Ordenação. cont. pág 356 até 369

Selection sort algorithm:

Outro algoritmo de ordenação popular é o selection sort. O algoritmo selection sort começa encontrando o menor elemento na lista e o troca com os dados armazenados na primeira posição da lista. Assim, ele ordena a sublista até o primeiro elemento. Esse processo é repetido por (n-1) vezes para ordenar n itens.

Em seguida, o segundo menor elemento, que é o menor elemento na lista restante, é identificado e trocado com a segunda posição na lista. Isso torna os dois primeiros elementos ordenados. O processo é repetido, e o menor elemento restante na lista é trocado com o elemento no terceiro índice da lista. Isso significa que os primeiros três elementos estão agora ordenados.

Vamos ver um exemplo para entender como o algoritmo funciona. Vamos ordenar a seguinte lista de quatro elementos {15, 12, 65, 10, 7}, como mostrado na Figura 11.14, juntamente com suas posições de índice usando o algoritmo selection sort:

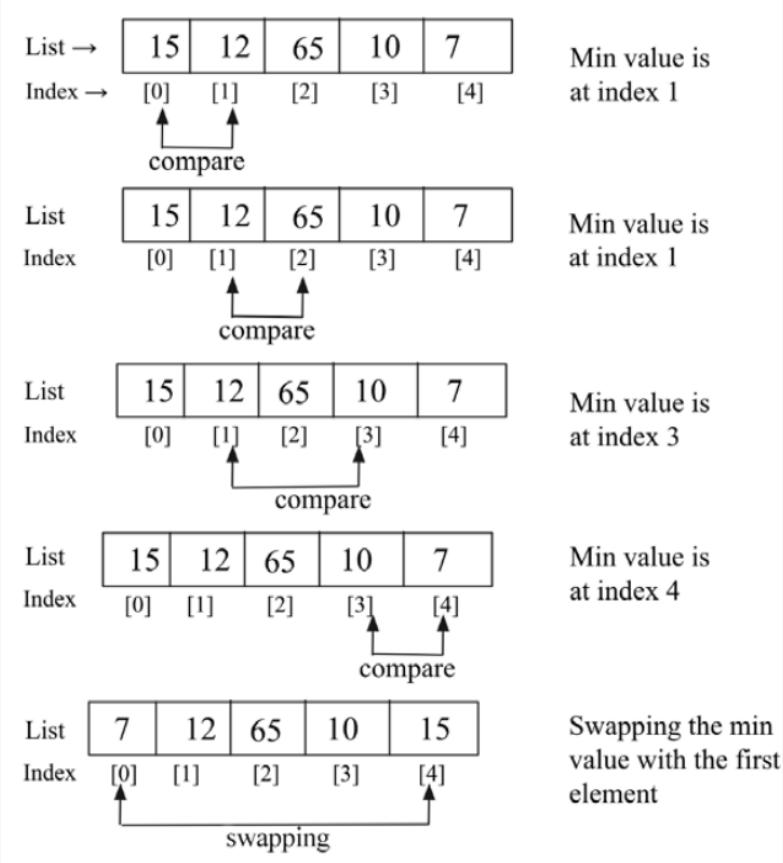

Figure 11.14: Demonstration of the first iteration of the selection sort

Na primeira iteração do selection sort, começamos no índice 0, procuramos pelo menor item na lista e quando o menor elemento é encontrado, ele é trocado com o primeiro elemento de dados da lista no índice 0. Simplesmente repetimos esse processo até que a lista esteja completamente ordenada. Após a primeira iteração, o menor elemento será colocado na primeira posição na lista.

Em seguida, começamos a partir do segundo elemento da lista na posição de índice 1 e procuramos o menor elemento na lista de dados da posição de índice 1 até o comprimento da lista. Assim que encontramos o menor elemento dessa lista restante de elementos, trocamos esse elemento com o segundo elemento da lista. O processo passo a passo da segunda iteração do selection sort é mostrado na Figura 11.15:

