

# MC102 – Aula29

## Recursão IV - MergeSort

Prof. Luiz F. Bittencourt

Turmas QR

Instituto de Computação – Unicamp

2019

Conteúdo adaptado de slides fornecidos pelo Prof. Eduardo Xavier.

# Introdução

- Problema:

- ▶ Temos uma lista  $\mathbf{v}$  de inteiros de tamanho  $\mathbf{n}$ .
- ▶ Devemos deixar  $\mathbf{v}$  ordenada crescentemente.

- Veremos um algoritmo baseado na técnica **dividir-e-conquistar** que usa recursão.

# Introdução

- Problema:
  - ▶ Temos uma lista  $\mathbf{v}$  de inteiros de tamanho  $\mathbf{n}$ .
  - ▶ Devemos deixar  $\mathbf{v}$  ordenada crescentemente.
- Veremos um algoritmo baseado na técnica **dividir-e-conquistar** que usa recursão.

# Dividir e Conquistar

- Temos que resolver um problema  $P$  de tamanho  $n$ .
- **Dividir:** Quebramos  $P$  em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior  $P$ .

# Dividir e Conquistar

- Temos que resolver um problema  $P$  de tamanho  $n$ .
- **Dividir:** Quebramos  $P$  em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior  $P$ .

# Dividir e Conquistar

- Temos que resolver um problema  $P$  de tamanho  $n$ .
- **Dividir:** Quebramos  $P$  em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior  $P$ .

# Dividir e Conquistar

- Temos que resolver um problema  $P$  de tamanho  $n$ .
- **Dividir:** Quebramos  $P$  em sub-problemas menores.
- Resolvemos os sub-problemas de forma recursiva.
- **Conquistar:** Unimos as soluções dos sub-problemas para obter solução do problema maior  $P$ .

# Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar uma lista de tamanho  $n$ .
  - ▶ **Dividir:** Dividimos a lista de tamanho  $n$  em duas sub-listas de tamanho aproximadamente iguais (uma de tamanho  $\lceil n/2 \rceil$  e outra de tamanho  $\lfloor n/2 \rfloor$ ).
  - ▶ Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas.
  - ▶ **Conquistar:** Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho  $n$ .



# Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar uma lista de tamanho  $n$ .
  - ▶ **Dividir:** Dividimos a lista de tamanho  $n$  em duas sub-listas de tamanho aproximadamente iguais (uma de tamanho  $\lceil n/2 \rceil$  e outra de tamanho  $\lfloor n/2 \rfloor$ ).
  - ▶ Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas.
  - ▶ **Conquistar:** Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho  $n$ .

# Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar uma lista de tamanho  $n$ .
  - ▶ **Dividir:** Dividimos a lista de tamanho  $n$  em duas sub-listas de tamanho aproximadamente iguais (uma de tamanho  $\lceil n/2 \rceil$  e outra de tamanho  $\lfloor n/2 \rfloor$ ).
  - ▶ Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas.
  - ▶ **Conquistar:** Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho  $n$ .

# Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar uma lista de tamanho  $n$ .
  - ▶ **Dividir:** Dividimos a lista de tamanho  $n$  em duas sub-listas de tamanho aproximadamente iguais (uma de tamanho  $\lceil n/2 \rceil$  e outra de tamanho  $\lfloor n/2 \rfloor$ ).
  - ▶ Resolvemos o problema de ordenação de forma recursiva para estas duas sub-listas.
  - ▶ **Conquistar:** Com as duas sub-listas ordenadas, construímos uma lista ordenada de tamanho  $n$ .

# Merge-Sort: Ordenação por intercalação

Conquistar: Dadas duas listas  $v_1$  e  $v_2$  ordenadas, como obter uma outra lista ordenada contendo os elementos de  $v_1$  e  $v_2$ ?

$v_1$	3	5	7	10	11	12
-------	---	---	---	----	----	----

$v_2$	4	6	8	9	11	13	14
-------	---	---	---	---	----	----	----

3	4	5	6	7	8	9	10	11	11	12	13	14
---	---	---	---	---	---	---	----	----	----	----	----	----

# Merge (Fusão)

- A ideia é executar um laço onde em cada iteração testamos quem é o menor elemento dentre  $v_1[i]$  e  $v_2[j]$ , e copiamos este elemento para uma nova lista.
- Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de uma das listas ( $v_1$  ou  $v_2$ ) foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes da outra lista.

