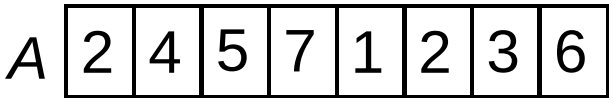
Mergesort

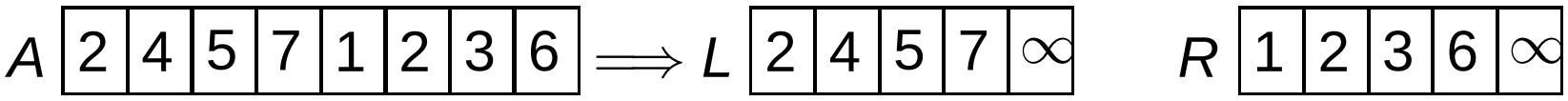
Letícia Rodrigues Bueno

UFABC

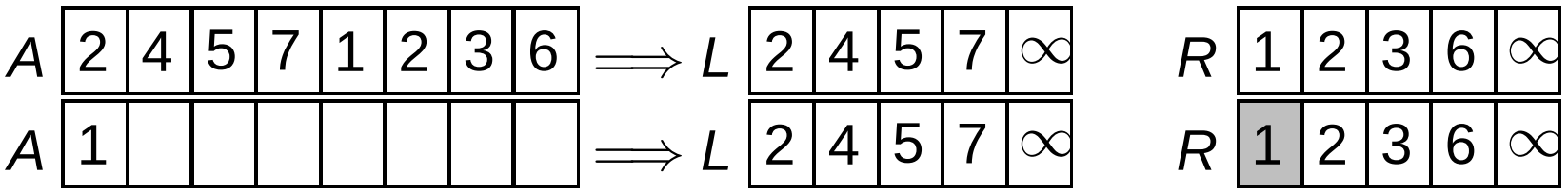
Intercalando duas listas ordenadas



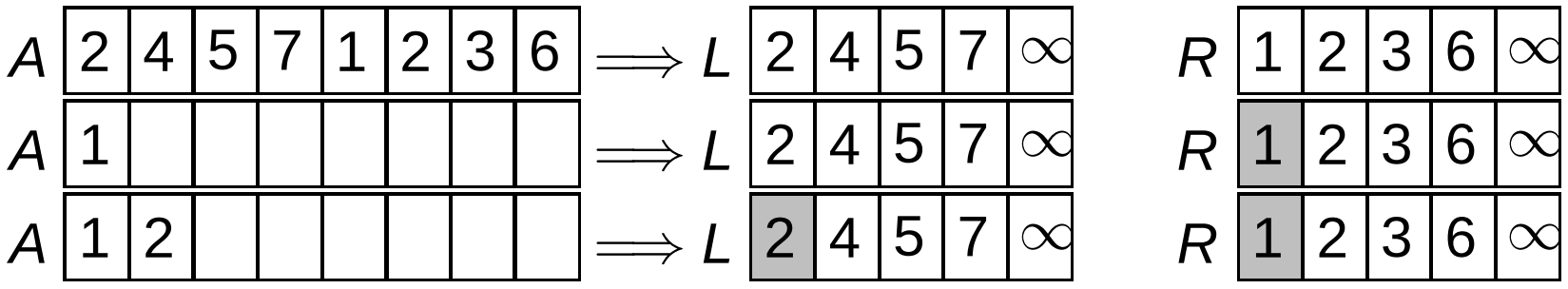
Intercalando duas listas ordenadas