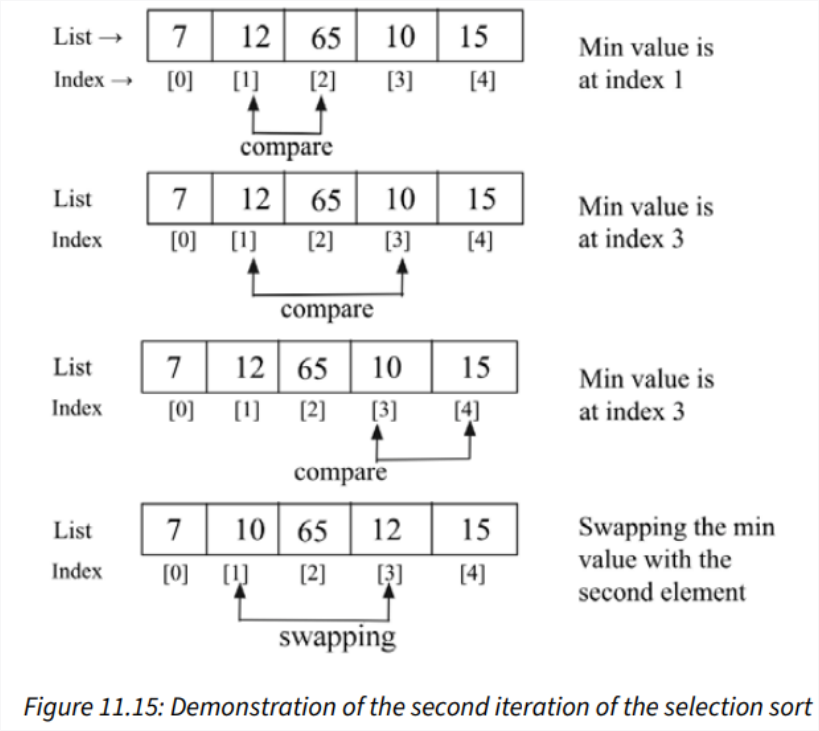

Figure 11.15: Demonstration of the second iteration of the selection sort

Na próxima iteração, descobrimos o menor elemento na lista restante nas posições de índice 2 a 4 e trocamos o menor elemento de dados com o elemento de dados na posição 2 na segunda iteração. Seguimos o mesmo processo até ordenarmos a lista completa.

A seguir está uma implementação do algoritmo selection sort. O argumento para a função é a lista não ordenada de itens que queremos colocar em ordem crescente de seus valores:

```
def selection_sort(unsorted_list):
    size_of_list = len(unsorted_list)
    for i in range(size_of_list):
        small = i
        for j in range(i+1, size_of_list):
            if unsorted_list[j] < unsorted_list[small]:
                small = j
        temp = unsorted_list[i]
        unsorted_list[i] = unsorted_list[small]
        unsorted_list[small] = temp
```

No código acima do selection sort, o algoritmo começa com o loop externo para percorrer a lista, começando do índice 0 até o tamanho_da_lista. Como passamos tamanho_da_lista para o método range, ele produzirá uma sequência de 0 até tamanho_da_lista-1.

Em seguida, declaramos uma variável small, que armazena o índice do menor elemento. Além disso, o loop interno é responsável por percorrer a lista e acompanhamos o índice do menor valor da lista. Quando o índice do menor elemento é encontrado, trocamos esse elemento pela posição correta na lista.
O seguinte código pode ser usado para criar uma lista de elementos e usamos o algoritmo de seleção para classificar a lista:

```
a_list = [3, 2, 35, 4, 32, 94, 5, 7]
print("List before sorting", a_list)
selection_sort(a_list)
print("List after sorting", a_list)
```

O resultado do código acima é o seguinte:

```
List before sorting [3, 2, 35, 4, 32, 94, 5, 7]
List after sorting [2, 3, 4, 5, 7, 32, 35, 94]
```

Na seleção de classificação, (n-1) comparações são necessárias na primeira iteração, e (n-2) comparações são necessárias na segunda iteração, e (n-3) comparações são necessárias na terceira iteração, e assim por diante. Portanto, o número total de comparações necessárias é: (n-1) + (n-2) + (n-3) + ... + 1 = n(n-1) / 2, o que é quase igual a n2. Assim, a pior complexidade de tempo do selection sort é O(n2). A situação de pior caso é quando a lista dada de elementos está ordenada inversamente. O algoritmo de classificação por seleção dá a melhor complexidade de tempo de execução do caso O(n2). O algoritmo de classificação por seleção pode ser usado quando temos uma pequena lista de elementos.

Quicksort algorithm:

O Quicksort é um algoritmo de ordenação eficiente. O algoritmo Quicksort é baseado na classe de algoritmos "dividir e conquistar", similar ao algoritmo Merge Sort, onde dividimos (dividir) um problema em pedaços menores que são mais simples de resolver, e em seguida, os resultados finais são obtidos combinando as saídas dos problemas menores (conquistar).

O conceito por trás do Quicksort é particionar uma lista ou array dado. Para particionar a lista, selecionamos primeiro um elemento de dados da lista, chamado elemento pivô. Podemos escolher qualquer elemento como pivô na lista. No entanto, por questão de simplicidade, tomamos o primeiro elemento do array como elemento pivô. Em seguida, todos os elementos da lista são comparados com este elemento pivô. Ao final da primeira iteração, todos os elementos da lista são organizados de tal maneira que os elementos que são menores que o elemento pivô são organizados à esquerda do pivô e os elementos que são maiores que o elemento pivô são organizados à direita do pivô.

