MC102 – Aula 24
Recursão III - MergeSort

Eduardo C. Xavier

Instituto de Computação – Unicamp

14 de Dezembro de 2020

Introdução

- Problema:
  - Temos um vetor **v** de inteiros de tamanho **n**.
  - Devemos deixar **v** ordenado crescentemente.
- Veremos um algoritmo baseado na técnica **dividir-e-conquistar** que usa recursão.

Introdução

- Problema:
  - Temos um vetor **v** de inteiros de tamanho **n**.
  - Devemos deixar **v** ordenado 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.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **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.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **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.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **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.
- Mostramos como resolver casos básicos, como quando $n = 1$.
- Para $n > 1$ fazemos:
  - **Dividir:** Quebramos P em sub-problemas menores.
  - Resolvemos os sub-problemas de forma recursiva (Hip. Ind.).
  - **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 um vetor de tamanho n.
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

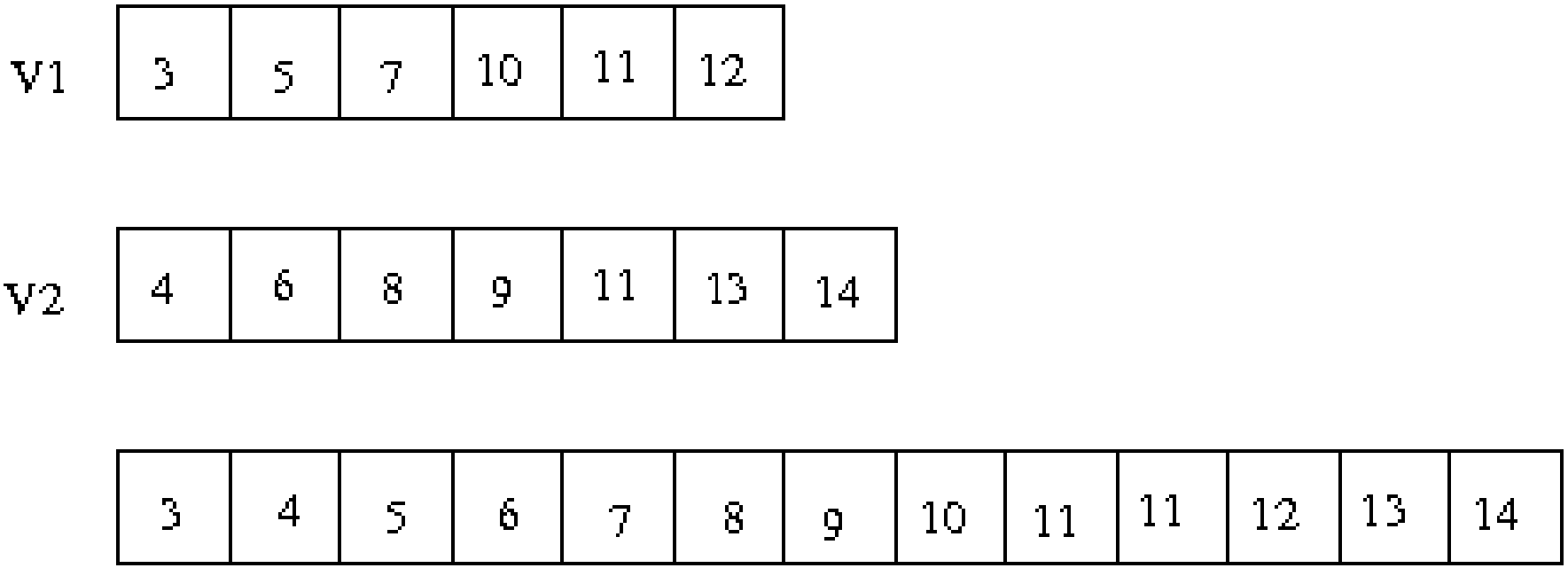
- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

- O Merge-Sort é um algoritmo baseado na técnica dividir-e-conquistar.
- Neste caso temos que ordenar um vetor de tamanho n.
- Se $n = 1$ então o problema está resolvido pois o vetor está ordenado.
- Para $n > 1$ temos:
  - **Dividir:** Dividimos o vetor de tamanho n em dois sub-vetores de tamanho aproximadamente iguais (um de tamanho $\lceil n/2 \rceil$ e outro de tamanho $\lfloor n/2 \rfloor$).
  - Resolvemos o problema de ordenação de forma recursiva para estes dois sub-vetores.
  - **Conquistar:** Com os dois sub-vetores ordenados, construímos um vetor de tamanho n ordenado.

Merge-Sort: Ordenação por intercalação

Conquistar: Dados dois vetores v_1 e v_2 ordenados, como obter um outro vetor ordenado contendo os elementos de v_1 e v_2?