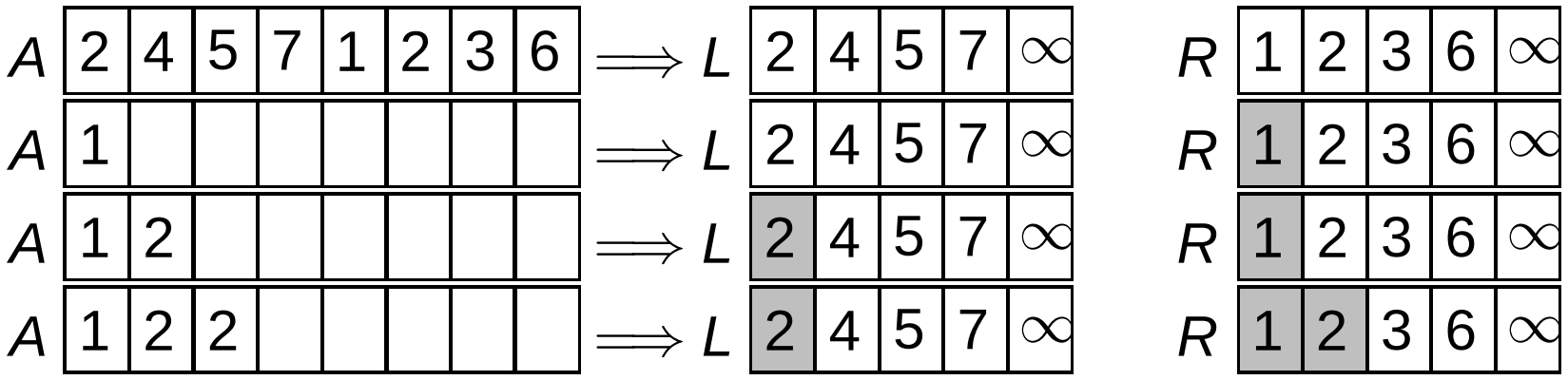
Intercalando duas listas ordenadas



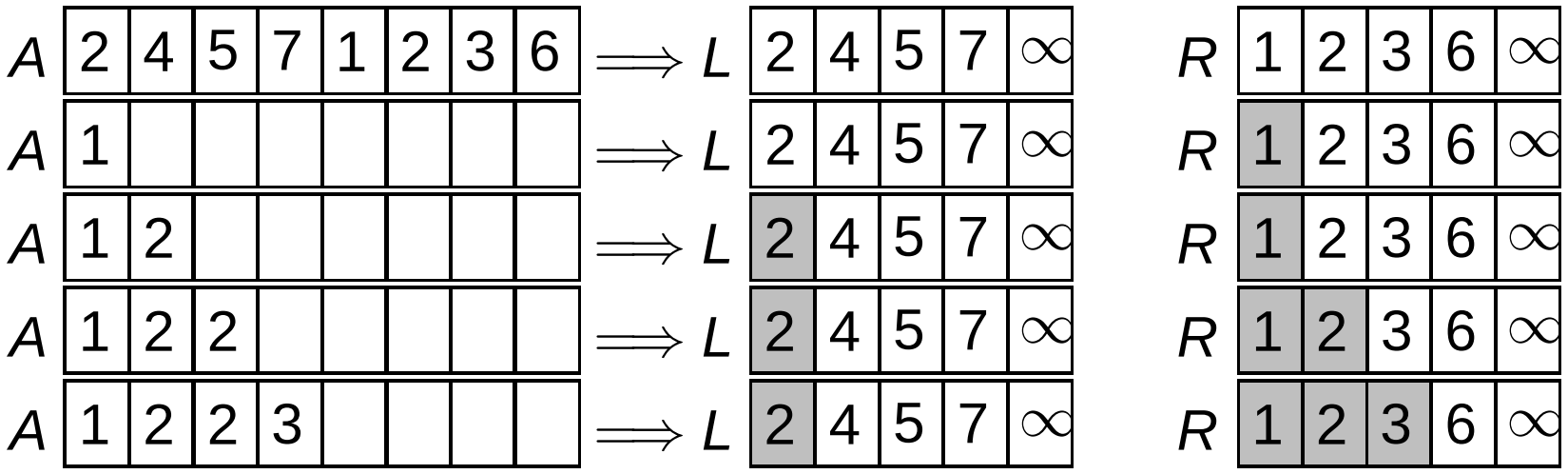
Intercalando duas listas ordenadas



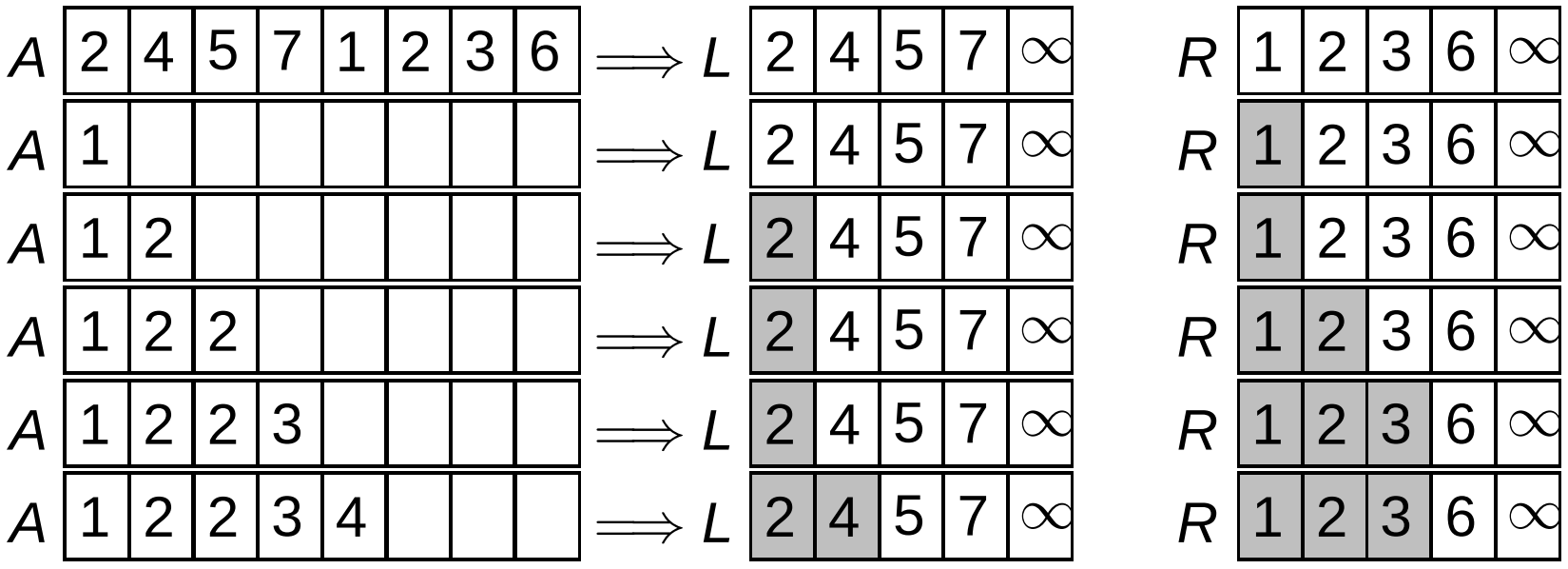
Intercalando duas listas ordenadas



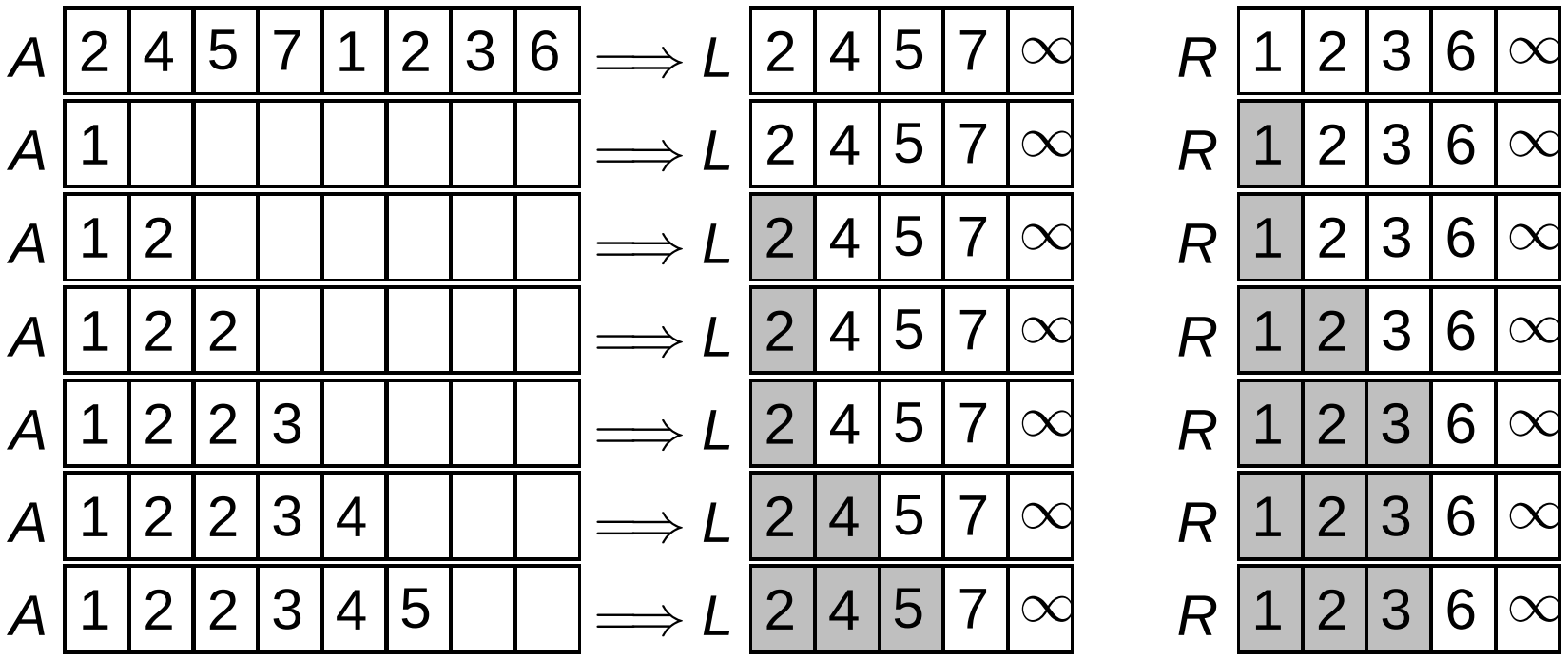