# Merge (Fusão)

- A ideia é executar um laço onde em cada iteração testamos quem é o menor elemento dentre  $v_1[i]$  e  $v_2[j]$ , e copiamos este elemento para uma nova lista.
- Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de uma das listas ( $v_1$  ou  $v_2$ ) foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes da outra lista.

# Merge (Fusão)

Retorna uma lista que é a fusão das listas passadas por parâmetro:

*#Devolve lista com fusão de a e b*

```
def merge(a, b):  
    i=0; j=0; #índice de a e b, respectivamente  
    c = []  
  
    while(i < len(a) and j < len(b)): #Enquanto não avaliou completamente um dos  
        if(a[i] <= b[j]): #vetores, copia menor elemento para c  
            c.append(a[i])  
            i = i + 1  
        else:  
            c.append(b[j])  
            j = j + 1  
  
    while(i < len(a)): #copia resto de a  
        c.append(a[i])  
        i = i + 1  
  
    while(j < len(b)): #copia resto de b  
        c.append(b[j])  
        j = j + 1  
  
    return c
```

# Merge (Fusão)

Retorna uma lista que é a fusão das listas passadas por parâmetro:

*#Devolve lista com fusão de a e b*

```
def merge(a, b):  
    i=0; j=0; #índice de a e b, respectivamente  
    c = []  
  
    while(i < len(a) and j < len(b)): #Enquanto não avaliou completamente um dos  
        if(a[i] <= b[j]): #vetores, copia menor elemento para c  
            c.append(a[i])  
            i = i + 1  
        else:  
            c.append(b[j])  
            j = j + 1  
  
    while(i < len(a)): #copia resto de a  
        c.append(a[i])  
        i = i + 1  
  
    while(j < len(b)): #copia resto de b  
        c.append(b[j])  
        j = j + 1  
  
    return c
```



# Merge (Fusão)

Retorna uma lista que é a fusão das listas passadas por parâmetro:

```
#Devolve lista com fusão de a e b
def merge(a, b):
    i=0; j=0; #índice de a e b, respectivamente
    c = []

    while(i < len(a) and j < len(b)): #Enquanto não avaliou completamente um dos
        if(a[i] <= b[j]): #vetores, copia menor elemento para c
            c.append(a[i])
            i = i + 1
        else:
            c.append(b[j])
            j = j + 1

    while(i < len(a)): #copia resto de a
        c.append(a[i])
        i = i + 1

    while(j < len(b)): #copia resto de b
        c.append(b[j])
        j = j + 1

    return c
```

# Merge (Fusão)

- A função descrita recebe duas listas ordenadas e devolve uma terceira contendo todos os elementos em ordem.
- Porém, no merge-sort faremos a intercalação de sub-listas de uma mesma lista.
- Isto evita a criação de várias listas durante as várias chamadas recursivas, melhorando o desempenho do algoritmo.
- Teremos posições **ini**, **meio**, **fim** de uma lista e devemos fazer a intercalação das duas sub-listas: uma de **ini** até **meio**, e outra de **meio+1** até **fim**.
  - ▶ Para isso, a função utiliza uma lista auxiliar que receberá o resultado da intercalação, e que no final é copiada para a lista original a ser ordenada.

# Merge (Fusão)

- A função descrita recebe duas listas ordenadas e devolve uma terceira contendo todos os elementos em ordem.
- Porém, no merge-sort faremos a intercalação de sub-listas de uma mesma lista.
- Isto evita a criação de várias listas durante as várias chamadas recursivas, melhorando o desempenho do algoritmo.
- Teremos posições **ini**, **meio**, **fim** de uma lista e devemos fazer a intercalação das duas sub-listas: uma de **ini** até **meio**, e outra de **meio+1** até **fim**.
  - ▶ Para isso, a função utiliza uma lista auxiliar que receberá o resultado da intercalação, e que no final é copiada para a lista original a ser ordenada.

