Merge sort e Tim Sort.
pág 63 até 68 e 369 até 376

PÁG 63 ATÉ 68:

Merge sort:

O Merge sort é um algoritmo para ordenar uma lista de n números naturais em ordem crescente. Primeiramente, a lista de elementos é dividida iterativamente em partes iguais até que cada sublista contenha um elemento e, em seguida, essas sublistas são combinadas para criar uma nova lista em ordem ordenada. Essa abordagem de programação para resolução de problemas é baseada na metodologia dividir e conquistar e enfatiza a necessidade de quebrar um problema em subproblemas menores do mesmo tipo ou forma que o problema original. Esses subproblemas são resolvidos separadamente e, em seguida, os resultados são combinados para obter a solução do problema original.

Nesse caso, dada uma lista de elementos não ordenados, dividimos a lista em duas metades aproximadas. Continuamos dividindo a lista em metades recursivamente.

Após um tempo, a sublista criada como resultado da chamada recursiva conterá apenas um elemento. Nesse ponto, começamos a mesclar as soluções na etapa de conquista ou mesclagem. Esse processo é mostrado na Figura 3.3:

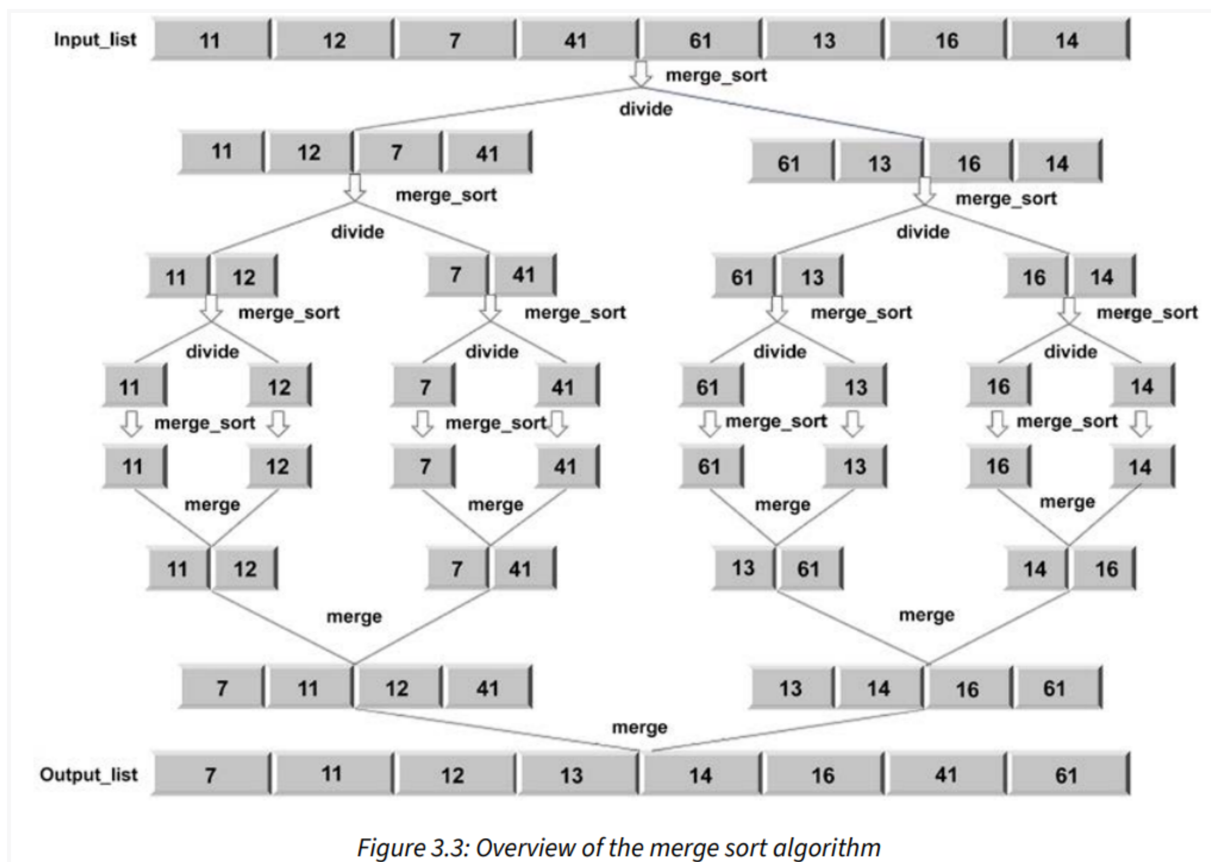

Figure 3.3: Overview of the merge sort algorithm

A implementação do algoritmo merge sort é feita usando principalmente dois métodos, nomeadamente, o método merge_sort, que divide recursivamente a lista. Em seguida, introduzimos o método merge para combinar os resultados:

```
def merge_sort(unsorted_list):
    if len(unsorted_list) == 1:
        return unsorted_list
    mid_point = int(len(unsorted_list)/2)
    first_half = unsorted_list[:mid_point]
    second_half = unsorted_list[mid_point:]

    half_a = merge_sort(first_half)
    half_b = merge_sort(second_half)

    return merge(half_a, half_b)
```

A implementação começa aceitando a lista de elementos não ordenados na função merge_sort. A declaração "if" é usada para estabelecer o caso base, onde, se houver apenas um elemento na lista não ordenada, simplesmente retornamos essa lista novamente. Se houver mais de um elemento na lista, encontramos o ponto médio aproximado usando mid_point = len(unsorted_list)//2. Usando esse ponto médio, dividimos a lista em duas sublistas, a primeira metade e a segunda metade:

```
first_half = unsorted_list[:mid_point]
second_half = unsorted_list[mid_point:]
```

Uma chamada recursiva é feita passando as duas sublistas para a função merge_sort novamente:

```
half_a = merge_sort(first_half)
half_b = merge_sort(second_half)
```

Agora, para a etapa de mesclagem, half_a e half_b estão ordenados. Quando half_a e half_b passarem seus valores, chamamos a função merge, que irá mesclar ou combinar as duas soluções armazenadas em half_a e half_b, que são listas:

```
def merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
```

```
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

A função merge recebe as duas listas que queremos mesclar, first_sublist e second_sublist.

As variáveis i e j são inicializadas em 0 e são usadas como ponteiros para nos dizer onde estamos no processo de mesclagem das duas listas.

A lista final merged_list conterá a lista mesclada.
O loop while começa comparando os elementos em first_sublist e second_sublist:

```
while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
```

A declaração if seleciona o menor dos dois, first_sublist[i] ou second_sublist[j], e o adiciona a merged_list. O índice i ou j é incrementado para refletir onde estamos no processo de mesclagem. O loop while para quando uma das sublistas está vazia.

Pode haver elementos deixados em first_sublist ou second_sublist. Os dois últimos loops while garantem que esses elementos sejam adicionados a merged_list antes que ela seja retornada. A última chamada para merge(half_a, half_b) retornará a lista ordenada. O código a seguir mostra como passar uma matriz para classificar os elementos usando merge sort:

```
a= [11, 12, 7, 41, 61, 13, 16, 14]
print(merge_sort(a))
```

A saída será:

```
[7, 11, 12, 14, 16, 41, 61]
```

Vamos dar uma olhada no algoritmo mesclando as duas sublistas [4, 6, 8] e [5, 7, 11, 40], mostradas na Tabela 3.1. Neste exemplo, inicialmente, as duas sublistas ordenadas são dadas e, em seguida, os primeiros elementos são combinados, e como o primeiro elemento da primeira lista é menor, ele é movido para merge_list. Em seguida, no passo 2, novamente, os elementos iniciais de ambas as listas são combinados, e o elemento menor, que é da segunda lista, é movido para merge_list. O mesmo processo é repetido até que uma das listas fique vazia.

Step	first_sublist	second_sublist	merged_list
0	[4 6 8]	[5 7 11 40]	[]
1	[6 8]	[5 7 11 40]	[4]
2	[6 8]	[7 11 40]	[4 5]
3	[8]	[7 11 40]	[4 5 6]
4	[8]	[11 40]	[4 5 6 7]
5	[ ]	[11 40]	[4 5 6 7 8]
6	[]	[ ]	[4 5 6 7 8 11 40]

Table 3.1: Example of merging two lists

Este processo também pode ser visto na Figura 3.4:

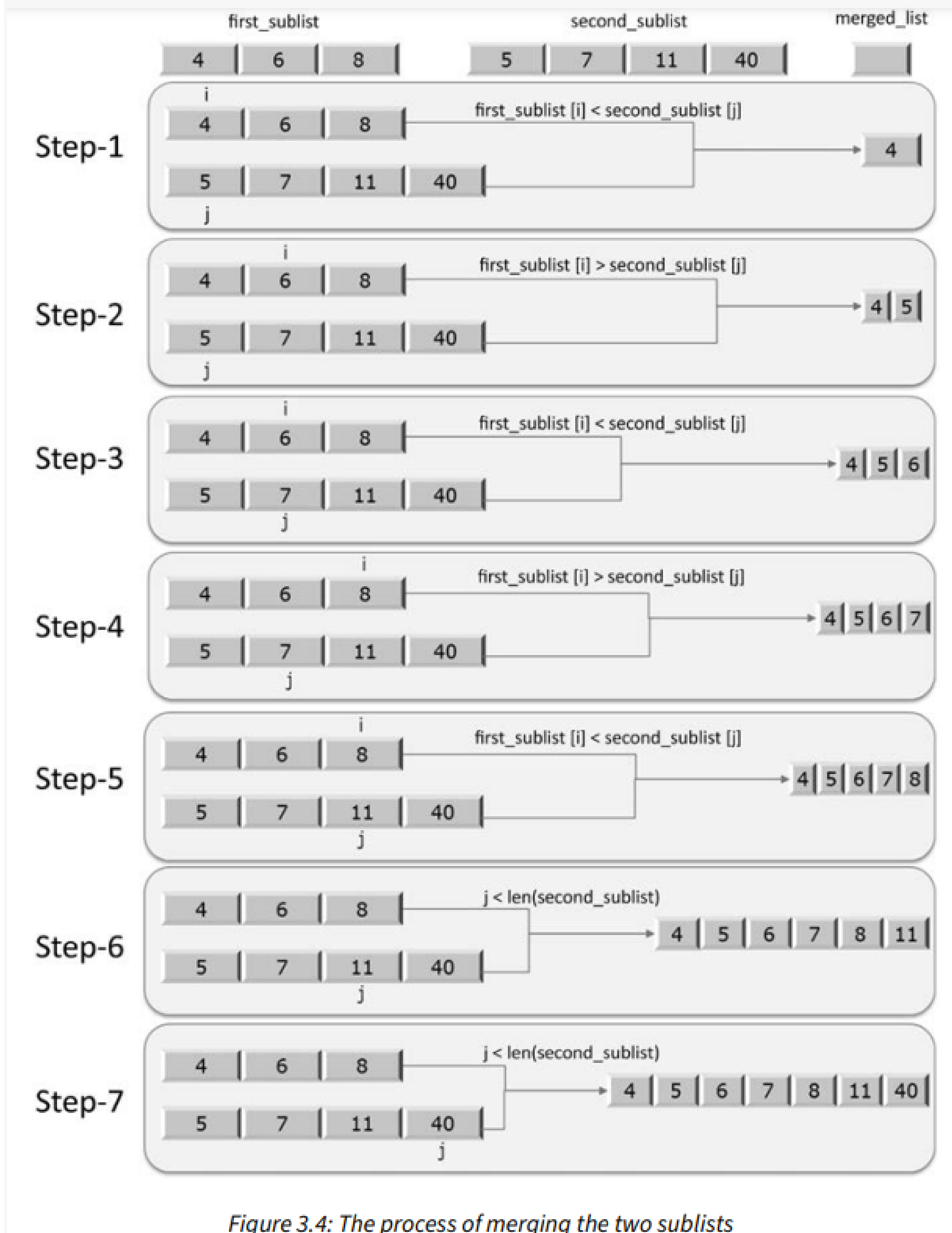

Figure 3.4: The process of merging the two sublists

Depois que uma das listas fica vazia, como após o passo 4 neste exemplo, neste ponto da execução, o terceiro loop while na função

merge entra em ação para mover 11 e 40 para merged_list. A merged_list retornada conterá a lista totalmente ordenada.
A complexidade de tempo de execução no pior caso do merge sort dependerá dos seguintes passos:

1. Em primeiro lugar, o passo de divisão levará um tempo constante, já que ele apenas calcula o ponto médio, o que pode ser feito em O(1) tempo.
2. Em seguida, em cada iteração, dividimos a lista pela metade recursivamente, o que levará O(log n), o que é bastante semelhante ao que vimos no algoritmo de busca binária.
3. Além disso, o passo de combinar/mesclar mescla todos os n elementos na matriz original, o que levará tempo (n).

Assim, o algoritmo merge sort tem uma complexidade de tempo de execução de O(log n) T(n) = O(n) * O(log n) = O(n log n).

