ACH2002

Aula 8

Técnicas de Desenvolvimento de Algoritmos -

Divisão e Conquista (parte 2) e equações de recorrência

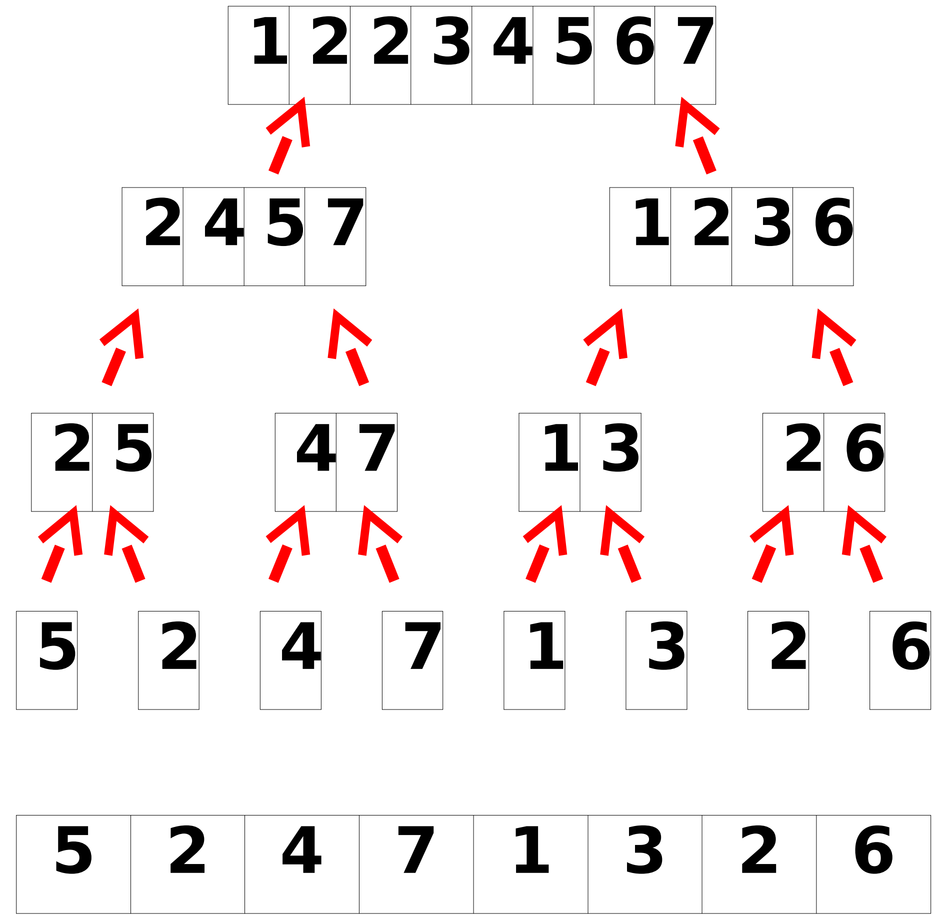
(adaptados dos slides de aula da Profa. Fátima L. S. Nunes)

Aula passada

- Algoritmo de ordenação Mergesort (ordenação por **intercalação**)
- **Divisão e conquista** (um tipo de técnica muito comum em recursividade)
- EP 1

MergeSort

```
mergeSort (A,i,f)
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```



Arranjo inicial

Divide até obter subarranjos com tamanho 1.
Então, começa a mesclar...

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // A[i..m] e A [m+1..f] estão ordenados
   // intercala os subarranjos para formar novo arranjo A

   // define subarranjos
   n1 ← m–i+1
   n2 ← f–m

   // preciso criar uma cópia dessas duas partes (subarranjos)
   // com sentinela (para evitar testar se chegou fim)
   criar arranjos L[1..n1+1] e R[1..n2+1]
   L[n1+1] ← ∞
   R[n2+1] ← ∞

   para j ← 1 até n1
       L[j] ← A[i+j-1]
   para j ← 1 até n2
       faça R[j] ← A[m+j]
       // continua
```

MergeSort – **A intercalação**

- Algoritmo da parte de intercalação (ainda não é o alg. todo):

```
merge (A,i,m,f) // A=arranjo; i,m,f=índices, i ≤ m < f
   // ... continuação

   //  mesclar subarranjos
   kL ← 1 // kL é o índice que percorre L
   kR ← 1 // kR é o índice que percorre R
   para k ← i até f   // k é o índice que percorre A
       se L[kL] ≤ R[kR]
            A[k] ← L[kL]
            kL ← kL + 1
       senão
            A[k] ← R[kR]
            kR ← kR + 1
       fim se
   fim para
```

EACH USP

MergeSort

- Ordenação por **intercalação** (que é a forma de combinar):
  - Dados *n* e uma sequência de *n* elementos:
    - Dividir: divide o arranjo em duas subsequências de *n/2* elementos;
    - Conquistar: classifica as duas subsequências recursivamente, utilizando a própria ordenação por intercalação;
    - Combinar: faz a intercalação das duas sequências ordenadas, de modo a produzir a resposta ordenada.