Intercalando duas listas ordenadas



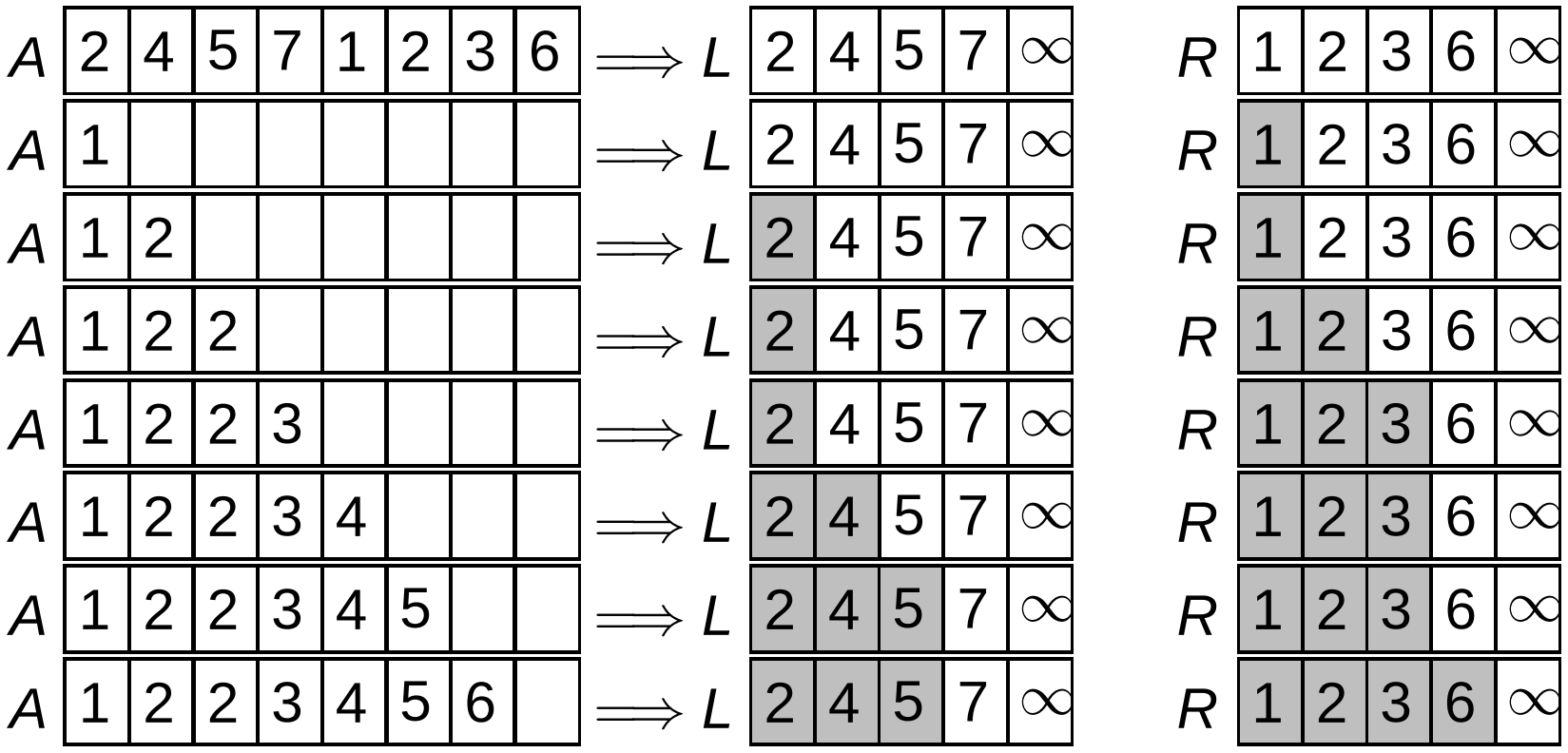
Intercalando duas listas ordenadas



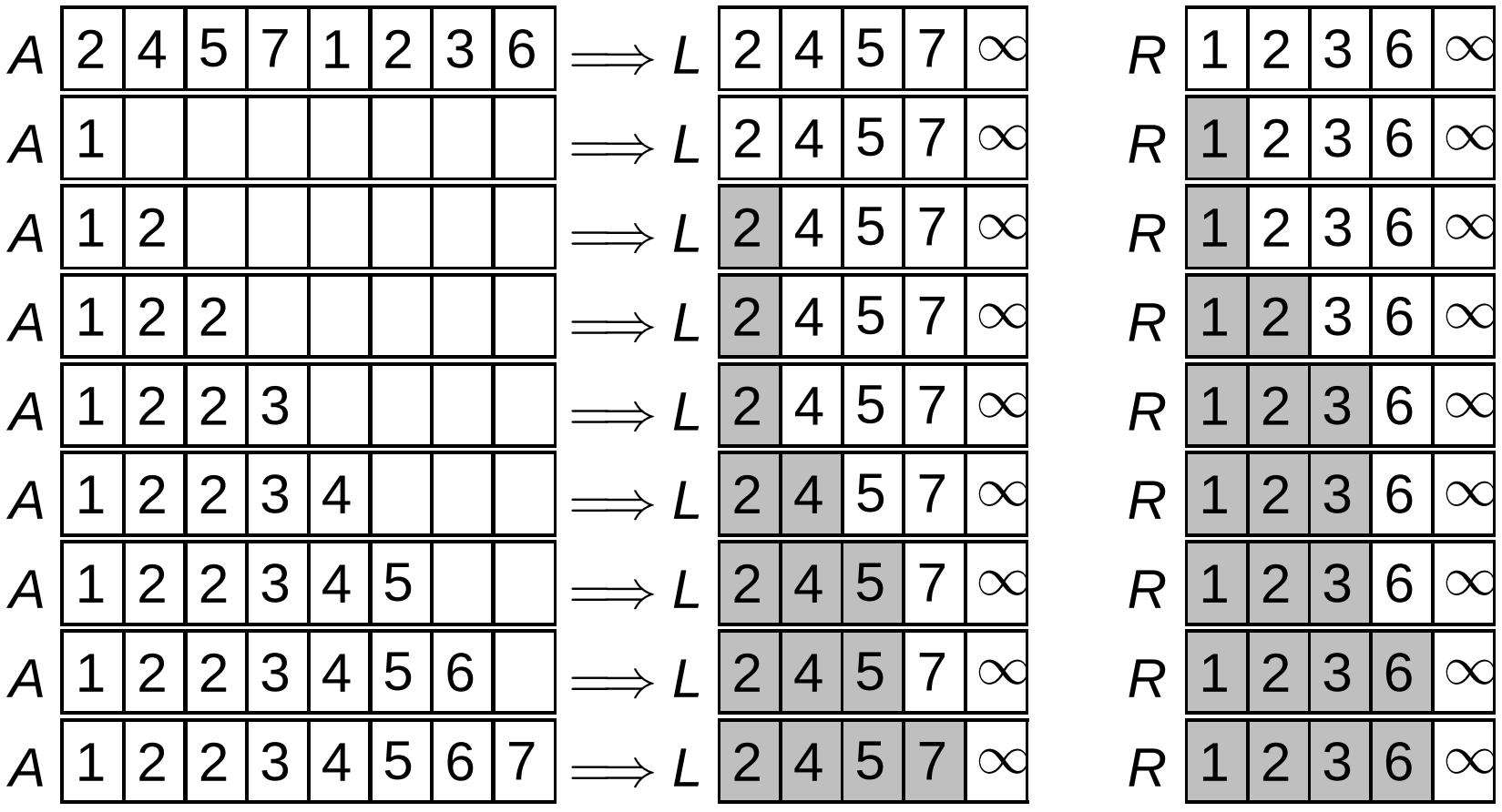
Intercalando duas listas ordenadas



Intercalando duas listas ordenadas



Intercalando duas listas ordenadas



Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Intercalando duas listas ordenadas

Análise da complexidade:

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

- Linha 6 executa *p* vezes;

Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

- Linha 6 executa *p* vezes;
- Linha 8 executa *m* vezes;

Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

- Linha 6 executa p vezes;
- Linha 8 executa m vezes;
- Laço na linha 13 executa $p + m$;

Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

- Linha 6 executa p vezes;
- Linha 8 executa m vezes;
- Laço na linha 13 executa $p + m$;
- **Total:**

Intercalando duas listas ordenadas

```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

- Linha 6 executa p vezes;
- Linha 8 executa m vezes;
- Laço na linha 13 executa $p + m$;
- **Total:** $p + m + (p + m) =$

Intercalando duas listas ordenadas

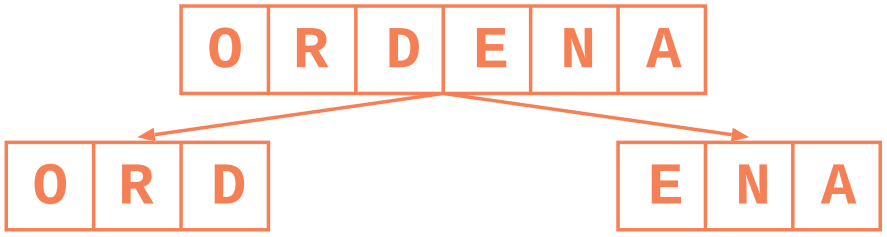
```
 1  merge(A, ini, meio, fim):
 2      p = meio-ini+1
 3      m = fim-meio
 4      criar arranjos L e R
 5      para (i=1; i≤p; i++) faça
 6          L[i]=A[ini+i-1]
 7      para (j=1; j≤m; j++) faça
 8          R[j]=A[meio+j]
 9      L[p+1] = ∞
10      R[m+1] = ∞
11      i = 1
12      j = 1
13      para (k=ini; k≤fim; k++) faça
14          se L[i]≤R[j] então A[k] = L[i]
15              i = i+1
16          senão A[k] = R[j]
17              j = j+1
```

Análise da complexidade:

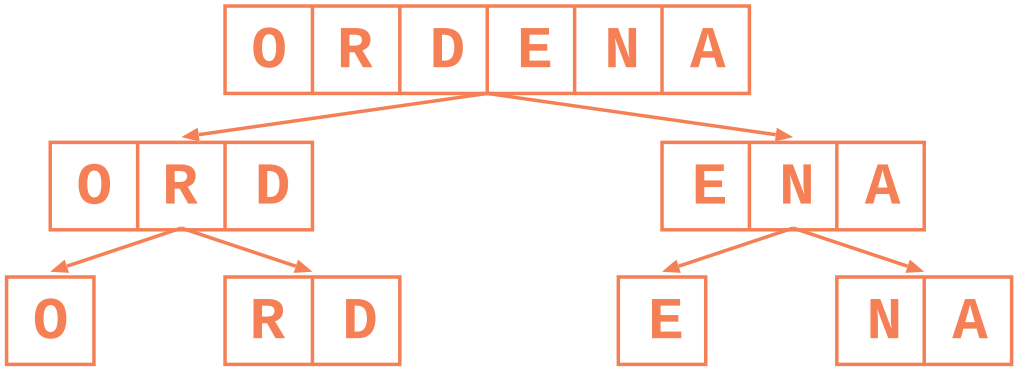
- Linha 6 executa p vezes;
- Linha 8 executa m vezes;
- Laço na linha 13 executa $p + m$;
