Ordenação: pág 345 até 356.

Ordenação:

Ordenar significa reorganizar dados de tal forma que eles estejam em ordem crescente ou decrescente. A ordenação é um dos algoritmos mais importantes da ciência da computação e é amplamente utilizada em algoritmos relacionados a bancos de dados. Para várias aplicações, se os dados estiverem ordenados, eles podem ser recuperados de forma eficiente, por exemplo, se for uma coleção de nomes, números de telefone ou itens em uma lista simples de tarefas. Neste capítulo, estudaremos algumas das técnicas de ordenação mais importantes e populares, incluindo as seguintes:

• Bubble sort

• Insertion sort

• Selection sort

• Quicksort

• Timsort

Algoritmos de ordenação:

Ordenação significa arranjar todos os itens em uma lista em ordem crescente ou decrescente. Podemos comparar diferentes algoritmos de ordenação pela quantidade de tempo e espaço de memória necessários para usá-los.
O tempo necessário por um algoritmo muda dependendo do tamanho da entrada. Além disso, alguns algoritmos são relativamente fáceis de

implementar, mas podem ter um mau desempenho em relação à complexidade de tempo e espaço, enquanto outros algoritmos são um pouco mais complexos de implementar, mas podem ter um bom desempenho ao ordenar listas mais longas de dados. Um dos algoritmos de ordenação, o merge sort, já discutimos no Capítulo 3, Técnicas e Estratégias de Design de Algoritmos. Discutiremos vários outros algoritmos de ordenação um por um em detalhes, juntamente com seus detalhes de implementação, começando pelo algoritmo de ordenação bubble sort.

Algoritmos de bubble sort:

A ideia por trás do algoritmo de ordenação bubble sort é muito simples. Dada uma lista desordenada, comparamos elementos adjacentes na lista e, após cada comparação, colocamos eles na ordem certa de acordo com seus valores. Então, trocamos os itens adjacentes se eles não estiverem na ordem correta. Esse processo é repetido n-1 vezes para uma lista de n itens.

Em cada iteração, o maior elemento da lista é movido para o final da lista. Após a segunda iteração, o segundo maior elemento será colocado na penúltima posição da lista. O mesmo processo é repetido até que a lista esteja ordenada.

Vamos pegar uma lista com apenas dois elementos, {5, 2}, para entender o conceito de bubble sort, como mostrado na Figura 11.1:

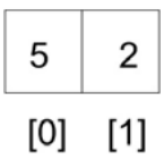

Figure 11.1: Example of bubble sort

Para ordenar esta lista de dois elementos, primeiro, comparamos 5 e 2; como 5 é maior que 2, significa que eles não estão na ordem correta, então trocamos esses valores para colocá-los na ordem correta. Para trocar esses dois números, primeiro, movemos o elemento armazenado no índice 0 em uma variável temporária (passo 1 da Figura 11.2), então o elemento armazenado no índice 1 é copiado para o índice 0 (passo 2 da Figura 11.2) e, finalmente, o primeiro elemento armazenado na variável temporária é armazenado de volta no índice 1 (passo 3 da Figura 11.2). Então, primeiro, o elemento 5 é copiado para uma variável temporária, temp. Em seguida, o elemento 2 é movido para o índice 0. Finalmente, 5 é movido de temp para o índice 1. A lista agora conterá os elementos [2, 5]:

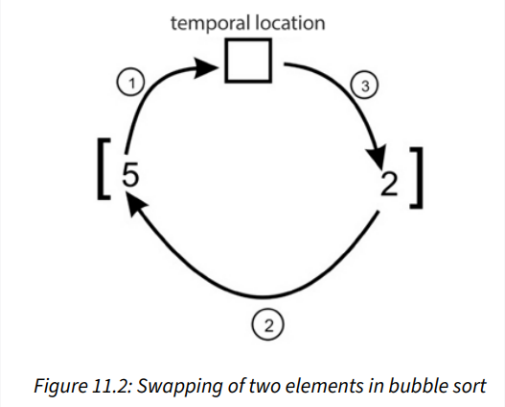

Figure 11.2: Swapping of two elements in bubble sort

O código a seguir irá trocar os elementos unordered_list[0] com unordered_list[1] se eles não estiverem na ordem correta:

```
unordered_list = [5, 2]
temp = unordered_list[0]
unordered_list[0] = unordered_list[1]
unordered_list[1] = temp
print(unordered_list)
```

SAÍDA:

```
[2, 5]
```

Agora que conseguimos trocar um array de dois elementos, deve ser simples usar a mesma ideia para ordenar uma lista inteira usando o algoritmo de ordenação bubble sort.