Agora, vamos entender o funcionamento do algoritmo Quicksort com um exemplo. Neste algoritmo, primeiro particionamos a lista de elementos de dados não ordenados em duas sublistas de tal maneira que todos os elementos à esquerda do ponto de partição (também chamado de pivô) devem ser menores que o pivô, e todos os elementos à direita do pivô devem ser maiores. Isso significa que os elementos das duas sublistas estarão desordenados, mas o elemento pivô estará na posição correta na lista completa. Isso é mostrado na Figura 11.16. Portanto, após a primeira iteração do algoritmo Quicksort, o ponto de pivô escolhido é colocado na

lista em sua posição correta, e após a primeira iteração, obtemos duas sublistas desordenadas e seguimos o mesmo processo novamente nessas duas sublistas. Assim, o algoritmo Quicksort particiona a lista em duas partes e aplica recursivamente o algoritmo Quicksort a essas duas sublistas para ordenar toda a lista:

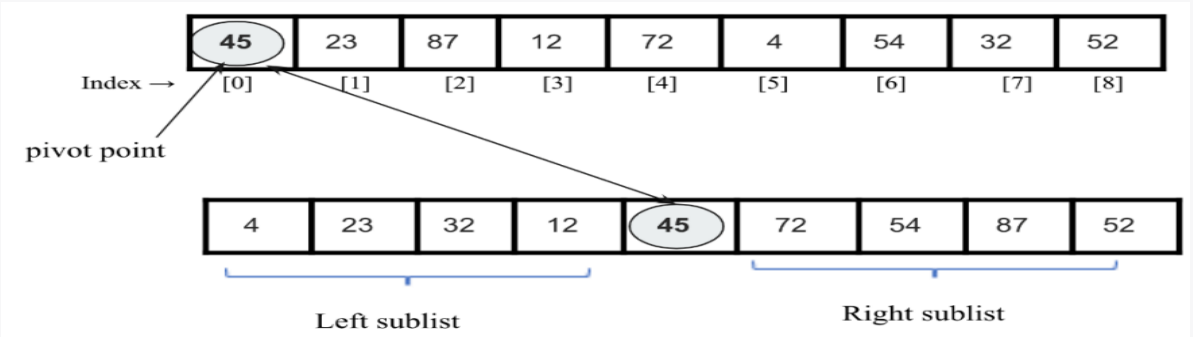

Figure 11.16: Illustration of sublists in quicksort

O algoritmo de classificação rápida (quicksort) funciona da seguinte maneira:

1. Começamos escolhendo um elemento pivô com o qual todos os elementos de dados serão comparados e, ao final da primeira iteração, este elemento pivô será colocado em sua posição correta na lista. Para colocar o elemento pivô em sua posição correta, usamos dois ponteiros, um ponteiro esquerdo e um ponteiro direito. Este processo é da seguinte forma:
    a. O ponteiro esquerdo aponta inicialmente para o valor no índice 1, e o ponteiro direito aponta para o valor no último índice. A ideia principal aqui é mover os itens de dados que estão no lado errado do elemento pivô. Começamos com o ponteiro esquerdo, movendo em uma direção da esquerda para a direita até chegarmos a uma posição em que o item de dados na lista tem um valor maior que o elemento pivô.
    b. Da mesma forma, movemos o ponteiro direito em direção à esquerda até encontrarmos um item de dados menor que o elemento pivô.
    c. Em seguida, trocamos esses dois valores indicados pelos ponteiros esquerdo e direito.
    d. Repetimos o mesmo processo até que ambos os ponteiros se cruzem, ou seja, até que o índice do ponteiro direito indique um valor menor que o do índice do ponteiro esquerdo.

2. Após cada iteração descrita na etapa 1, o elemento pivô será colocado em sua posição correta na lista e a lista original será dividida em duas sub-listas não ordenadas, esquerda e direita. Seguimos o mesmo processo (conforme descrito na etapa 1) para ambas as sub-listas esquerda e direita até que cada uma das sub-listas contenha um único elemento.
3. Finalmente, todos os elementos serão colocados em suas posições corretas, o que dará a lista ordenada como saída.

Vamos pegar um exemplo de uma lista de números, {45, 23, 87, 12, 72, 4, 54, 32, 52}, para entender como o algoritmo quicksort funciona. Vamos assumir que o elemento pivô (também chamado de ponto de pivô) em nossa lista é o primeiro elemento, 45. Movemos o ponteiro esquerdo do índice 1 em direção à direita e paramos quando chegamos ao valor 87, porque (87>45). Em seguida, movemos o ponteiro direito para a esquerda e paramos quando encontramos o valor 32, porque (32<45). Agora, trocamos esses dois valores. Este processo é mostrado na Figura 11.17:

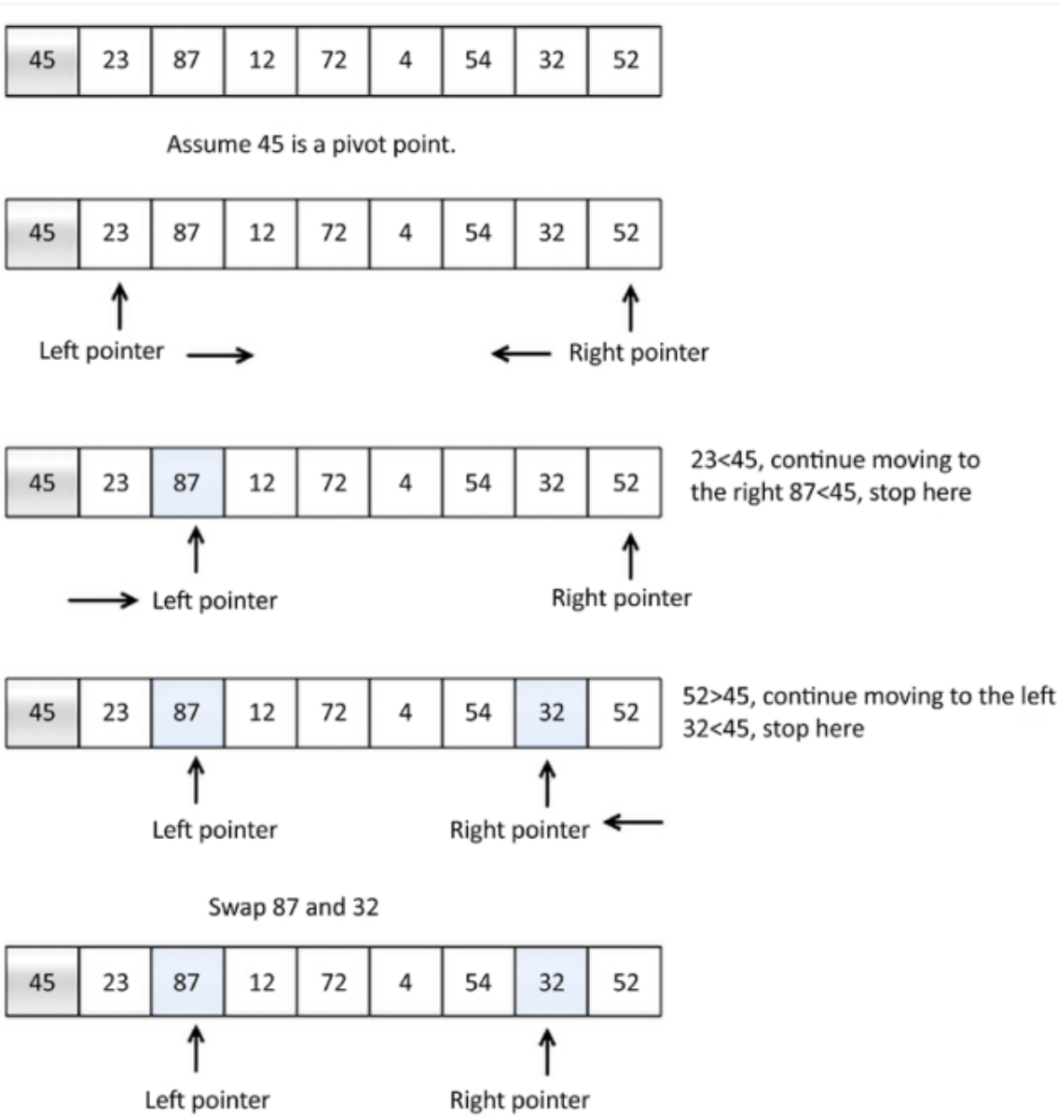

Figure 11.17: An illustrative example of the quicksort algorithm

Depois disso, repetimos o mesmo processo e movemos o ponteiro esquerdo em direção à direita, e paramos quando encontramos o valor 72, porque (72>45). Em seguida, movemos o ponteiro direito em direção à esquerda e paramos quando chegamos ao valor 4, porque (4<45). Agora, trocamos esses dois valores, porque eles estão nos lados errados do valor do pivô. Repetimos o mesmo processo e paramos quando o valor do índice do ponteiro direito se torna menor do que o do índice do ponteiro esquerdo. Aqui, encontramos o valor 4 como o ponto de divisão e o trocamos com o valor do pivô. Isso é mostrado na Figura 11.18:

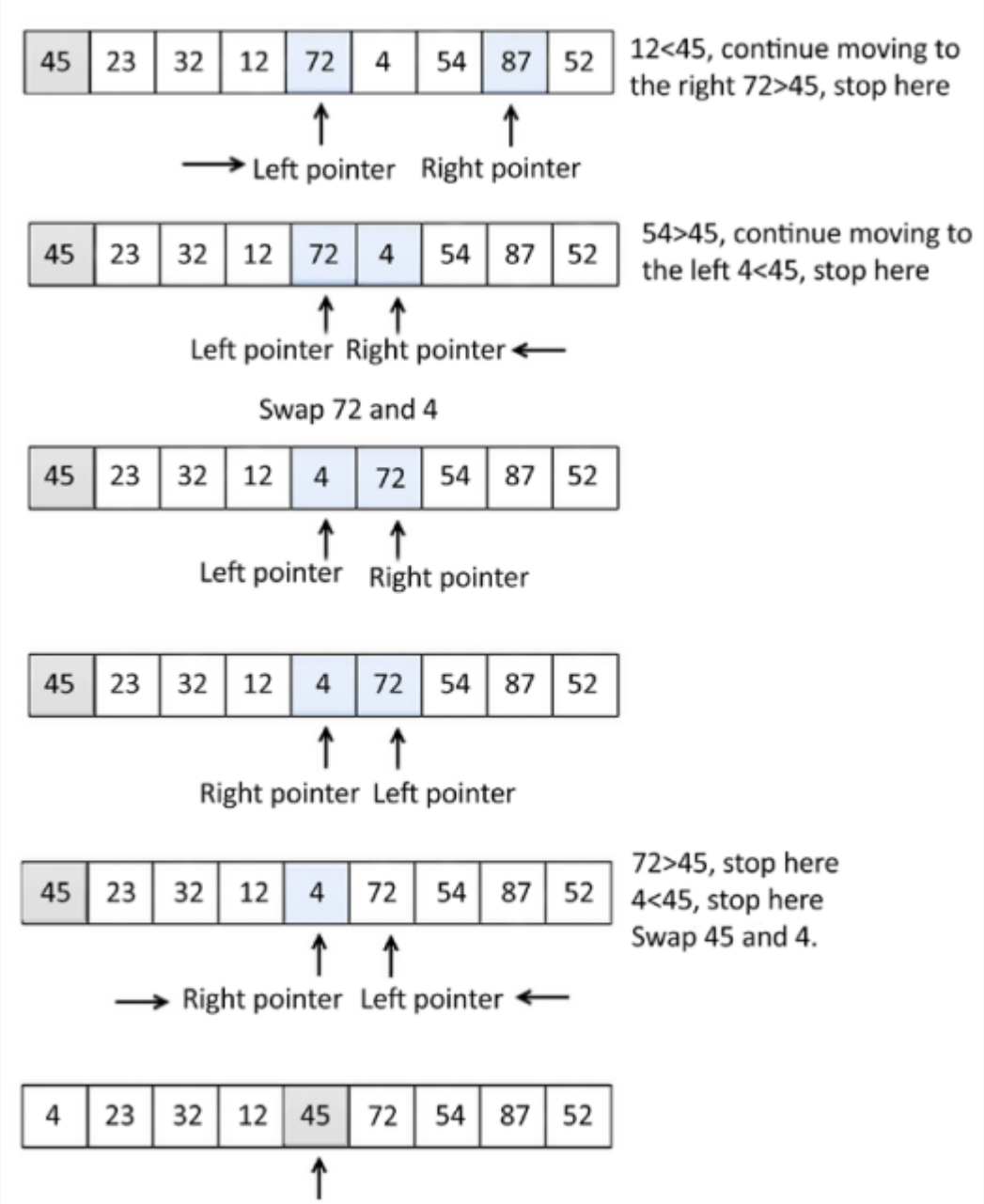

Figure 11.18: An example of the quicksort algorithm (continued)

Pode-se observar que, após a primeira iteração do algoritmo quicksort, o valor do pivô 45 é colocado em sua posição correta na lista. Agora temos duas sub-listas:

1. A sublista à esquerda do valor do pivô, 45, tem valores menores que 45.
2. Outra sublista à direita do valor do pivô contém valores maiores que 45. Aplicaremos o algoritmo quicksort recursivamente nessas duas sub-listas e o

repetiremos até que toda a lista esteja ordenada, como mostrado na Figura 11.19:

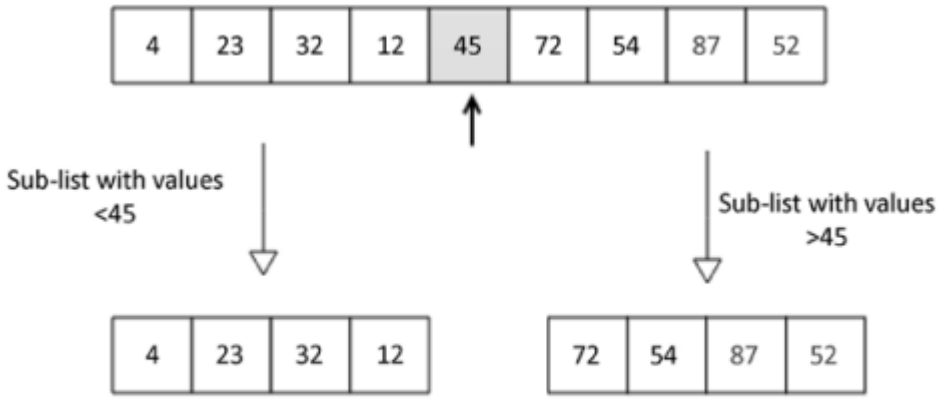

Figure 11.19: After the first iteration of the quicksort algorithm on an example list of elements

Nós vamos dar uma olhada na implementação do algoritmo quicksort na próxima seção.