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão:
  - Dividir – o problema em um determinado número de subproblemas.
  - Conquistar – os subproblemas, resolvendo-os recursivamente. Se os tamanhos dos subproblemas forem pequenos o suficiente, resolvê-los diretamente.
  - Combinar – as soluções dadas aos subproblemas para formar solução procurada para problema original.

Aula de hoje

- Análise de complexidade do MergeSort
  - Meio que intuitiva…
  - Isso muda na versão iterativa?

- Mais sobre equações de recorrência
  - Para auxiliar análises de forma mais geral

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão: dividir, conquistar e combinar

- Como se calcula a complexidade geral, dados esses 3 passos, considerando entrada de tamanho ***n***?

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão: dividir, conquistar e combinar

- Como se calcula a complexidade geral, dados esses 3 passos, considerando entrada de tamanho ***n***?
  - T(n) = complexidade(dividir(n)) + complexidade(conquistar(n)) + complexidade(combinar(n))

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão: dividir, conquistar e combinar

- Como se calcula a complexidade geral, dados esses 3 passos, considerando entrada de tamanho ***n***?
  - T(n) = complexidade(dividir(n)) + complexidade(conquistar(n)) + complexidade(combinar(n))
  - Para entradas pequenas ($n \leq c$, c pequeno), podemos assumir T(n) = O(1) - quando paramos de dividir

Paradigma Dividir e conquistar

- Três passos em cada nível de recursão: dividir, conquistar e combinar

- Como se calcula a complexidade geral, dados esses 3 passos, considerando entrada de tamanho ***n***?
  - T(n) = complexidade(dividir(n)) + complexidade(conquistar(n)) + complexidade(combinar(n))
  - Para entradas pequenas ($n \leq c$, c pequeno), podemos assumir T(n) = O(1) - quando paramos de dividir

- Problemas resolvidos com o paradigma dividir e conquistar têm T(n) expressa em função da própria T(n) na complexidade de conquistar.
  - Nesses casos, dizemos que T(n) é uma equação de recorrência.
  - T(n) da etapa de ***conquistar*** é expresso pelo tamanho do subproblema.
  - Exemplo: se temos ***a*** chamadas recursivas e em cada chamada o problema é dividido em subproblemas de tamanho **n/b**:
  - $T(n) = aT(n/b)$

Paradigma Dividir e conquistar

- Chamando o tempo de **dividir** e **combinar** de D(n) e C(n), respectivamente:
- T(n) = aT(n/b) + D(n) + C(n)
- Considerando que para ***n*** suficiente pequeno T(n) = O(1) (quando paramos de dividir)

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+f(n), caso contrário \end{cases}$$

f(n)=D(n)+C(n)

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **a** ?

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **a** ?
Quantidade de subproblemas disjuntos que eu divido em cada passo (ou seja, número de chamadas recursivas)

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **n/b** ?

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **n/b** ?
Tamanho dos subproblemas
(nem sempre a = b)

onde :

$a \geq 1$;

$b > 1$;

f(n) é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **f(n)**?

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Forma geral de uma recorrência que usa paradigma *dividir e conquistar* :

$$T(n) = aT(n/b) + f(n)$$

O que é **f(n)**?
Função que fornece a complexidade das etapas de divisão e combinação.

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

Equações de recorrência

- Qual a equação de recorrência de complexidade do algoritmo MergeSort?

$$T(n) = aT(n/b) + f(n)$$

onde :

$a \geq 1$;

$b > 1$;

$f(n)$ é uma função assintoticamente positiva.

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

- Dividir: ⇒ D(n) = ?
- Conquistar: resolvemos recursivamente dois subproblemas, cada um com tamanho n/2 ⇒ ?
- Combinar: o método ***merge*** *em um subarranjo com* ***n*** *elementos tem o tempo* ?