# Merge (Fusão)

- A função descrita recebe duas listas ordenadas e devolve uma terceira contendo todos os elementos em ordem.
- Porém, no merge-sort faremos a intercalação de sub-listas de uma mesma lista.
- Isto evita a criação de várias listas durante as várias chamadas recursivas, melhorando o desempenho do algoritmo.
- Teremos posições **ini**, **meio**, **fim** de uma lista e devemos fazer a intercalação das duas sub-listas: uma de **ini** até **meio**, e outra de **meio+1** até **fim**.
  - ▶ Para isso, a função utiliza uma lista auxiliar que receberá o resultado da intercalação, e que no final é copiada para a lista original a ser ordenada.

## Merge (Fusão)

Faz intercalação de pedaços de **v**. No fim **v** estará ordenada entre as posições **ini** e **fim**:

```
def merge(v, ini, meio, fim, aux):
    i=ini; j=meio+1; k=0; #índices da metade inf, sup e aux, respectivamente.

    while(i <= meio and j <= fim): #Enquanto não avaliou completamente um dos
                                    #vetores, copia menor elemento para aux
        if(v[i] <= v[j]):
            aux[k] = v[i]
            k = k + 1
            i = i + 1
        else:
            aux[k] = v[j]
            k = k + 1
            j = j + 1

    while(i <= meio): #copia resto da primeira sub-lista
        aux[k] = v[i]
        k = k + 1
        i = i + 1

    while(j <= fim): #copia resto da segunda sub-lista
        aux[k] = v[j]
        k = k + 1
        j = j + 1

    i = ini; k = 0;
    while(i <= fim): #copia lista ordenada aux para v
        v[i]=aux[k]
        i = i + 1
        k = k + 1
```

# Merge (Fusão)

Faz intercalação de pedaços de **v**. No fim **v** estará ordenada entre as posições **ini** e **fim**:

```
def merge(v, ini, meio, fim, aux):
    i=ini; j=meio+1; k=0; #índices da metade inf, sup e aux, respectivamente.

    while(i <= meio and j <= fim):#Enquanto não avaliou completamente um dos
                                   #vetores, copia menor elemento para aux
        if(v[i] <= v[j]):
            aux[k] = v[i]
            k = k + 1
            i = i + 1
        else:
            aux[k] = v[j]
            k = k + 1
            j = j + 1

    while(i <= meio): #copia resto da primeira sub-lista
        aux[k] = v[i]
        k = k + 1
        i = i + 1

    while(j <= fim): #copia resto da segunda sub-lista
        aux[k] = v[j]
        k = k + 1
        j = j + 1

    i = ini; k = 0;
    while(i <= fim): #copia lista ordenada aux para v
        v[i]=aux[k]
        i = i + 1
        k = k + 1
```

# Merge (Fusão)

Faz intercalação de pedaços de **v**. No fim **v** estará ordenada entre as posições **ini** e **fim**:

```
def merge(v, ini, meio, fim, aux):
    i=ini; j=meio+1; k=0; #índices da metade inf, sup e aux, respectivamente.

    while(i <= meio and j <= fim):#Enquanto não avaliou completamente um dos
                                   #vetores, copia menor elemento para aux
        if(v[i] <= v[j]):
            aux[k] = v[i]
            k = k + 1
            i = i + 1
        else:
            aux[k] = v[j]
            k = k + 1
            j = j + 1

    while(i <= meio): #copia resto da primeira sub-lista
        aux[k] = v[i]
        k = k + 1
        i = i + 1

    while(j <= fim): #copia resto da segunda sub-lista
        aux[k] = v[j]
        k = k + 1
        j = j + 1

    i = ini; k = 0;
    while(i <= fim): #copia lista ordenada aux para v
        v[i]=aux[k]
        i = i + 1
        k = k + 1
```

# Merge-Sort

- O merge-sort resolve de forma recursiva dois sub-problemas, cada um contendo uma metade da lista original.
- Com a resposta das chamadas recursivas podemos chamar a função **merge** para obter uma lista ordenada.