Vamos considerar outro exemplo para entender o funcionamento do algoritmo de bubble sort e ordenar uma lista desordenada de seis elementos, como {45, 23, 87, 12, 32, 4}. Na primeira iteração, começamos comparando os dois primeiros elementos, 45 e 23, e os trocamos, já que 45 deve ser colocado após 23. Então, comparamos os próximos valores adjacentes, 45 e 87, para ver se estão na ordem correta. Como 87 é um valor maior do que 45, não precisamos trocá-los. Trocamos dois elementos se eles não estiverem na ordem correta.

Podemos ver, na Figura 11.3, que após a primeira iteração do bubble sort, o maior elemento, 87, é colocado na última posição da lista:

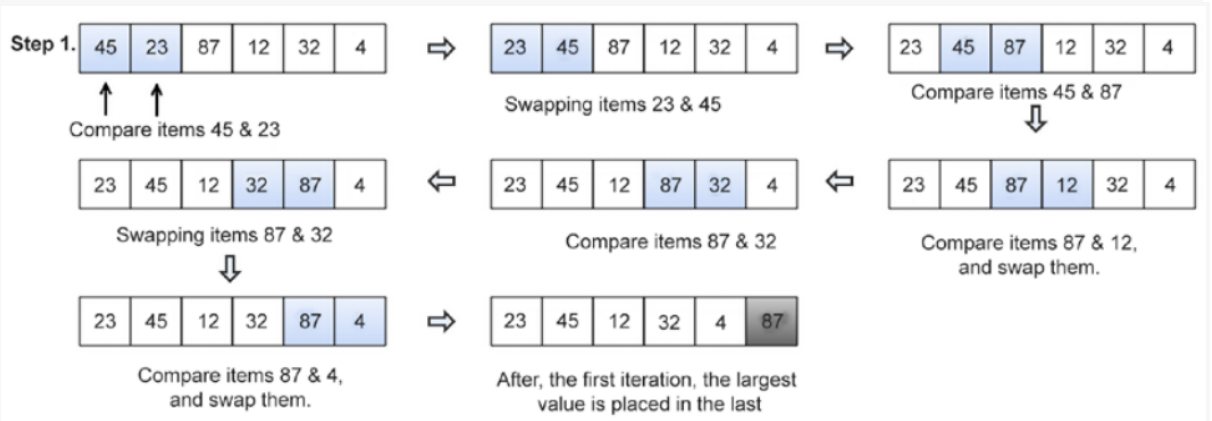

Figure 11.3: Steps of the first iteration to sort an example array using bubble sort

Depois da primeira iteração, só precisamos arranjar os (n-1) elementos restantes; repetimos o mesmo processo comparando os elementos adjacentes para os cinco elementos restantes. Após a segunda iteração, o segundo maior elemento, 45, é colocado na penúltima posição da lista, como mostrado na Figura 11.4:

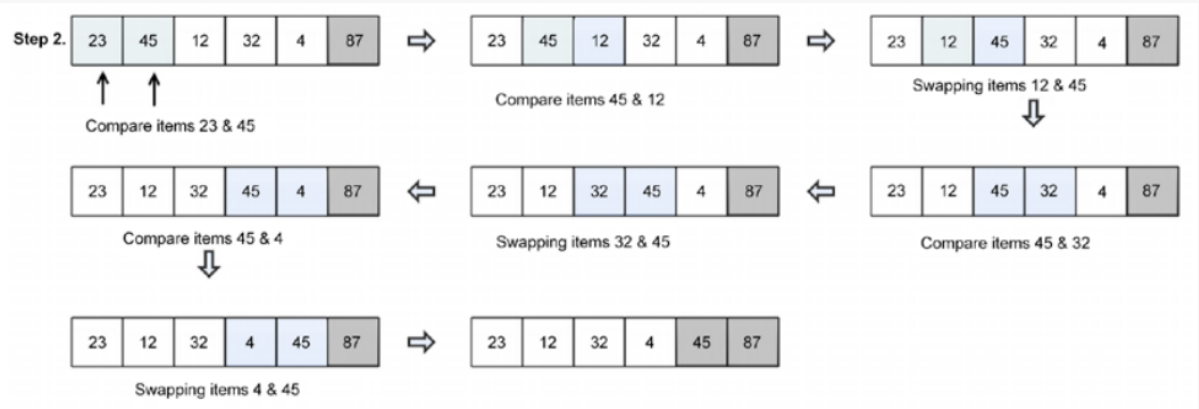

Figure 11.4: Steps of the second iteration to sort an example array using bubble sort

Em seguida, temos que comparar os (n-2) elementos restantes para arranjá-los, como mostrado na Figura 11.5:

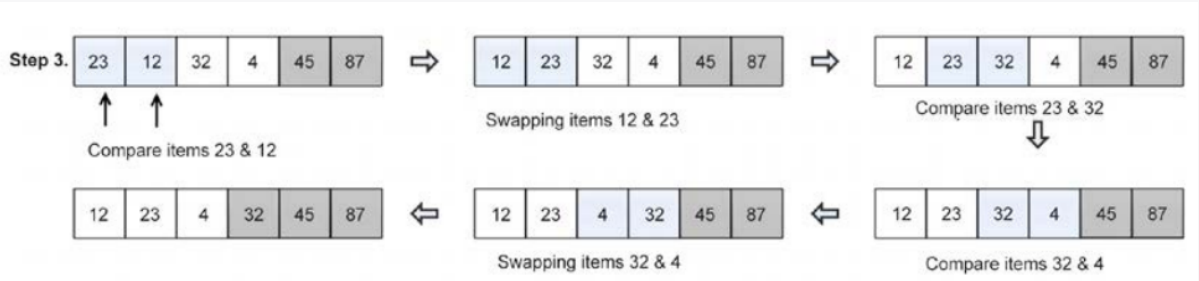

Figure 11.5: Steps of the third iteration to sort an example array using bubble sort

Da mesma forma, comparamos os elementos restantes para ordená-los, como mostrado na Figura 11.6:

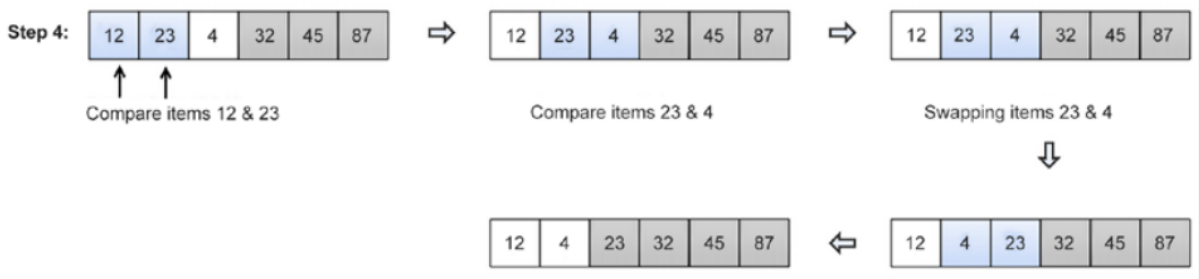

Figure 11.6: Steps of the fourth iteration to sort an example array using bubble sort

Por fim, para os dois elementos restantes, colocamos-os na ordem correta para obter a lista final ordenada, como mostrado na Figura 11.7:

Figure 11.7: Steps of the fifth iteration to sort an example array using bubble sort

O código completo em Python do algoritmo bubble sort é mostrado abaixo e, em seguida, cada etapa é explicada em detalhes:

```
def bubble_sort(unordered_list):
    iteration_number = len(unordered_list)-1
    for i in range(iteration_number,0,-1):
        for j in range(i):
            if unordered_list[j] > unordered_list[j+1]:
                temp = unordered_list[j]
                unordered_list[j] = unordered_list[j+1]
                unordered_list[j+1] = temp
```

O bubble sort é implementado usando um loop duplo aninhado, onde um loop está dentro de outro loop. No bubble sort, o loop interno compara e troca repetidamente os elementos adjacentes em cada iteração para uma lista dada, e o loop externo acompanha quantas vezes o loop interno deve ser repetido.