- **Total:** $p + m + (p + m) =$ $\mathbf{O(p + m)}$

Mergesort - Ordenação por Intercalação

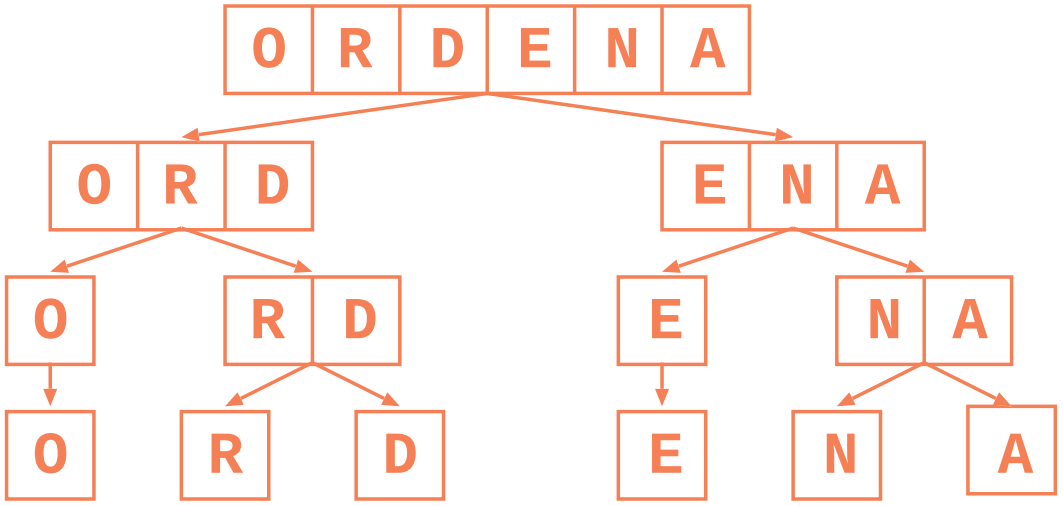
Mergesort - Ordenação por Intercalação



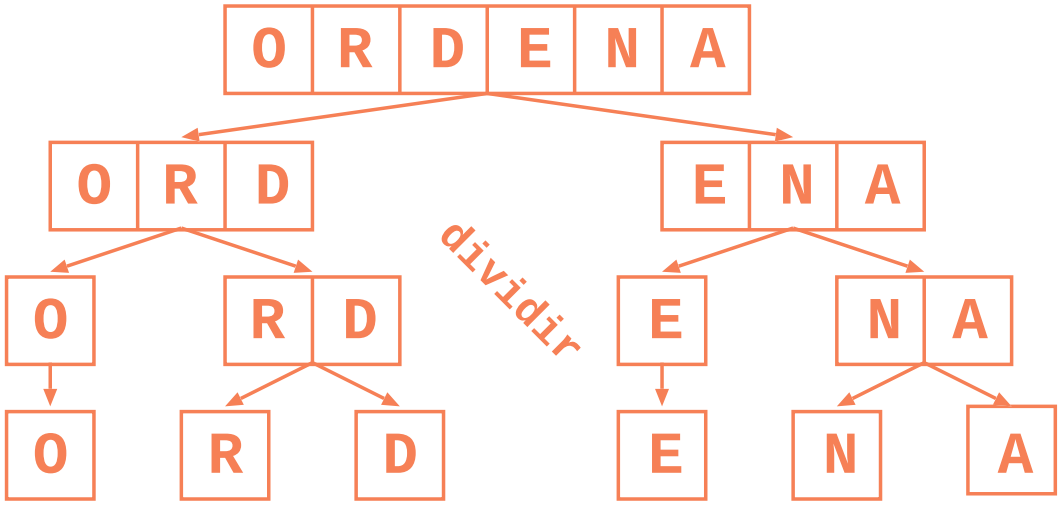
Mergesort - Ordenação por Intercalação



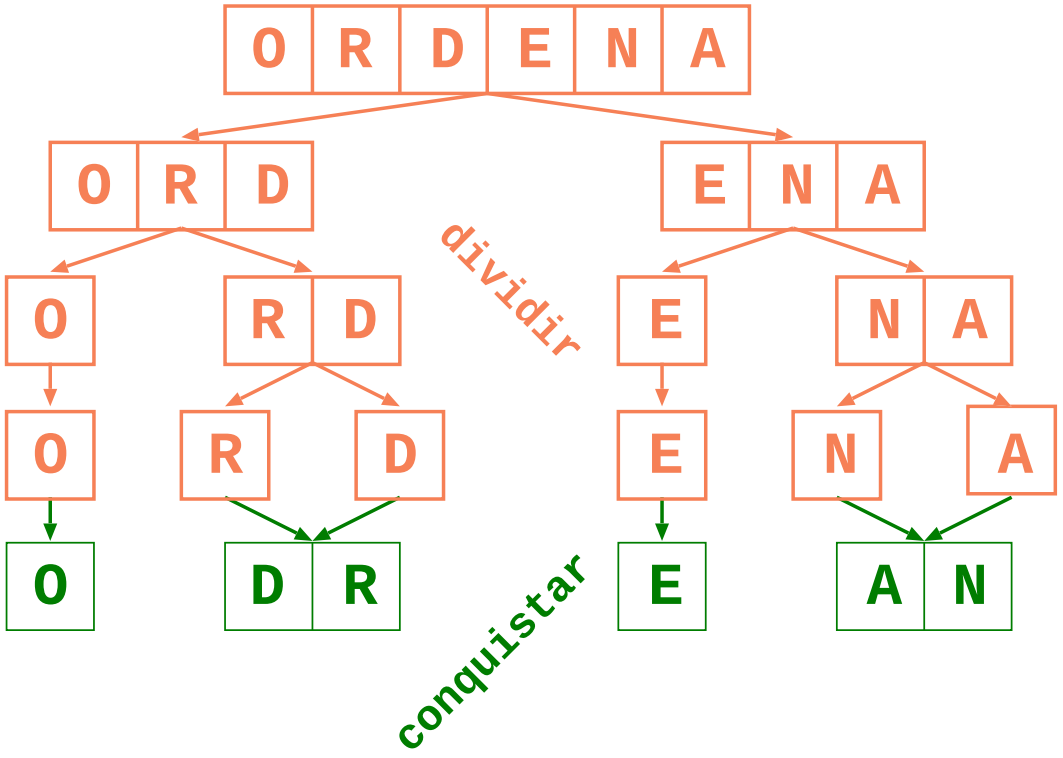
Mergesort - Ordenação por Intercalação



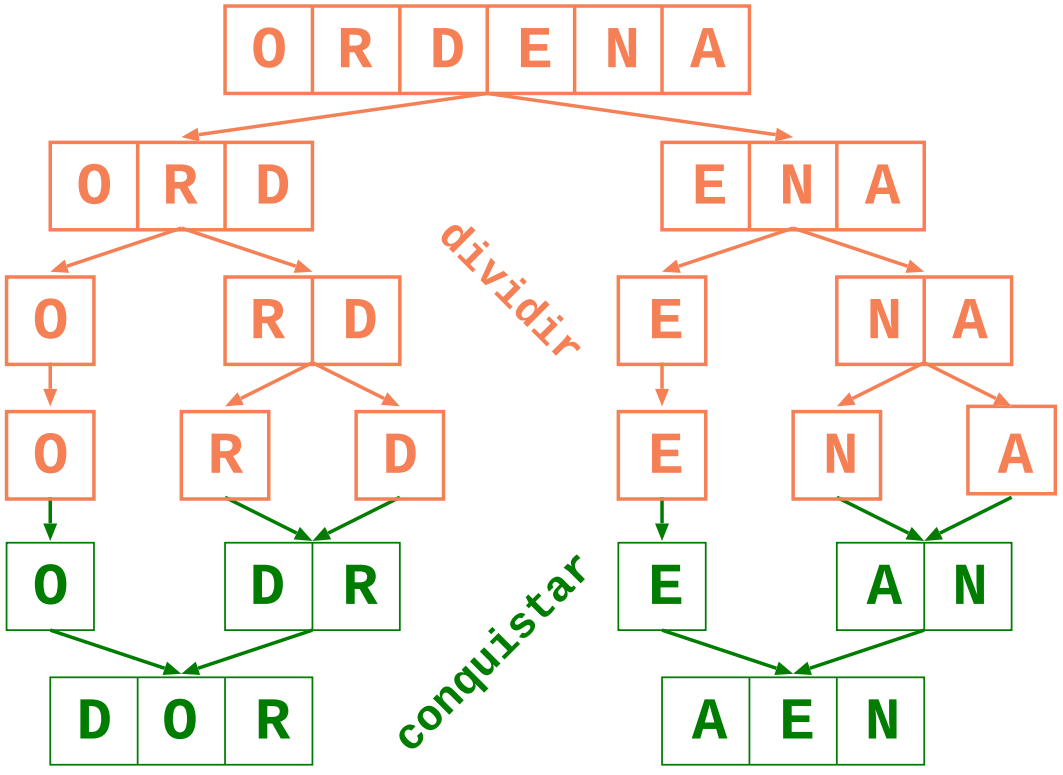
Mergesort - Ordenação por Intercalação



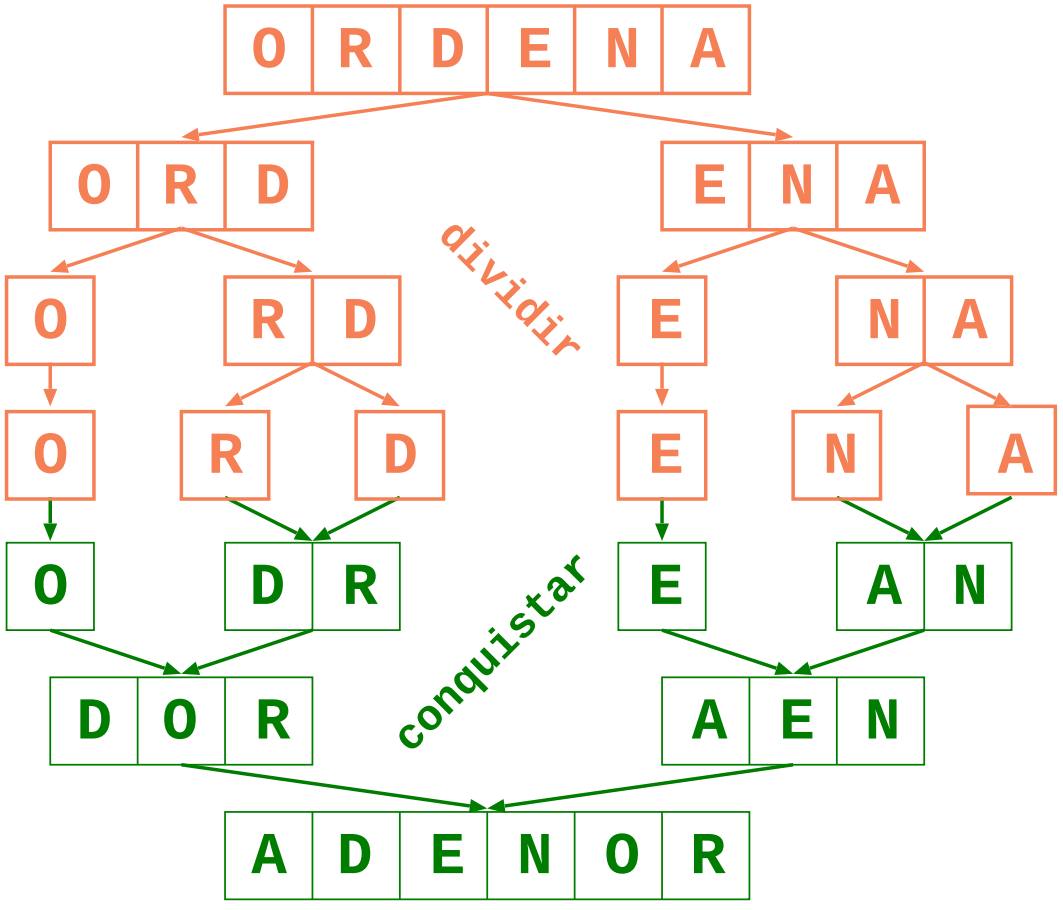
Mergesort - Ordenação por Intercalação



Mergesort - Ordenação por Intercalação



Mergesort - Ordenação por Intercalação



Mergesort - Ordenação por Intercalação