# Merge-Sort

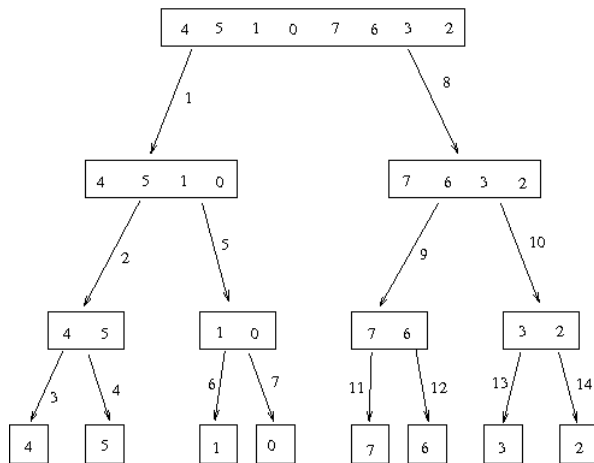
- O merge-sort resolve de forma recursiva dois sub-problemas, cada um contendo uma metade da lista original.
- Com a resposta das chamadas recursivas podemos chamar a função **merge** para obter uma lista ordenada.

# Merge-Sort

```
def mergeSort(v, ini, fim, aux):  
    meio = (fim+ini)//2  
    if(ini < fim): #lista tem pelo menos 2 elementos  
                   #para ordenar  
        mergeSort(v, ini, meio, aux)  
        mergeSort(v, meio+1, fim, aux)  
        merge(v, ini, meio, fim, aux)
```

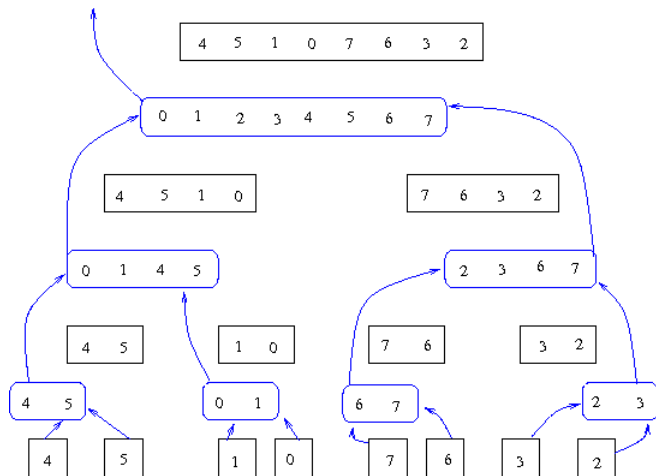
# Merge-Sort

Abaixo temos um exemplo com a ordem de execução das chamadas recursivas.



# Merge-Sort

Abaixo temos o retorno do exemplo anterior.



# Merge-Sort: Exemplo de uso

- Note que só criamos 2 listas, **v** a ser ordenada e **aux** do mesmo tamanho de **v**.
- Somente estas duas listas existirão durante todas as chamadas recursivas.

```
def main():  
    v = [12, 90, 47, -9, 78, 45, 78, 3323, 1, 2, 34, 20]  
    aux = [0 for i in range(12)]    #tem o mesmo tamanho de v  
    print(v)  
    mergeSort(v, 0, 11, aux)  
    print(v)
```

# Exercícios

- 1 Mostre passo a passo a execução da função merge considerando dois sub-vetores: (3, 5, 7, 10, 11, 12) e (4, 6, 8, 9, 11, 13, 14).
- 2 Faça uma execução Passo-a-Passo do Merge-Sort para o vetor: (30, 45, 21, 20, 6, 715, 100, 65, 33).
- 3 Reescreva o algoritmo Merge-Sort para que este passe a ordenar um vetor em ordem decrescente.
- 4 Considere o seguinte problema: Temos como entrada um vetor de inteiros  $v$  (não necessariamente ordenado), e um inteiro  $x$ . Desenvolva um algoritmo que determina se há dois números em  $v$  cuja soma seja  $x$ . Tente fazer o algoritmo o mais eficiente possível. Utilize um dos algoritmos de ordenação na sua solução.