Discutimos a técnica de design de algoritmos divide-and-conquer com a ajuda de alguns exemplos. Na próxima seção, discutiremos outra técnica de design de algoritmos: programação dinâmica.

PÁG 369 ATÉ 376:

Timsort algorithm:

O Timsort é usado como o algoritmo de classificação padrão em todas as versões do Python >=2.3. O algoritmo Timsort é um algoritmo ótimo para listas longas do mundo real, baseado na combinação dos algoritmos merge sort e insertion sort. O algoritmo Timsort utiliza o melhor de ambos os algoritmos; o insertion sort funciona melhor quando o array está parcialmente ordenado e seu tamanho é pequeno, e o método merge do algoritmo merge sort funciona rapidamente quando temos que combinar pequenas listas ordenadas. O conceito principal do algoritmo Timsort é que ele usa o algoritmo insertion sort para ordenar blocos pequenos (também conhecidos como chunks) de elementos de dados e, em seguida, usa o algoritmo merge sort para mesclar todos os chunks ordenados. A principal característica do algoritmo Timsort é que ele aproveita elementos de dados já ordenados, conhecidos como "natural runs", que ocorrem com muita frequência em dados do mundo real.

O algoritmo Timsort funciona da seguinte maneira:

1. Primeiramente, dividimos o array de elementos de dados em um número de blocos, também conhecidos como run.
2. Geralmente, usamos 32 ou 64 como o tamanho da run, pois é adequado para o Timsort; no entanto, podemos usar qualquer outro tamanho que possa ser calculado a partir do comprimento do array dado (digamos N). O minrun é o comprimento mínimo de cada run. O tamanho do minrun pode ser calculado seguindo os princípios dados:
    a. O tamanho do minrun não deve ser muito longo, pois usamos o algoritmo insertion sort para ordenar esses blocos

pequenos, que funcionam bem para listas curtas de elementos.