```
1  mergesort(V, ini, fim):
2      se (ini<fim) então
3          meio = ⌊(ini+fim)/2⌋
4          mergesort(V,ini,meio)
5          mergesort(V,meio+1,fim)
6          merge(V,ini,meio,fim)
```

Mergesort - Ordenação por Intercalação

Análise da complexidade:

```
1  mergesort(V, ini, fim):
2      se (ini<fim) então
3          meio = ⌊(ini+fim)/2⌋
4          mergesort(V,ini,meio)
5          mergesort(V,meio+1,fim)
6          merge(V,ini,meio,fim)
```

Mergesort - Ordenação por Intercalação

```
1  mergesort(V, ini, fim):
2      se (ini<fim) então
3          meio = ⌊(ini+fim)/2⌋
4          mergesort(V,ini,meio)
5          mergesort(V,meio+1,fim)
6          merge(V,ini,meio,fim)
```

Análise da complexidade:

- Como fazer análise de algoritmos recursivos divisão e conquista?

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

- **Dividir:** dividir problema em a subproblemas, cada um de tamanho n/b, gasta tempo $D(n)$;

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

- **Dividir:** dividir problema em a subproblemas, cada um de tamanho n/b, gasta tempo $D(n)$;
- **Conquistar:** resolver subproblemas recursivamente;

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

- **Dividir:** dividir problema em a subproblemas, cada um de tamanho n/b, gasta tempo $D(n)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções dos subproblemas gasta tempo $C(n)$;

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

- **Dividir:** dividir problema em a subproblemas, cada um de tamanho n/b, gasta tempo $D(n)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções dos subproblemas gasta tempo $C(n)$;
- Tempo $\Theta(1)$ para subproblema “suficientemente” pequeno;

Análise de Algoritmos Recursivos

Para algoritmos recursivos divisão e conquista analisamos tempo $T(n)$ de:

- **Dividir:** dividir problema em a subproblemas, cada um de tamanho n/b, gasta tempo $D(n)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções dos subproblemas gasta tempo $C(n)$;
- Tempo $\Theta(1)$ para subproblema "suficientemente" pequeno;

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

Mergesort - Complexidade

Mergesort - Complexidade

- **Dividir:** $a = 2$ subproblemas, de tamanho $n/b = n/2$, gasta tempo $D(n) = \Theta(1)$;

Mergesort - Complexidade

- **Dividir:** $a = 2$ subproblemas, de tamanho $n/b = n/2$, gasta tempo $D(n) = \Theta(1)$;
- **Conquistar:** resolver subproblemas recursivamente;

Mergesort - Complexidade

- **Dividir:** $a = 2$ subproblemas, de tamanho $n/b = n/2$, gasta tempo $D(n) = \Theta(1)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções (algoritmo merge) gasta tempo $C(n) = \Theta(n)$;

Mergesort - Complexidade

- **Dividir:** $a = 2$ subproblemas, de tamanho $n/b = n/2$, gasta tempo $D(n) = \Theta(1)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções (algoritmo merge) gasta tempo $C(n) = \Theta(n)$;
- Tempo $\Theta(1)$ para $n = 1$;

Mergesort - Complexidade

- **Dividir:** $a = 2$ subproblemas, de tamanho $n/b = n/2$, gasta tempo $D(n) = \Theta(1)$;
- **Conquistar:** resolver subproblemas recursivamente;
- **Combinar:** combinar soluções (algoritmo merge) gasta tempo $C(n) = \Theta(n)$;
- Tempo $\Theta(1)$ para $n = 1$;

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

Mergesort - Resolução da Recorrência

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

Mergesort - Resolução da Recorrência

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

- Pelo Método Mestre, temos $a = b = 2$ e $f(n) = \Theta(n)$;

Mergesort - Resolução da Recorrência

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

- Pelo Método Mestre, temos $a = b = 2$ e $f(n) = \Theta(n)$;
- **Caso 2**: se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;

Mergesort - Resolução da Recorrência

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

- Pelo Método Mestre, temos $a = b = 2$ e $f(n) = \Theta(n)$;
- **Caso 2**: se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
- **Pior caso *Mergesort*:** $\Theta(n \lg n)$;

Mergesort - Resolução da Recorrência

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

- Pelo Método Mestre, temos $a = b = 2$ e $f(n) = \Theta(n)$;
- **Caso 2**: se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
- **Pior caso *Mergesort*:** $\Theta(n \lg n)$;
- **Limite inferior problema ordenação (comparação):** $\Omega(n \log n)$;

Mergesort - Resolução da Recorrência

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

- Pelo Método Mestre, temos $a = b = 2$ e $f(n) = \Theta(n)$;
- **Caso 2**: se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$;
- **Pior caso *Mergesort*:** $\Theta(n \lg n)$;
- **Limite inferior problema ordenação (comparação):** $\Omega(n \log n)$;
- Portanto, *mergesort* é **algoritmo ótimo!**

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *mergesort* é estável? Exemplifique. Se não é estável, como podemos modificá-lo para ser estável?

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *mergesort* é estável? Exemplifique. Se não é estável, como podemos modificá-lo para ser estável?
2. Modifique o *mergesort* para fazer ordenação decrescente.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *mergesort* é estável? Exemplifique. Se não é estável, como podemos modificá-lo para ser estável?
2. Modifique o *mergesort* para fazer ordenação decrescente.
3. Escreva e analise uma versão iterativa do *mergesort*.

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *mergesort* é estável? Exemplifique. Se não é estável, como podemos modificá-lo para ser estável?
2. Modifique o *mergesort* para fazer ordenação decrescente.
3. Escreva e analise uma versão iterativa do *mergesort*.
4. Qual o melhor caso do *mergesort*? Qual seu custo neste caso?

Exercícios

1. Um algoritmo de ordenação é **estável** se preserva ordem relativa de itens com valores idênticos. Responda: *mergesort* é estável? Exemplifique. Se não é estável, como podemos modificá-lo para ser estável?
2. Modifique o *mergesort* para fazer ordenação decrescente.
3. Escreva e analise uma versão iterativa do *mergesort*.
4. Qual o melhor caso do *mergesort*? Qual seu custo neste caso?
5. Escreva uma versão do *mergesort* para ordenar uma lista encadeada. Sua função não deve alocar novas células na memória.

Bibliografia Utilizada

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.

SZWARCFITER, J. L. e MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994.

ZIVIANI, N. Projeto de Algoritmos: com implementações em Pascal e C, 2ª edição, Cengage Learning, 2009.