, meio+1, fim)
5      Combina (A, ini, meio, fim)
```

- Na linha 3 fazemos a chamada recursiva MergeSort(A, ini, meio)
- Note que

$$k = meio - ini + 1 = \left\lfloor \frac{fim + ini}{2} \right\rfloor - ini + 1$$

$$\leq \frac{fim + ini - 2ini + 2}{2} = \frac{fim - ini + 2}{2} = \frac{n+1}{2} < n \quad (n \geq 2)$$

- Então, por hipótese de indução, MergeSort(A, ini, meio) ordena o subvetor A[ini .. meio].
- Analogamente

$$k' = fim - (meio+1) + 1 < n$$

- Assim, por H.I., MergeSort(A, meio+1, fim) ordena A[meio+1 .. fim]

```
Merge-Sort (A[1..p], ini, fim)
1 Se   ini < fim
2      meio = ⌊(ini + fim)/2⌋
3      Merge-Sort (A, ini, meio)
4      Merge-Sort (A, meio+1, fim)
5      Combina (A, ini, meio, fim)
```

- Então, por hipótese de indução, MergeSort(A, ini, meio) ordena o subvetor A[ini..meio].
- Analogamente

$$k' = fim - (meio+1) + 1 < n$$

- Assim, por H.I., MergeSort(A, meio+1, fim) ordena A[meio+1..fim]
- Após a execução da linha 4, executamos a linha 5
- Como os subvetores A[ini..meio] e A[meio+1..fim] estão ordenados, temos pelo teorema 1, que após a execução de Combina(A, ini, meio, fim) o subvetor A[ini..fim] está ordenado.
- Isso finaliza a prova.

```
Merge-Sort(A[1..p], ini, fim)
1 Se   ini < fim
2      meio = ⌊(ini + fim)/2⌋
3      Merge-Sort(A, ini, meio)
4      Merge-Sort(A, meio+1, fim)
5      Combina(A, ini, meio, fim)
```

Divisão e Conquista

- É uma técnica de projeto de algoritmos que envolve recursão
- A ideia é dividir a instância do problema em duas ou mais instâncias menores (divisão), resolvê-las de forma recursiva, e combinar as soluções para as instâncias menores em uma solução da instância original (conquista).

Exemplo clássico: Merge-Sort