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemos recursivamente dois subproblemas, cada um com tamanho n/2 ⇒ ?
- Combinar: o método ***merge*** *em um subarranjo com* ***n*** *elementos tem o tempo* **?**

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemos recursivamente dois subproblemas, cada um com tamanho n/2 ⇒ a = b = 2 ⇒ 2T(n/2)
- Combinar: o método ***merge*** *em um subarranjo com* ***n*** *elementos tem o tempo* **?**

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemos recursivamente dois subproblemas, cada um com tamanho n/2 ⇒ a = b = 2 ⇒ 2T(n/2)
- Combinar: o método ***merge*** *em um subarranjo com* ***n*** *elementos tem o tempo* O(n)

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n)=\begin{cases} O(1), n \leq c \\ aT(n/b)+D(n)+C(n), \text{caso contrário} \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemo … as, cada um com tamanho …
- Combinar: já vimos que o método ***merge*** *em um subarranjo com n elementos tem o tempo* O(n)

D(n) + C(n) = O(1) + O(n) = O(n)

MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n) = \begin{cases} O(1), n \leq c \\ aT(n/b) + D(n) + C(n), caso contrário \end{cases}$$

- Dividir: somente calcula o ponto médio do subarranjo ⇒ constante ⇒ D(n) = O(1)
- Conquistar: resolvemo ... as, cada um com tamanho
- Combinar: já vimos que o método ***merge*** *em um subarranjo com n elementos tem o tempo* O(n)

D(n) + C(n) = O(1) + O(n) = O(n)

Além de O, é mais alguma coisa?

Complexidade de tempo do MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

Complexidade de tempo do MergeSort

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1\,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1\,. \end{cases}$$

E quanto é isso afinal?

Veremos técnicas para resolver equações de recorrências, mas esta dá para resolver intuitivamente...

Complexidade de tempo do MergeSort (cálculo “por intuição”)

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

Complexidade de tempo do MergeSort (cálculo "por intuição")

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

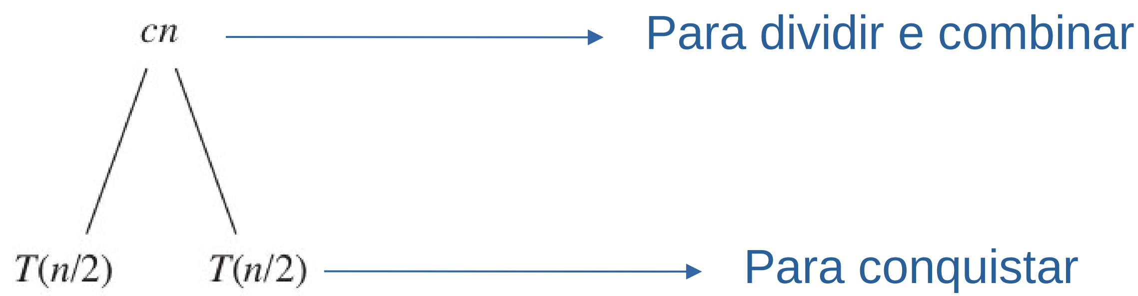
$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

$$T(n)$$

Complexidade de tempo do MergeSort (cálculo "por intuição")

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

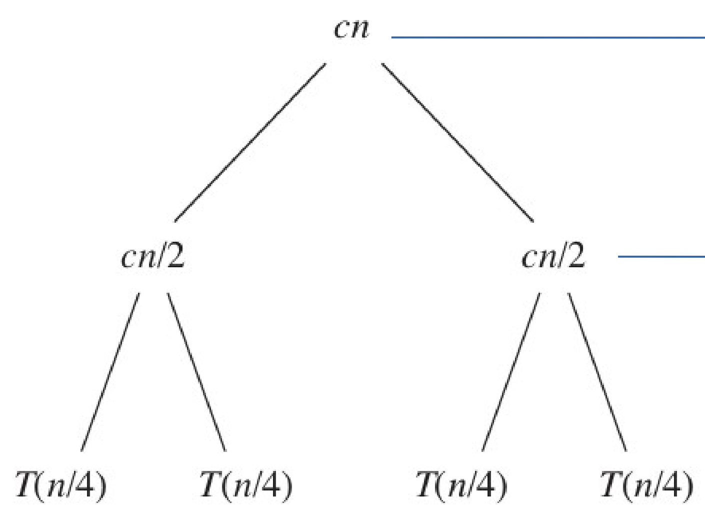
$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$



Complexidade de tempo do MergeSort (cálculo “por intuição”)

$$T(n) = \begin{cases} c & \text{if } n = 1 , \\ 2T(n/2) + cn & \text{if } n > 1 , \end{cases}$$