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 o novo vetor.
- Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de um dos vetores (v_1 ou v_2) foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes do outro vetor.

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 o novo vetor.
- Durante a execução deste laço podemos chegar em uma situação onde todos os elementos de um dos vetores (v_1 ou v_2) foram todos avaliados. Neste caso terminamos o laço e copiamos os elementos restantes do outro vetor.

Merge (Fusão)

Retorna um vetor ordenado que é a fusão dos vetores ordenados passados por parâmetro:

```
def intercala(v1, v2):
        v3 = [0 for i in range(len(v1)+len(v2))]
        i = 0; j = 0; k = 0
        #enquanto nao termina um dos vetores
        while i<len(v1) and j<len(v2):
                if v1[i] <= v2[j]:
                        v3[k] = v1[i]
                        i += 1; k += 1
                else:
                        v3[k] = v2[j]
                        j += 1; k += 1

        #termina de copiar v1, se for o caso
        while i<len(v1):
                v3[k] = v1[i]
                i += 1; k += 1

        #termina de copiar v2, se for o caso
        while j<len(v2):
                v3[k] = v2[j]
                j += 1; k += 1

        return v3
```

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de sub-vetores usando sempre um mesmo vetor auxiliar.
  - Isto evitará a alocação de vários vetores durante as chamadas recursivas, melhorando a performance do algoritmo.
- Além disso os sub-vetores farão parte do vetor original a ser ordenado:
  - Teremos posições **ini, meio, fim** do vetor e devemos fazer a intercalação dos dois sub-vetores: um de **ini** até **meio**, e outro de **meio+1** até **fim**.
- A função utiliza o vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de sub-vetores usando sempre um mesmo vetor auxiliar.
  - Isto evitará a alocação de vários vetores durante as chamadas recursivas, melhorando a performance do algoritmo.
- Além disso os sub-vetores farão parte do vetor original a ser ordenado:
  - Teremos posições **ini**, **meio**, **fim** do vetor e devemos fazer a intercalação dos dois sub-vetores: um de **ini** até **meio**, e outro de **meio+1** até **fim**.
- A função utiliza o vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

- A função descrita recebe dois vetores ordenados e devolve um terceiro contendo todos os elementos.
- Porém no merge-sort faremos a intercalação de sub-vetores usando sempre um mesmo vetor auxiliar.
  - Isto evitará a alocação de vários vetores durante as chamadas recursivas, melhorando a performance do algoritmo.
- Além disso os sub-vetores farão parte do vetor original a ser ordenado:
  - Teremos posições **ini, meio, fim** do vetor e devemos fazer a intercalação dos dois sub-vetores: um de **ini** até **meio**, e outro de **meio+1** até **fim**.
- A função utiliza o vetor auxiliar, que receberá o resultado da intercalação, e que no final é copiado para o vetor a ser ordenado.

Merge (Fusão)

- Abaixo está a função que faz intercalação de pedaços de **v**.
- No fim **v** estará ordenado entre as posições **ini** e **fim**:

```
def intercala2(v, ini, meio, fim, aux):
        i = ini; j = meio+1; k = ini
        #enquanto nao termina um dos sub-vetores
        while i<=meio and j<=fim:
                if v[i] <= v[j]:
                        aux[k] = v[i]
                        i += 1; k += 1
                else:
                        aux[k] = v[j]
                        j += 1; k += 1

        #termina de copiar sub-vet 1, se for o caso
        while i<=meio:
                aux[k] = v[i]
                i += 1; k += 1

        #termina de copiar sub-vet 2, se for o caso
        while j<=fim:
                aux[k] = v[j]
                j += 1; k += 1
        #copia vetor ordenado aux para v
        for i in range(ini, fim+1):
                v[i] = aux [i]
```