   b. O comprimento da run não deve ser muito curto; nesse caso, resultará em um número maior de runs, o que tornará o algoritmo de mesclagem lento.
   c. Como o merge sort funciona melhor quando temos o número de runs como uma potência de 2, seria bom se o número de runs que computamos como N/minrun fosse uma potência de 2.
3. Por exemplo, se tomarmos um tamanho de run de 32, então o número de runs será (tamanho_do_array/32); se isso for uma potência de 2, o processo de mesclagem será muito eficiente.
4. Classifique cada uma das runs uma por uma usando o algoritmo insertion sort.
5. Mesclar todas as runs ordenadas uma por uma usando o método de mesclagem do algoritmo merge sort.
6. Após cada iteração, dobramos o tamanho do subarray mesclado.

Vamos usar um exemplo para entender o funcionamento do algoritmo Timsort. Digamos que temos o array [4, 6, 3, 9, 2, 8, 7, 5]. Nós o classificamos usando o algoritmo Timsort; aqui, para simplificar, tomamos o tamanho do "run" como 4. Portanto, dividimos o array em dois "runs", "run 1" e "run 2". Em seguida, classificamos "run 1" usando o algoritmo de ordenação por inserção e, em seguida, classificamos "run 2" usando o mesmo algoritmo. Depois de termos todos os "runs" classificados, usamos o método de mesclagem do algoritmo de ordenação por mesclagem para obter a lista completa finalmente classificada. O processo completo é mostrado na Figura 11.24:

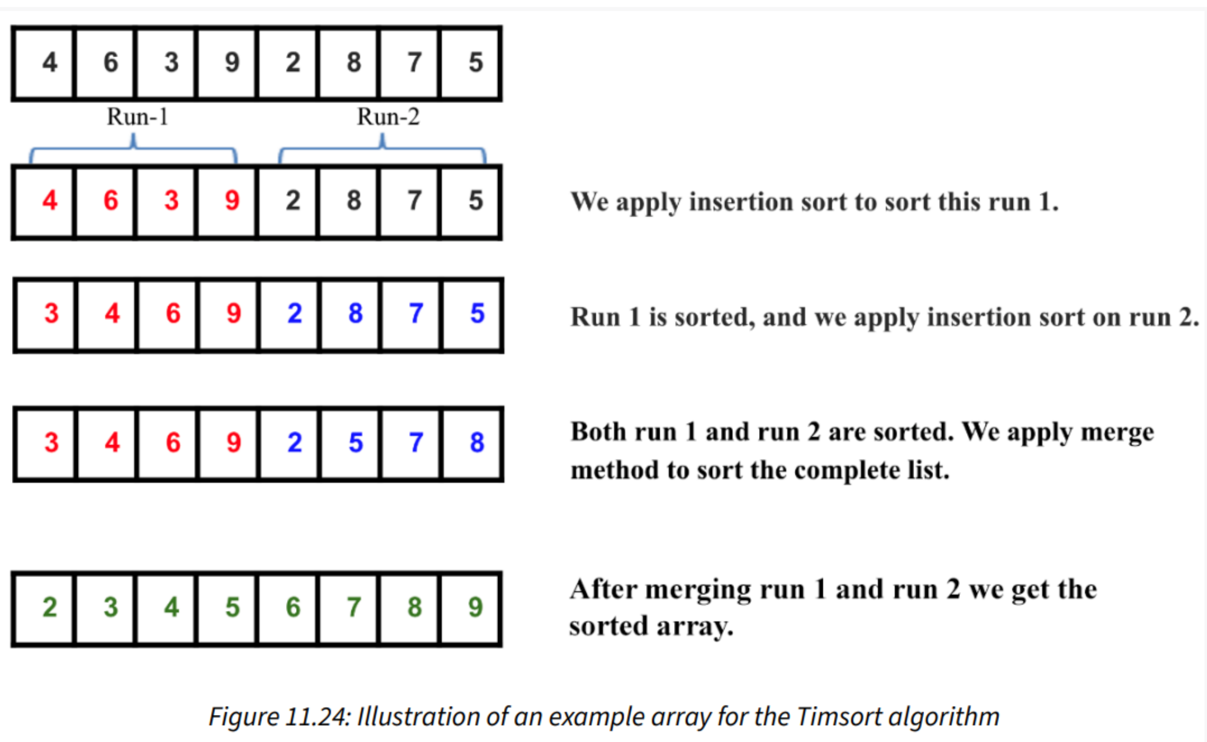

Figure 11.24: Illustration of an example array for the Timsort algorithm

Em seguida, discutimos a implementação do algoritmo Timsort. Em primeiro lugar, implementamos o algoritmo de ordenação por inserção e o método de mesclagem do algoritmo de ordenação por mesclagem. O algoritmo de ordenação por inserção já foi discutido em detalhes em seções anteriores. Para completar, ele é fornecido abaixo novamente:

```
def Insertion_Sort(unsorted_list):
    for index in range(1, len(unsorted_list)):
        search_index = index
        insert_value = unsorted_list[index]
        while search_index > 0 and unsorted_list[search_index-1] > insert_value:
            unsorted_list[search_index] = unsorted_list[search_index-1]
            search_index -= 1
        unsorted_list[search_index] = insert_value
    return unsorted_list
```

Acima, o método de ordenação por inserção é responsável por classificar o "run". Em seguida, apresentamos o método de mesclagem do algoritmo de ordenação por mesclagem; isso foi discutido em detalhes no Capítulo 3, Técnicas e Estratégias de Projeto de Algoritmos.

Esta função "Merge()" é usada para mesclar os "runs" classificados e é definida da seguinte forma:

```
def Merge(first_sublist, second_sublist):
    i = j = 0
    merged_list = []
    while i < len(first_sublist) and j < len(second_sublist):
        if first_sublist[i] < second_sublist[j]:
            merged_list.append(first_sublist[i])
            i += 1
        else:
            merged_list.append(second_sublist[j])
            j += 1
    while i < len(first_sublist):
        merged_list.append(first_sublist[i])
        i += 1
    while j < len(second_sublist):
        merged_list.append(second_sublist[j])
        j += 1
    return merged_list
```

A seguir, discutimos o algoritmo Timsort. Sua implementação é dada abaixo. Vamos entendê-lo pouco a pouco:

```
def Tim_Sort(arr, run):
    for x in range(0, len(arr), run):
        arr[x : x + run] = Insertion_Sort(arr[x : x + run])

    runSize = run
    while runSize < len(arr):
        for x in range(0, len(arr), 2 * runSize):
            arr[x : x + 2 * runSize] = Merge(arr[x : x + runSize], arr[x + runSize: x + 2
            * runSize])

        runSize = runSize * 2
```

Na implementação acima, primeiro passamos dois parâmetros, o array a ser classificado e o tamanho do "run". Em seguida, usamos o algoritmo de ordenação por inserção para classificar os subarrays individuais por tamanho de "run" no trecho de código abaixo:

```
for x in range(0, len(arr), run):
        arr[x : x + run] = Insertion_Sort(arr[x : x + run])
```

No código acima, para a lista de exemplo [4, 6, 3, 9, 2, 8, 7, 5], suponha que o tamanho do "run" seja 2, então teremos um total de quatro blocos/chunks/runs, e depois de sair deste loop, o array ficará assim: [4, 6, 3, 9, 2, 8, 5, 7], indicando que todos os "runs" de tamanho 2 estão classificados. Depois disso, inicializamos o tamanho do "run" e iteramos até que o tamanho do "run" se torne igual ao comprimento do array. Portanto, usamos o método de mesclagem para combinar as pequenas listas classificadas:

```
runSize = run
while runSize < len(arr):
    for x in range(0, len(arr), 2 * runSize):
        arr[x : x + 2 * runSize] = Merge(arr[x : x + runSize], arr[x + runSize: x + 2
        * runSize])

    runSize = runSize * 2
```

No código acima, o loop for está usando a função Merge para mesclar as execuções de tamanho runSize.
Para o exemplo acima, o runSize é 2. Na primeira iteração, ele irá mesclar a execução esquerda do índice (0 a 1) e a execução direita do índice (2 a 3) para formar um array classificado do índice (0 a 3), e o array se tornará [3, 4, 6, 9, 2, 8, 5, 7].

Além disso, na segunda iteração, ele irá mesclar a execução esquerda do índice (4 a 5) e a execução direita do índice (6 a 7) para formar uma execução classificada do índice (4 a 7). Após a segunda iteração, o loop for terminará e o array se tornará [3, 4, 6, 9, 2, 5, 7, 8], o que indica que o array foi classificado do índice (0 a 3) e (4 a 7).

Agora, atualizamos o tamanho da execução como 2 * runSize e repetimos o mesmo processo para o tamanho de execução atualizado. Agora, o tamanho de execução é 4. Na primeira iteração, ele irá mesclar a execução esquerda (índice 0 a 3) e a execução direita (índice 4 a 7) para formar um array classificado do índice (0 a 7) e depois disso o loop for terminará e o array se tornará [2, 3, 4, 5, 6, 7, 8, 9], o que indica que o array foi classificado.

Agora, o tamanho da execução se tornará igual ao comprimento do array, então o loop while terminará e, por fim, teremos o array classificado.

Podemos usar o trecho de código abaixo para criar uma lista e, em seguida, classificar a lista usando o algoritmo Timsort:

```
arr = [4, 6, 3, 9, 2, 8, 7, 5]
run = 2
Tim_Sort(arr, run)
print(arr)
```

saída:

```
[2,3,4,5,6,7,8,9]
```

O Timsort é muito eficiente para aplicações do mundo real, uma vez que possui uma complexidade de pior caso de O(n log n). O Timsort é a melhor escolha para ordenação, mesmo se o comprimento da lista

fornecida for curto. Nesse caso, ele usa o algoritmo de ordenação por inserção, que é muito rápido para listas menores, e o algoritmo Timsort funciona rápido para listas longas devido ao método de mesclagem; portanto, o algoritmo Timsort é uma boa escolha para ordenação devido à sua adaptabilidade para ordenar matrizes de qualquer tamanho em uso do mundo real.

Uma comparação das complexidades de diferentes algoritmos de ordenação é dada na tabela a seguir:

Algorithm	worst-case	average-case	best-case
Bubble sort	O(n2)	O(n2)	O(n)
Insertion sort	O(n2)	O(n2)	O(n)
Selection sort	O(n2)	O(n2)	O(n2)
Quicksort	O(n2)	O(n log n)	O(n log n)
Timsort	O(n log n)	O(n log n)	O(n)

Table 11.1: Comparing the complexity of different sorting algorithms

Resumo:

Neste capítulo, exploramos algoritmos de ordenação importantes e populares que são muito úteis para muitas aplicações do mundo real. Discutimos os algoritmos de ordenação por bolha, inserção, seleção, quicksort e Timsort, juntamente com a explicação de sua implementação em Python. Em geral, o algoritmo quicksort tem um desempenho melhor do que os outros algoritmos de ordenação, e o algoritmo Timsort é a melhor escolha para usar em aplicações do mundo real.

No próximo capítulo, discutiremos algoritmos de seleção.

Exercícios:

1. Se uma matriz arr = {55, 42, 4, 31} é dada e a ordenação por bolha é usada para ordenar os elementos da matriz, quantas iterações serão necessárias para ordenar a matriz?

a. 3

b. 2

c. 1

d. 0

2. Qual é a complexidade de pior caso do algoritmo de ordenação por bolha?

a. O(n log n)

b. O(log n)

c. O(n)

d. O(n2)

3. Aplicar o quicksort à sequência (56, 89, 23, 99, 45, 12, 66, 78, 34). Qual é a sequência após a primeira fase e qual é o pivô o primeiro elemento?

a. 45, 23, 12, 34, 56, 99, 66, 78, 89

b. 34, 12, 23, 45, 56, 99, 66, 78, 89

c. 12, 45, 23, 34, 56, 89, 78, 66, 99

d. 34, 12, 23, 45, 99, 66, 89, 78, 56

4. Quicksort é um algoritmo de ___________

a. Greedy algorithm

b. Algoritmo de divisão e conquista

c. Algoritmo de programação dinâmica

d. Algoritmo de backtracking

5. Considere uma situação em que a operação de troca é muito custosa. Qual dos seguintes algoritmos de ordenação deve ser usado para minimizar o número de operações de troca?

a. Heap sort

b. Selection sort

c. Insertion sort

d. Merge sort

6. Se o array de entrada A = {15, 9, 33, 35, 100, 95, 13, 11, 2, 13} for dado, usando selection sort, qual seria a ordem do array após a quinta troca? (Observação: isso conta independentemente de trocarem de lugar ou permanecerem na mesma posição.)

a. 2, 9, 11, 13, 13, 95, 35, 33, 15, 100

b. 2, 9, 11, 13, 13, 15, 35, 33, 95, 100

c. 35, 100, 95, 2, 9, 11, 13, 33, 15, 13

d. 11, 13, 9, 2, 100, 95, 35, 33, 13, 13

7. Qual será o número de iterações para ordenar os elementos {44, 21, 61, 6, 13, 1} usando insertion sort?

a. 6

b. 5

c. 7

d. 1

8. Como os elementos do array A= [35, 7, 64, 52, 32, 22] ficarão após a segunda iteração, se os elementos forem ordenados usando insertion sort?

a. 7, 22, 32, 35, 52, 64

b. 7, 32, 35, 52, 64, 22

c. 7, 35, 52, 64, 32, 22

d. 7, 35, 64, 52, 32, 22