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```



Para dividir e combinar

Para dividir e combinar

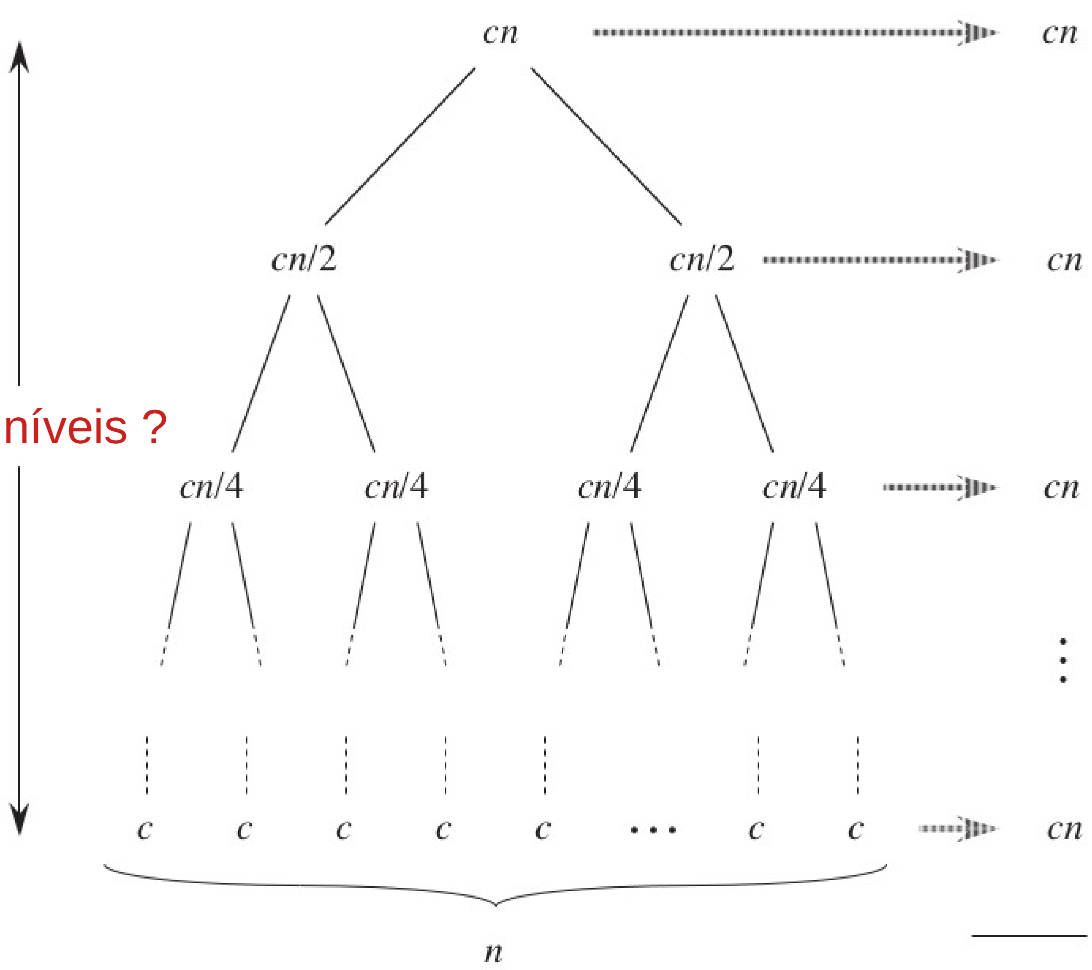
Para conquistar

Complexidade de tempo do MergeSort (cálculo "por intuição")

$$T(n) = \begin{cases} c & \text{if } n = 1 \,, \\ 2T(n/2) + cn & \text{if } n > 1 \,, \end{cases}$$

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

Quantos níveis ?



(d)