Implementation of quicksort:

A tarefa principal do algoritmo quicksort é primeiro colocar o elemento pivô na sua posição correta para que possamos dividir a lista desordenada em duas sub-listas (sub-listas esquerda e direita); esse processo é chamado de etapa de particionamento. A etapa de particionamento é muito importante para entender a implementação do algoritmo quicksort, então primeiro entenderemos a implementação da etapa de particionamento com um exemplo. Nesse exemplo, dada uma lista de elementos, todos os elementos serão organizados de tal maneira que os elementos menores que o elemento pivô estarão do lado esquerdo dele e os elementos maiores que o pivô estarão do lado direito do elemento pivô.

Vamos olhar um exemplo para entender a implementação. Considere a seguinte lista de inteiros. [43, 3, 20, 89, 4, 77]. Vamos particionar essa lista usando a função de particionamento:

[43, 3, 20, 89, 4, 77]

Considere o código da função de particionamento abaixo; discutiremos cada linha detalhadamente:

```
def partition(unsorted_array, first_index, last_index):
    pivot = unsorted_array[first_index]
    pivot_index = first_index
    index_of_last_element = last_index
    less_than_pivot_index = index_of_last_element
    greater_than_pivot_index = first_index + 1
    while True:
        while unsorted_array[greater_than_pivot_index] < pivot and greater_than_pivot_index < last_index:
            greater_than_pivot_index += 1
        while unsorted_array[less_than_pivot_index] > pivot and less_than_pivot_index >= first_index:
            less_than_pivot_index -= 1
        if greater_than_pivot_index < less_than_pivot_index:
            temp = unsorted_array[greater_than_pivot_index]
            unsorted_array[greater_than_pivot_index] = unsorted_array[less_than_pivot_index]
            unsorted_array[less_than_pivot_index] = temp
        else:
            break
    unsorted_array[pivot_index] = unsorted_array[less_than_pivot_index]
    unsorted_array[less_than_pivot_index] = pivot
    return less_than_pivot_index
```

A função de particionamento recebe, como seus parâmetros, os índices do primeiro e último elementos da matriz que precisamos particionar.

O valor do pivô é armazenado na variável pivot, enquanto seu índice é armazenado em pivot_index.

Não estamos usando unsorted_array[0], porque quando o parâmetro de array não ordenado é chamado com um segmento de uma matriz, o índice 0 não necessariamente aponta para o primeiro elemento nessa matriz.

O índice do elemento ao lado do pivô, isto é, o ponteiro esquerdo, primeiro_index + 1, marca a posição onde começamos a procurar um elemento na matriz. Essa matriz é maior que o pivô, como sugere greater_than_pivot_index = primeiro_index + 1. O ponteiro direito less_than_pivot_index aponta para a posição do último elemento na lista less_than_pivot_index = index_of_last_element, onde começamos a procurar o elemento que é menor que o pivô.

Além disso, no início da execução do loop while principal, a matriz parece como mostrado na Figura 11.20:

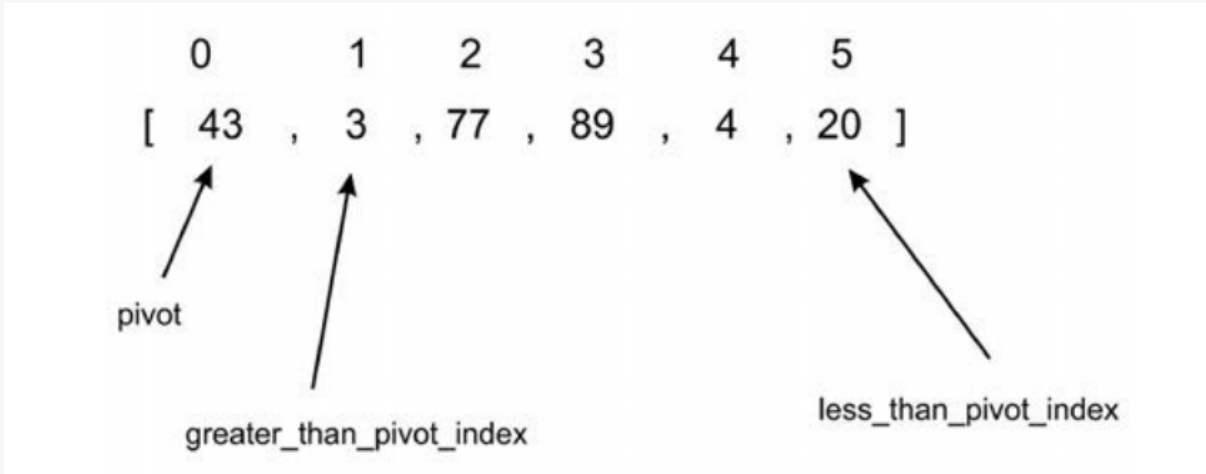

Figure 11.20: Illustration 1 of an example array for the quicksort algorithm

O primeiro loop interno while move um índice para a direita até chegar ao índice 2 porque o valor nesse índice é maior que 43. Neste ponto, o primeiro loop while quebra e não continua. Em cada teste da condição do primeiro loop while, greater_than_pivot_index += 1 é avaliado somente se a condição de teste do loop while avaliar para Verdadeiro. Isso faz com que a busca por um elemento maior que o pivô progrida para o próximo elemento à direita.

O segundo loop interno while move um índice de cada vez para a esquerda, até chegar ao índice 5, cujo valor, 20, é menor que 43, como mostrado na Figura 11.21:

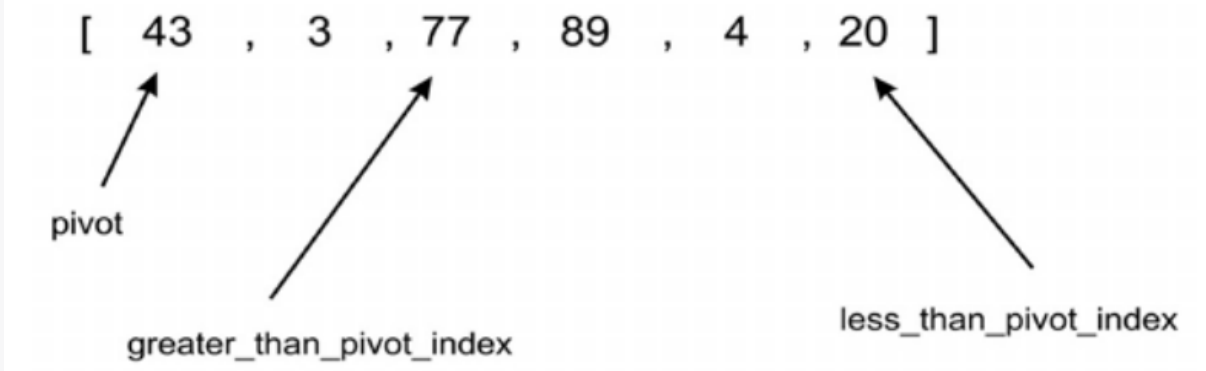

Figure 11.21 Illustration 2 of example array for quicksort algorithm

Em seguida, neste ponto, nenhum dos loops while internos pode ser executado mais adiante, e o próximo trecho de código é mostrado abaixo:

```
if greater_than_pivot_index < less_than_pivot_index:
    temp = unsorted_array[greater_than_pivot_index]
    unsorted_array[greater_than_pivot_index] = unsorted_array
    [less_than_pivot_index]
    unsorted_array[less_than_pivot_index] = temp
else:
    break
```

Aqui, uma vez que greater_than_pivot_index < less_than_pivot_index, o corpo da instrução if troca o elemento nesses índices. A condição else interrompe o loop infinito sempre que greater_than_pivot_index se torna maior que less_than_pivot_index. Nessa condição, significa que greater_than_pivot_index e less_than_pivot_index se cruzaram.
O array agora parece como mostrado na Figura 11.22:

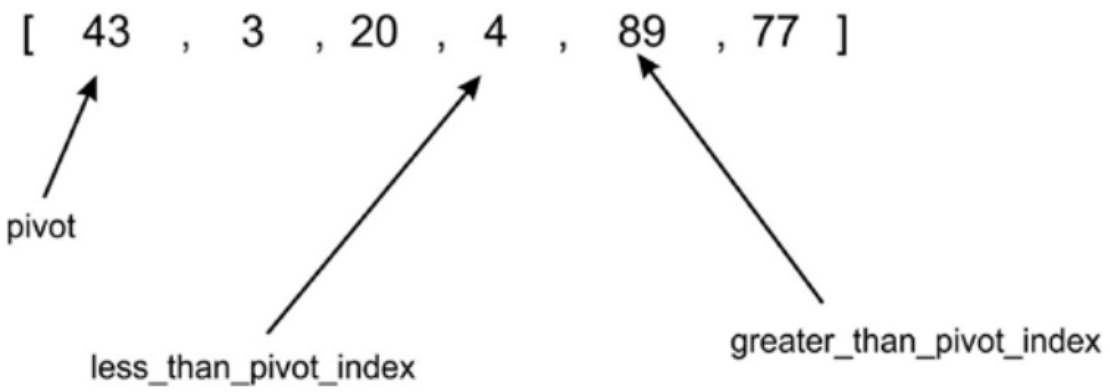

Figure 11.22: Illustration 3 of an example array for the quicksort algorithm

A instrução break é executada quando less_than_pivot_index é igual a 3 e greater_than_pivot_index é igual a 4. Assim que saímos do loop while, trocamos o elemento em unsorted_array[less_than_pivot_index] com o de less_than_pivot_index, que é retornado como o índice do pivô:

```
unsorted_array[pivot_index]                                    =
unsorted_array[less_than_pivot_index]
unsorted_array[less_than_pivot_index] = pivot
return less_than_pivot_index
```