Merge-Sort

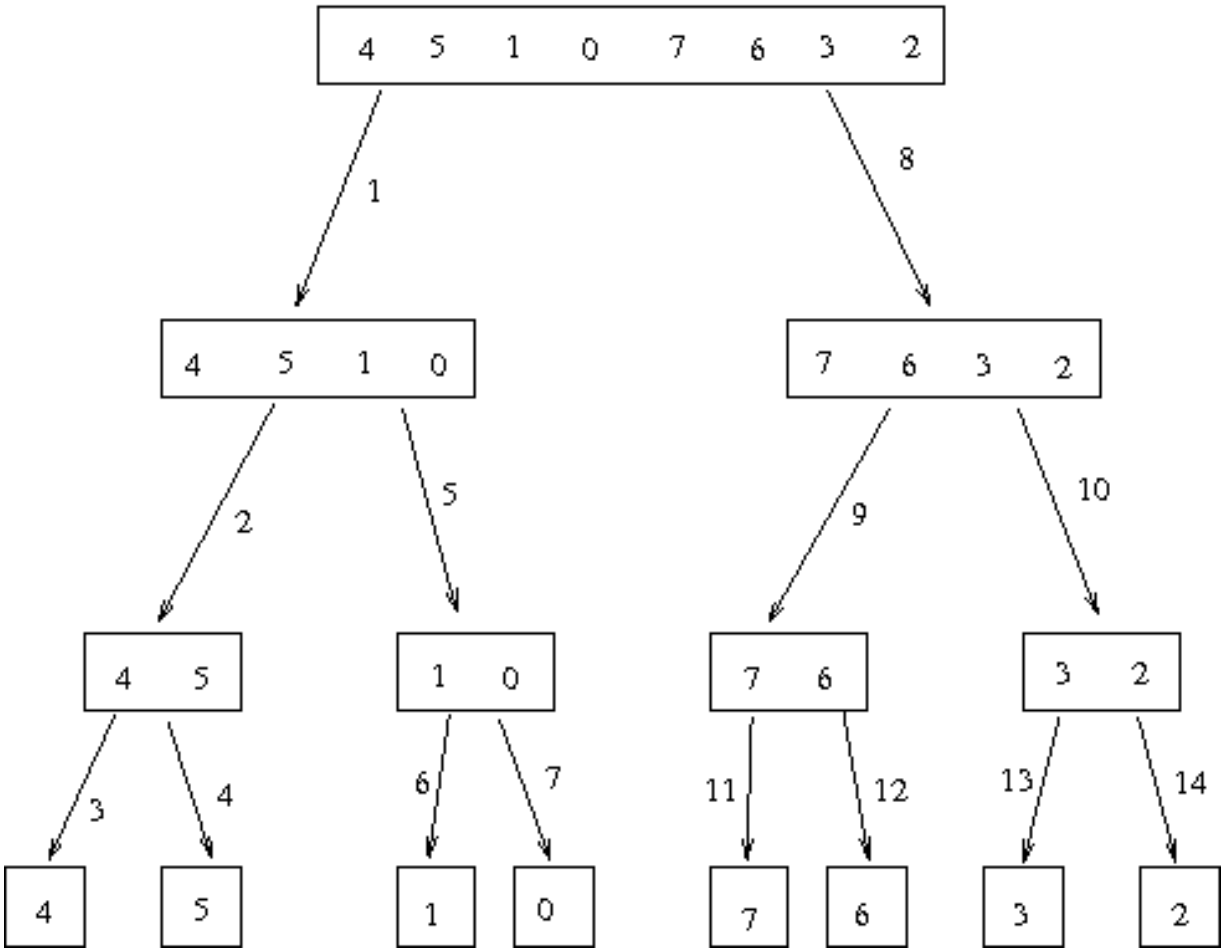
- O merge-sort resolve de forma recursiva dois sub-problemas, cada um contendo uma metade do vetor original.
- **No caso base** temos apenas um elemento e nada precisa ser feito.
- **No caso geral** resolvemos a ordenação dos sub-vetores recursivamente e depois chamamos a função **intercala** para obter o vetor inteiro ordenado.

Merge-Sort

```
def mergeSort(v, ini, fim, aux):
        if ini < fim:
                meio = (ini + fim)//2
                mergeSort(v, ini, meio, aux)
                mergeSort(v, meio+1, fim, aux)
                intercala2(v, ini, meio, fim, aux)
```