Primeiramente, no código acima, calculamos quantas vezes o loop deve ser executado para concluir todas as trocas; isso é igual ao comprimento da lista menos 1 e pode ser escrito como número_de_iterações = len(lista_desordenada) - 1. Aqui, a função len fornecerá o comprimento da lista. Subtraímos 1 porque isso nos dá exatamente o número máximo de iterações para executar. O loop externo garante isso e é executado para um número menor que o tamanho da lista.

Além disso, no código acima, para cada iteração, no loop interno, comparamos os elementos adjacentes usando a declaração if e verificamos se os elementos adjacentes estão na ordem correta ou não. Para a primeira iteração, o loop interno deve ser executado n vezes, para a segunda iteração, o loop interno deve ser executado n-1 vezes e assim por diante. Por exemplo, para ordenar uma lista de três números, digamos [3, 2, 1], o loop interno é executado duas vezes e precisamos trocar os elementos um máximo de duas vezes, conforme mostrado na Figura 11.8.

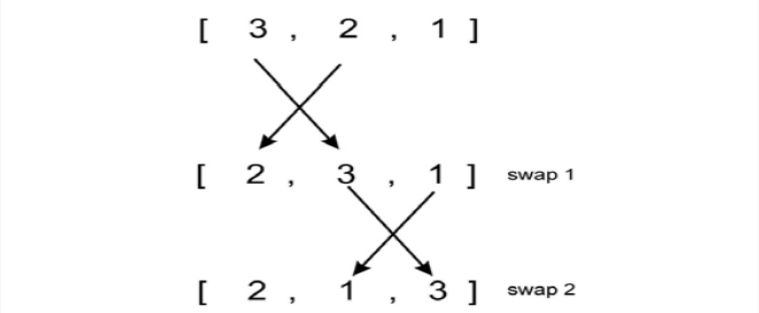

Figure 11.8: Number of swaps in iteration 1 for an example list [3, 2, 1]

Além disso, após a primeira iteração, na segunda iteração, executamos o loop interno uma vez, como mostrado na Figura 11.9.

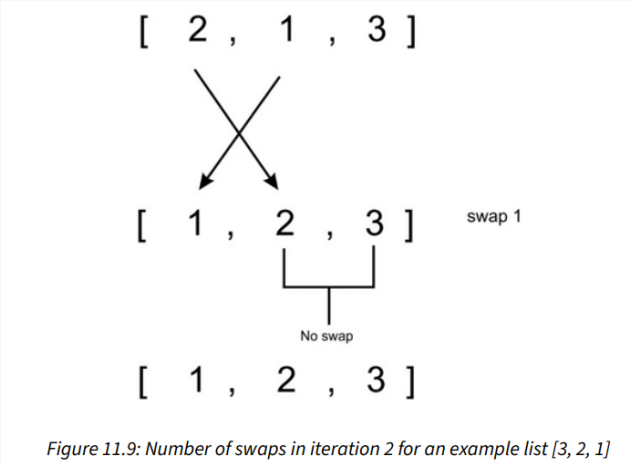

Figure 11.9: Number of swaps in iteration 2 for an example list [3, 2, 1]

O seguinte trecho de código pode ser usado para implementar o algoritmo bubble sort:

```
my_list = [4,3,2,1]
bubble_sort(my_list)
print(my_list)

my_list = [1,12,3,4]
bubble_sort(my_list)
print(my_list)
```

A saída é a seguinte:

```
[1, 2, 3, 4]
[1, 3, 4, 12]
```

No pior caso, o número de comparações necessárias na primeira iteração será (n-1), na segunda, o número de comparações será (n-2), e na terceira iteração será (n-3), e assim por diante. Portanto, o número total de comparações necessárias no bubble sort será o seguinte:

```
(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2
                    n(n+1)/2
                      O(n2)
```

O algoritmo bubble sort não é um algoritmo de ordenação eficiente, pois fornece uma complexidade de tempo de execução pior caso de O(n2) e uma complexidade de melhor caso de O(n). A situação de pior caso ocorre quando queremos classificar a lista dada em ordem crescente e a lista dada está em ordem decrescente, e o melhor caso ocorre quando a lista dada já está ordenada; nesse caso, não haverá necessidade de troca.