A Figura 11.23 mostra como o código intercala 4 com 43 como a última etapa no processo de particionamento:

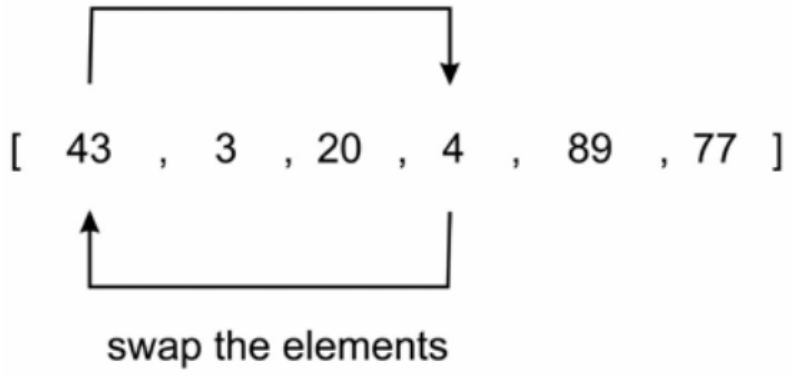

Figure 11.23: Illustration 4 of an example array for the quicksort algorithm

Para recapitular, na primeira vez que a função quick_sort foi chamada, ela foi particionada no elemento no índice 0. Após o retorno da função de particionamento, obtemos o array na ordem de [4, 3, 20, 43, 89, 77]. Como você pode ver, todos os elementos à direita do elemento 43 são maiores que 43, enquanto os à esquerda são menores. Assim, o particionamento está completo. Usando o ponto de divisão 43 com o índice 3, vamos classificar recursivamente os dois subarrays, [4, 30, 20] e [89, 77], usando o mesmo processo que acabamos de passar.

O corpo da função principal quick_sort é o seguinte:

```
def quick_sort(unsorted_array, first, last):
    if last - first <= 0:
        return
    else:
        partition_point = partition(unsorted_array, first, last)
        quick_sort(unsorted_array, first, partition_point-1)
        quick_sort(unsorted_array, partition_point+1, last)
```

A função quick_sort é bastante simples; inicialmente, o método de partição é chamado, que retorna o ponto de partição. Este ponto de partição está no array unsorted_array onde todos os elementos à esquerda são menores que o valor do pivô e todos os elementos à direita são maiores. Imprimimos o estado de unsorted_array imediatamente após o progresso da partição para ver o status do array depois de cada chamada. Após a primeira partição, a primeira submatriz [4, 3,

20] será feita; a partição dessa submatriz vai parar quando greater_than_pivot_index estiver no índice 2 e less_than_pivot_index estiver no índice.

1. Nesse ponto, os dois marcadores são ditos ter cruzado. Como greater_than_pivot_index é maior que less_than_pivot_index, a execução adicional do loop while cessará. O pivô 4 será trocado por 3, enquanto o índice 1 é retornado como o ponto de partição. Podemos usar o trecho de código abaixo para criar uma lista de elementos e usar o algoritmo quicksort para ordená-la:

```
my_array = [43, 3, 77, 89, 4, 20]
print(my_array)
quick_sort(my_array, 0, 5)
print(my_array)
```

A saída do código acima é a seguinte:

```
[43, 3, 77, 89, 4, 20]
[3, 4, 20, 43, 77, 89]
```

No algoritmo quicksort, o algoritmo de partição leva O(n) tempo. Como o algoritmo quicksort segue o paradigma dividir e conquistar, ele leva O(logn) tempo; portanto, a complexidade de tempo médio geral do quicksort é O(n) * O(logn) = O(nlogn). O algoritmo quicksort tem uma complexidade de tempo de execução pior caso de O(n^2). O pior caso de complexidade para o algoritmo quicksort ocorreria quando seleciona o pior ponto de pivô todas as vezes, e uma das partições sempre tem um único elemento. Por exemplo, se a lista já estiver classificada, a complexidade de pior caso ocorreria se a partição escolhesse o menor elemento como ponto de pivô. Quando a complexidade de pior caso ocorre, o algoritmo quicksort pode ser melhorado usando o quicksort aleatório. O algoritmo quicksort é eficiente quando a lista de elementos dada é muito longa; ele funciona melhor em comparação com os outros algoritmos mencionados para classificação em tais situações.