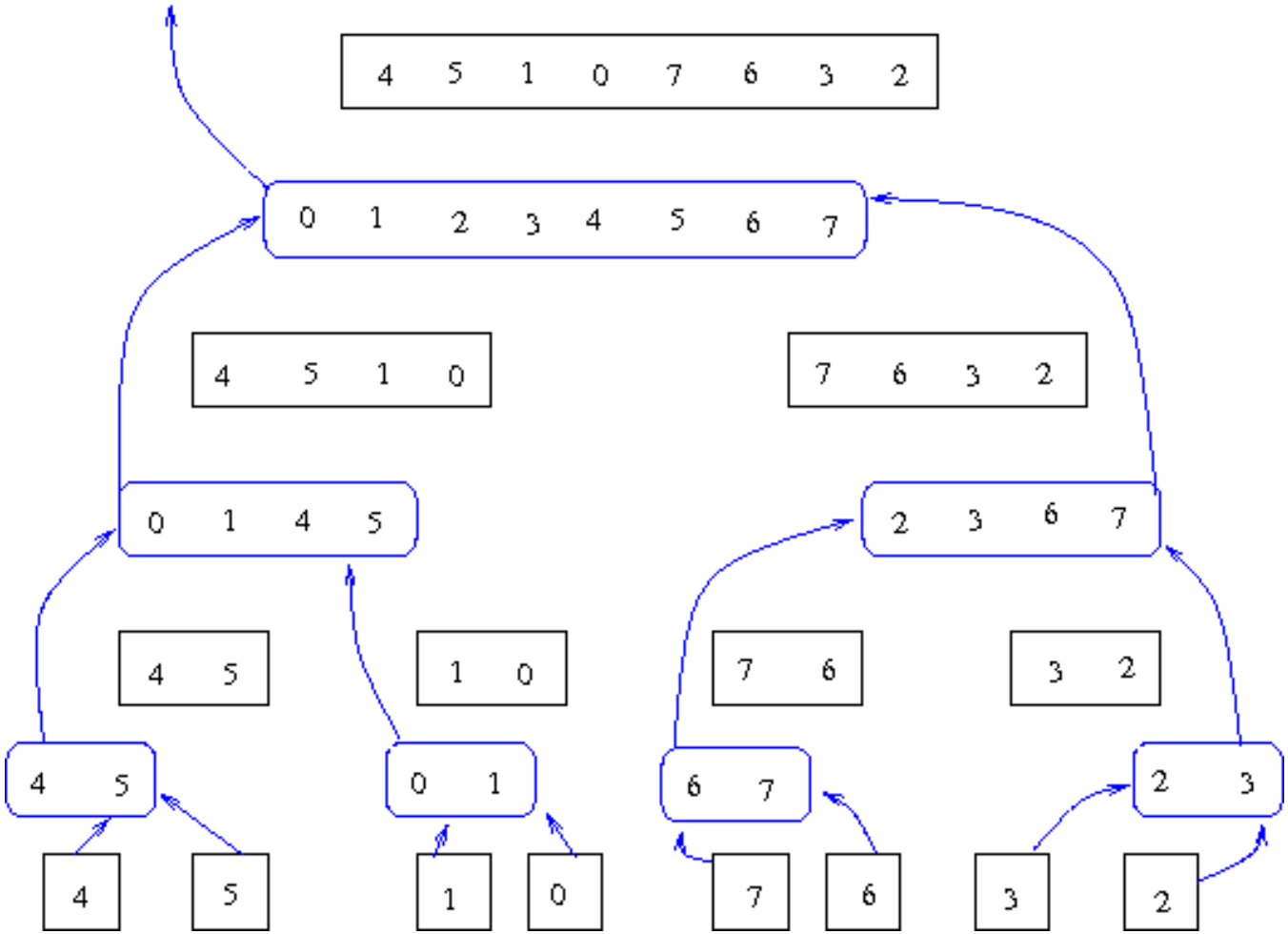
Merge-Sort

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 vetores, **v** a ser ordenado e **aux** do mesmo tamanho de **v**.
- Somente estes dois vetores existirão durante todas as chamadas recursivas.

```
def main():
    tam = 13
    v = [random.randint(0, 2*tam) for i in range(tam)]
    aux = [0 for i in range(tam)]
    print(v)
    mergeSort(v, 0, tam-1, aux)
    print(v)
```

Merge-Sort: Eficiência

Considere um vetor v de n elementos a ser ordenado:

- O Merge-Sort faz um número de operações proporcional a $\approx n \log n$ no pior caso.
- Os algoritmos vistos anteriormente, em um pior caso, fazem um número de operações proporcional a $\approx n^2$.
- Considerando um vetor aleatório de 1 Milhão de elementos (10^6), na média o Merge-Sort executa $\approx 10^6 \log 10^6 = 19.94 \cdot 10^6$ instruções.
- Para este mesmo vetor, os algoritmos anteriores executam na média 1 Trilhão (10^{12}) de instruções!

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.

Exercícios

1. 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.
2. Note que a solução com 2 laços encaixados tem tempo proporcional a $\approx n^2$. A sua solução deve ter tempo proporcional a $\approx n \log n$.

Exercícios

Considere a seguinte variação do problema das torres de Hanoi. O objetivo continua sendo em levar os n discos de A para C com as restrições (1) Somente um disco e que está no topo pode ser movimentado por vez, e (2) um disco só pode ser posicionado sobre um disco de maior tamanho, mas agora adicionamos a restrição (3) de que não é possível mover um disco diretamente de A para C (ou C para A), ou seja todos os movimentos requerem usar a estaca intermediária B. Escreva um algoritmo que gera a solução para este problema. O protótipo da função é

```
#chamada inicial -> hanoi(n, 'A', 'C', 'B');
def hanoi(n, ini, fim, inter);
```