Para dividir e combinar

Para dividir e combinar

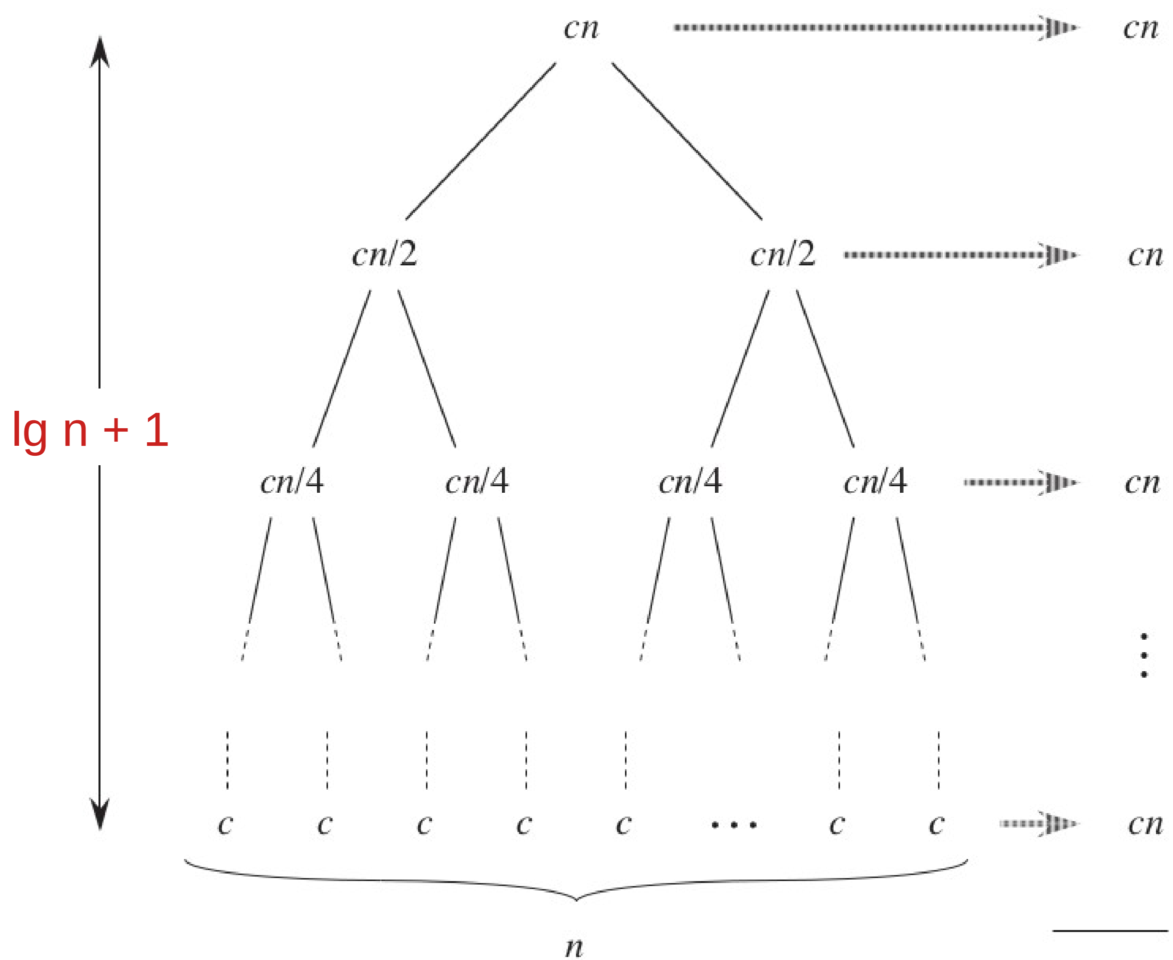
Para dividir e combinar

Para conquistar o caso base

Complexidade de tempo do MergeSort (cálculo "por intuição")

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```

$$T(n) = \begin{cases} c & \text{if } n = 1 \, , \\ 2T(n/2) + cn & \text{if } n > 1 \, , \end{cases}$$



Para dividir e combinar

Para dividir e combinar

Para dividir e combinar

Para conquistar o caso base

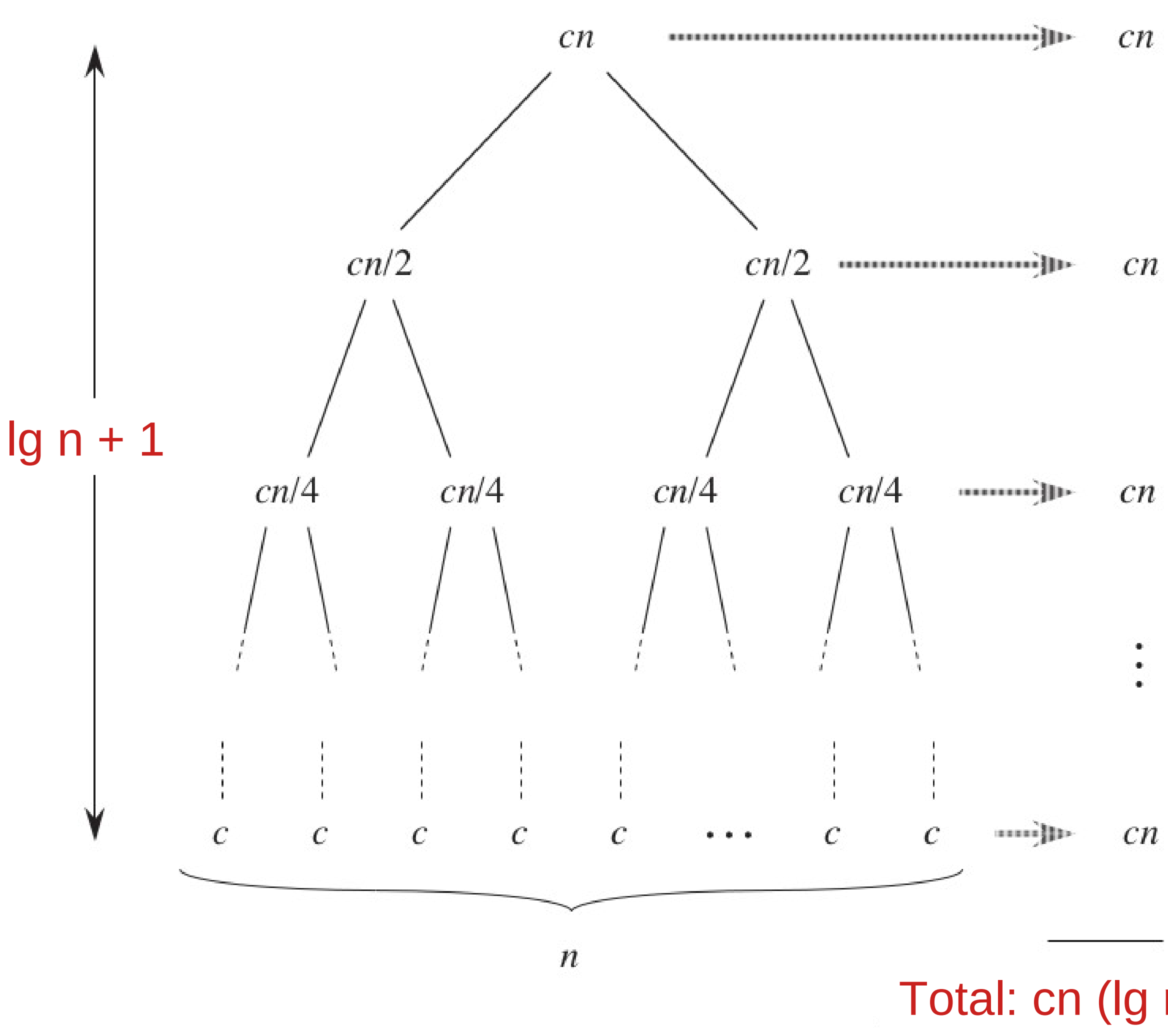
Total:

(d)

Complexidade de tempo do MergeSort (cálculo “por intuição”)

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```



(d)

Para dividir e combinar

Para dividir e combinar

Para dividir e combinar

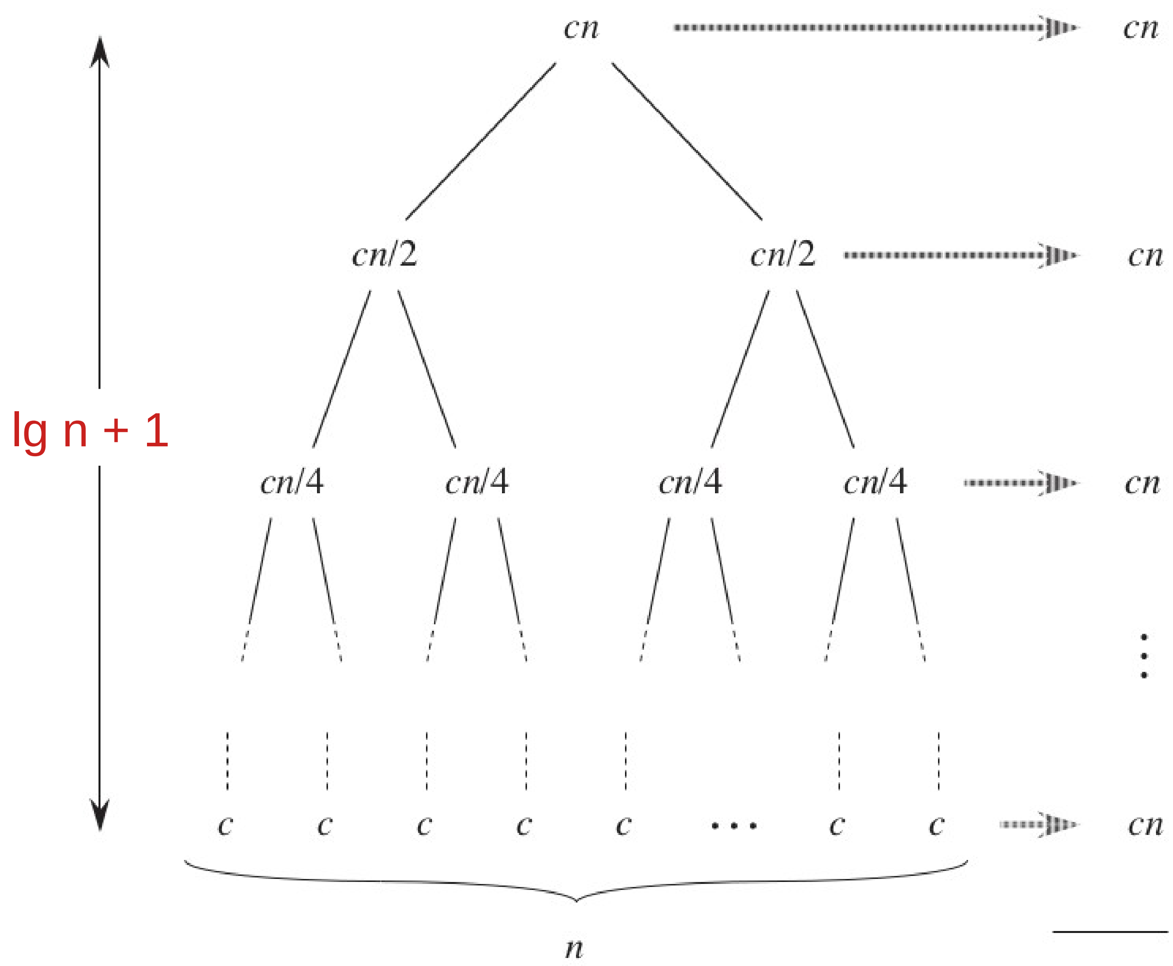
Para conquistar o caso base

Total: cn (lg n + 1) = O(?)

Complexidade de tempo do MergeSort (cálculo “por intuição”)

$$T(n) = \begin{cases} c & \text{if } n = 1\,, \\ 2T(n/2) + cn & \text{if } n > 1\,, \end{cases}$$

```
mergeSort (A,p,r)
se p < r
  q ← ⌊(p+r)/2⌋
  mergeSort(A, p, q)
  mergeSort(A, q+1,r)
  merge(A,p,q,r)
```



(d)

Para dividir e combinar

Para dividir e combinar

Para dividir e combinar

Para conquistar o caso base

Total: cn (lg n + 1) = O(n lg n)

InsertionSort x MergeSort

- Em termos de complexidade de tempo:
  - InsertionSort: O(?)
  - MergeSort: O(?)

- Quem é mais rápido?

InsertionSort x MergeSort

- Em termos de complexidade de tempo:
  - InsertionSort: $O(n^2)$
  - MergeSort: $O(n \lg n)$

- Quem é mais rápido?

InsertionSort x MergeSort

- Em termos de complexidade de tempo:
  - InsertionSort: $O(n^2)$
  - MergeSort: $O(n \lg n)$

- Quem é mais rápido?
  - MergeSort, pois

$$n \lg n = o(n^2)$$

InsertionSort x MergeSort

- Em termos de complexidade de espaço:
  - InsertionSort: O(?)
  - MergeSort: O(?)

InsertionSort x MergeSort

- Em termos de complexidade de espaço:
  - InsertionSort: O(n)
  - MergeSort: O(n)

assumindo que não tem vazamento de memória

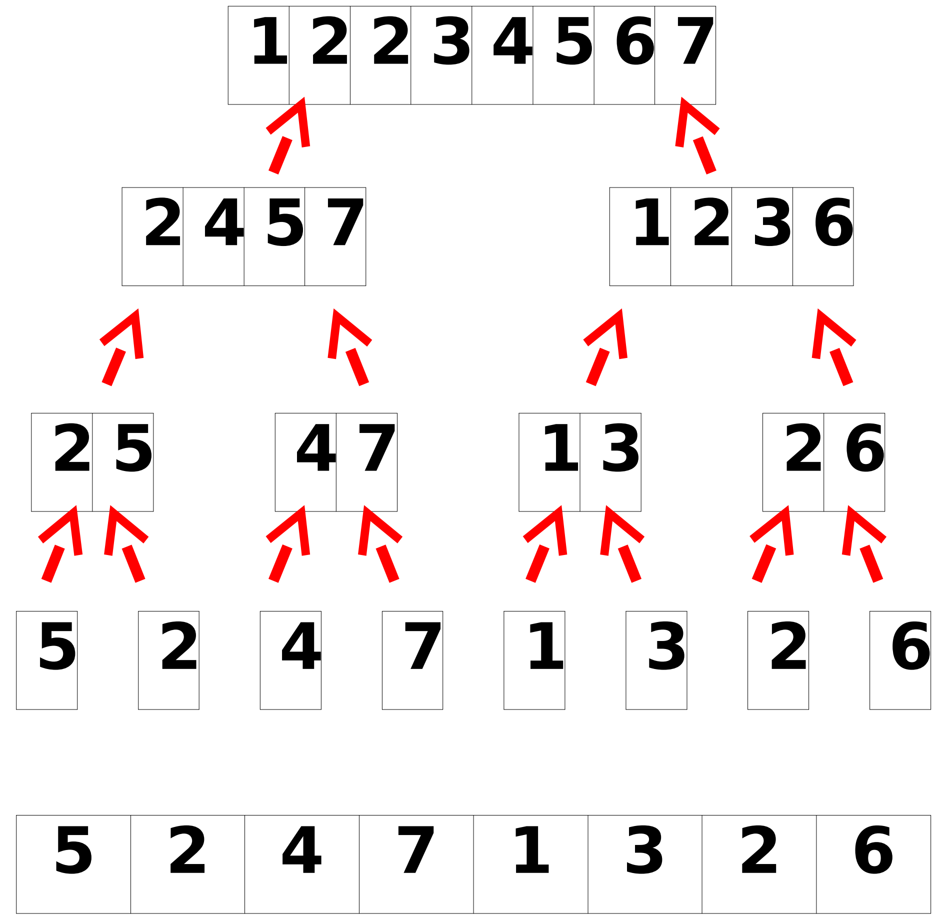
(mas usa um vetor auxiliar que o InsertionSort não usa)