Geralmente, o algoritmo bubble sort não deve ser usado para classificar grandes listas. O algoritmo bubble sort é adequado para aplicativos onde o desempenho não é importante ou o comprimento da lista dada é

curto e, além disso, um código curto e simples é preferido. O algoritmo bubble sort funciona bem em listas relativamente pequenas.

Agora, vamos dar uma olhada no algoritmo insertion sort.

Insertion sort algorithm:

A ideia do algoritmo de ordenação por inserção é manter duas sub-listas (uma sublista é uma parte da lista maior original), uma que está ordenada e outra que não está ordenada, na qual elementos são adicionados um por um da sublista não ordenada para a sublista ordenada. Então, pegamos elementos da sublista não ordenada e os inserimos na posição correta na sublista ordenada, de tal forma que essa sublista permaneça ordenada.

No algoritmo de ordenação por inserção, sempre começamos com um elemento, considerando-o como ordenado, e depois pegamos elementos um por um da sublista não ordenada e os colocamos nas posições corretas (em relação ao primeiro elemento) na sublista ordenada. Então, depois de pegar um elemento da sublista não ordenada e adicioná-lo à sublista ordenada, agora temos dois elementos na sublista ordenada. Em seguida, pegamos outro elemento da sublista não ordenada e o colocamos na posição correta (em relação aos dois elementos já ordenados) na sublista ordenada. Seguimos repetindo esse processo para inserir todos os elementos um por um da sublista não ordenada na sublista ordenada. Os elementos sombreados denotam as sub-listas ordenadas na Figura 11.10, e em cada iteração, um elemento da sublista não ordenada é inserido na posição correta na sublista ordenada.

Vamos considerar um exemplo para entender o funcionamento do algoritmo de ordenação por inserção. Digamos que temos que ordenar uma lista de seis elementos: {45, 23, 87, 12, 32, 4}. Primeiramente,

começamos com um elemento, considerando-o como ordenado, e então pegamos o próximo elemento, 23, da sublista não ordenada e o inserimos na posição correta na sublista ordenada. Na próxima iteração, pegamos o terceiro elemento, 87, da sublista não ordenada e o inserimos novamente na sublista ordenada na posição correta. Seguimos o mesmo processo até que todos os elementos estejam na sublista ordenada. Todo o processo é mostrado na Figura 11.10.

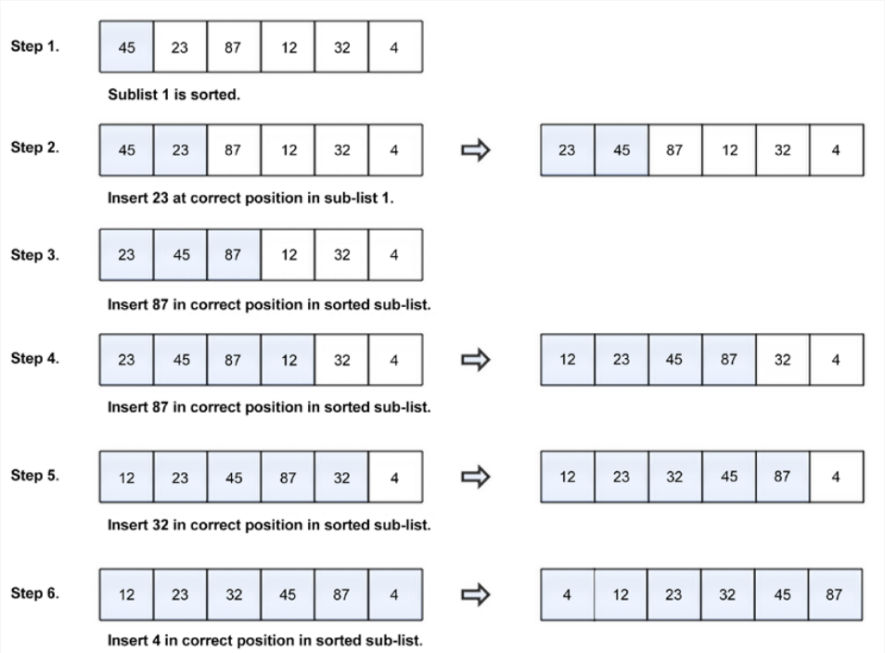

Figure 11.10: Steps to sort example array elements using the insertion sort algorithm

O código completo em Python para o algoritmo de ordenação por inserção é dado abaixo; cada declaração do algoritmo é explicada em detalhes com um exemplo.

```
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
```

Para entender a implementação do algoritmo de ordenação por inserção, vamos considerar outro exemplo de cinco elementos, {5, 1, 100, 2, 10}, e examinar o processo com uma explicação detalhada. Vamos considerar a seguinte matriz, como mostrado na Figura 11.11.

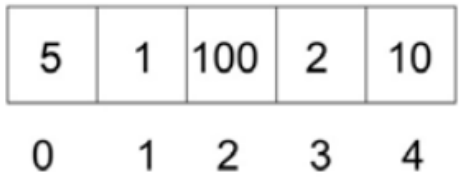

Figure 11.11: An example array with index positions

O algoritmo começa usando um loop for para percorrer os índices 1 e 4. Começamos a partir do índice 1 porque consideramos o elemento armazenado no índice 0 como estando na submatriz ordenada e os elementos entre o índice 1 e 4 pertencem à sublista não ordenada, como mostrado na Figura 11.12.

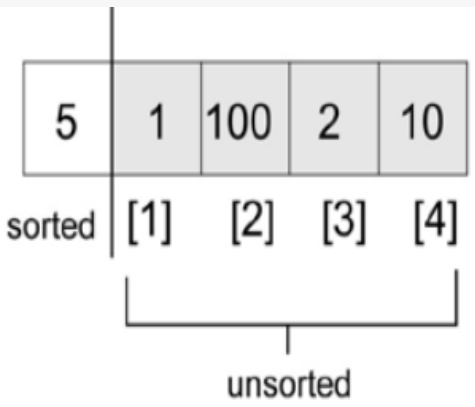

Figure 11.12: Demonstration of sorted and unsorted sublists in insertion sorting

No início da execução do loop, temos o seguinte trecho de código:

```
for index in range(1, len(unsorted_list)):
    search_index = index
    insert_value = unsorted_list[index]
```

No início da execução de cada execução do laço for, o elemento em unsorted_list [index] é armazenado na variável insert_value. Mais tarde, quando encontrarmos a posição apropriada na porção ordenada da sublista, insert_value será armazenado nesse índice na sublista ordenada. O próximo trecho de código é mostrado abaixo:

```
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
```

search_index é usado para fornecer informações ao laço while, ou seja, exatamente onde encontrar o próximo elemento que precisa ser inserido na sublista ordenada.

O laço while percorre a lista de trás para frente, guiado por duas condições. Primeiro, se search_index> 0, então significa que existem mais elementos na parte ordenada da lista; segundo, para que o laço while seja executado, unsorted_list [search_index-1] deve ser maior que a variável insert_value.

O array unsorted_list [search_index-1] fará uma das seguintes coisas:

• Apontar para o elemento imediatamente antes de unsorted_list [search_index], antes que o laço while seja executado pela primeira vez

• Apontar para um elemento antes de unsorted_list [search_index-1], depois que o laço while foi executado pela primeira vez

Na lista de exemplo, o laço while será executado porque 5> 1. No corpo do laço while, o elemento em unsorted_list [search_index-1] é armazenado em unsorted_list [search_index].

E, search_index - = 1 move a travessia da lista para trás até que ela mantenha um valor de 0.