Como seria a versão iterativa do algoritmo MergeSort?
Será que a complexidade muda?

Usando a função merge(A, i, m, f) já pronta.

Lembrando o MergeSort recursivo...

```
mergeSort (A,i,f)
se i < f
        m ← ⌊(i+f)/2⌋
        mergeSort(A, i, m)
        mergeSort(A, m+1,f)
        merge(A,i,m,f)
```



Arranjo inicial

Divide até obter subarranjos com tamanho 1.
Então, começa a mesclar...

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores

9	1	2	9	3	8	4	7	5	6	5

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$ (b = tamanho do bloco)

in in+2b

Ex: b = 1

9	1	2	9	3	8	4	7	5	6	5

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$ (b = tamanho do bloco)

in in+2b

Ex: b = 1										
9	1	2	9	3	8	4	7	5	6	5

mergeSortIterativo (A,n)

```
b ← 1      /* tamanho do bloco */
enquanto b < n
        in ← 1  /* início do bloco da esquerda */
        enquanto ((in + b) <= n)
                fim ← in + 2*b - 1
                se (fim > n) fim ← n
                merge(A, in, in+b-1, fim)
                in ← in + 2*b
        b ← 2*b
```

Enquanto tiver um bloco à direita para intercalar….

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$ (b = tamanho do bloco)

in in+2b

Ex: b = 1										
9	1	2	9	3	8	4	7	5	6	5

mergeSortIterativo (A,n)

```
b ← 1      /* tamanho do bloco */
enquanto b < n
        in ← 1  /* início do bloco da esquerda */
        enquanto ((in + b) <= n)
                fim ← in + 2*b - 1
                se (fim > n) fim ← n
                merge(A, in, in+b-1, fim)
                in ← in + 2*b
        b ← 2*b
```

Enquanto tiver um bloco à direita para intercalar….

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$ (b = tamanho do bloco)

	in		in+2b								
Ex: b = 1	9	1	2	9	3	8	4	7	5	6	5

mergeSortIterativo (A,n)

```
b ← 1     /* tamanho do bloco */
enquanto b < n
        in ← 1  /* início do bloco da esquerda */
        enquanto ((in + b) <= n)
                fim ← in + 2*b - 1
                se (fim > n) fim ← n
                merge(A, in, in+b-1, fim)
                in ← in + 2*b
        b ← 2*b
```

Complexidade:

https://www.youtube.com/watch?v=IN_ZOU-LK08

MergeSort iterativo

Na versão iterativa do algoritmo Mergesort, cada iteração intercala dois "blocos" de b elementos: o primeiro bloco com o segundo, o terceiro com o quarto etc. A variável b assume os valores $1, 2, 4, 8, \ldots$ (b = tamanho do bloco)

in in+2b

Ex: b = 1

9	1	2	9	3	8	4	7	5	6	5

```
mergeSortIterativo (A,n)
b ← 1     /* tamanho do bloco */
enquanto b < n
        in ← 1  /* início do bloco da esquerda */
        enquanto ((in + b) <= n)
                fim ← in + 2*b - 1
                se (fim > n) fim ← n
                merge(A, in, in+b-1, fim)
                in ← in + 2*b
        b ← 2*b
```

Complexidade:

A análise é similar à da versão recursiva: **O(n lg n)**

https://www.youtube.com/watch?v=IN_ZOU-LK08

Exercício para fazer ANTES da próxima aula: Como seria a versão recursiva do algoritmo de busca binária?

A solução é também do tipo "dividir e conquistar"

Referências

- **Mergesort iterativo**: Paulo Feofiloff. Algoritmos em linguagem C. Cap 9.
- Nívio Ziviani. Projeto de Algoritmos com implementações em C e Pascal. Editora Thomson, 2a. Edição, 2004. Cap 2.4 e 2.5
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Algoritmos - Tradução da 2a. Edição Americana. Editora Campus, 2002 (Cap 2.3)