Depois que o laço while sai, a última posição conhecida de search_index (que, neste caso, é 0) agora nos ajuda a saber onde inserir insert_value. A Figura 11.13 mostra a posição dos elementos após a primeira iteração:

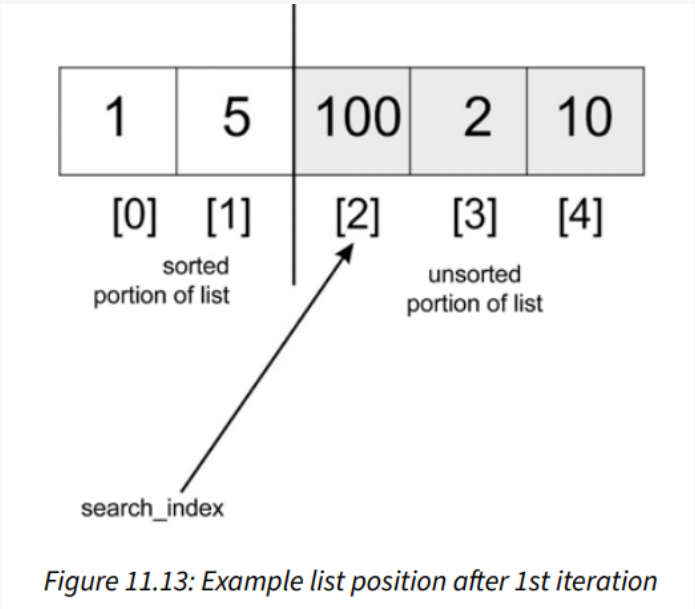

Figure 11.13: Example list position after 1st iteration

Na segunda iteração do loop for, search_index terá um valor de 2, que é o índice do terceiro elemento no array. Neste ponto, iniciamos nossa comparação na direção esquerda (em direção ao índice 0). 100 será comparado com 5, mas como 100 é maior que 5, o loop while não será executado. 100 será substituído por si mesmo, porque a variável

search_index nunca foi decrementada. Como tal, unsorted_list [search_index] = insert_value não terá efeito.

Quando search_index está apontando para o índice 3, comparamos 2 com 100 e movemos 100 para onde 2 está armazenado. Em seguida, comparamos 2 com 5 e movemos 5 para onde 100 foi armazenado inicialmente. Neste ponto, o loop while será interrompido e 2 será armazenado no índice 1. O array será parcialmente classificado com os valores [1, 2, 5, 100, 10]. O passo anterior ocorrerá mais uma vez para que a lista seja classificada. O código a seguir pode ser usado para criar uma lista de elementos, que podemos classificar usando o método insertion_sort () definido:

```
my_list = [5, 1, 100, 2, 10]
print("Original list", my_list)
insertion_sort(my_list)
print("Sorted list", my_list)
```

A saída do código acima é a seguinte:

```
Original list [5, 1, 100, 2, 10]
Sorted list [1, 2, 5, 10, 100]
```

A complexidade de tempo no pior caso do insertion sort é quando a lista dada de elementos está classificada em ordem inversa. Nesse caso, cada elemento terá que ser comparado com cada um dos outros elementos. Então, precisaremos de uma comparação na primeira iteração, duas comparações na segunda iteração e três comparações na terceira iteração e (n-1) comparações na (n-1) iteração. Assim, o número total de comparações é:

1 + 2 + 3 .. (n-1)

n(n-1)/2

Assim, o algoritmo de ordenação por inserção tem uma complexidade de tempo no pior caso de $O(n^2)$. Além disso, a complexidade de tempo no melhor caso do algoritmo de ordenação por inserção é $O(n)$, na situação em que a lista de entrada dada já está ordenada, em que cada elemento da sublista não ordenada é comparado apenas com o elemento mais à direita da sublista ordenada em cada iteração. O algoritmo de ordenação por inserção é bom para ser usado quando a lista dada tem um pequeno número de elementos e é mais adequado quando os dados de entrada chegam um por um e precisamos manter a lista ordenada. Agora vamos dar uma olhada no algoritmo de ordenação por seleção. ***(Selection sort).***