Sorting Algorithm

The best way to practice your programming skills is by writing actual code. One of the common computation is to sort some items in some way. For example, sorting a number from smallest to biggest or names from A to Z. In this notebook, we will describe some sorting algorithms which you can implement in Python.

Bubble Sort

Bubble sort is one of the simplest sorting algorithms. We will be following the PCDIT framework (**P**roblem statement, Test **C**ases, **D**esign of Algorithm, **I**mplementation, and **T**esting) in describing the steps of these algorithms.

Problem Statement

The problem is specified as follows. Given a sequence of numbers, write some steps to sort the sequence in some order. Usually, we will sort the sequence from the smallest to the largest.

Test Case

For example, given the following input:

```
# Python Code
numbers = [16, 14, 10, 8, 7, 8, 3, 2, 4, 1]
```

We want to write some steps that sort the numbers, such that the output will be:

```
[1, 2, 3, 4, 7, 8, 8, 10, 14, 16]
```

We can intuitively try to sort the numbers by comparing two numbers (a pair) at a time. If the order is incorrect, we will swap the two numbers. Let's illustrate the steps!

- We start from the input:

  [**16, 14**, 10, 8, 7, 8, 3, 2, 4, 1]
- We compare the first two numbers (16, 14), i.e. **positions 0 and 1** in the list. Since 16 is bigger than 14, we will swap them.

  [**14, 16**, 10, 8, 7, 8, 3, 2, 4, 1]
- Now we move to the next pair (16, 10), i.e. **positions 1 and 2** in the list. Since 16 is bigger than 10, we will swap them again.

  [14, **16, 10**, 8, 7, 8, 3, 2, 4, 1]

  [14, **10, 16**, 8, 7, 8, 3, 2, 4, 1]
- And next pair (16, 8), i.e. **positions 2 and 3**. Since 16 is bigger than 8, we will swap.

  [14, 10, **16, 8**, 7, 8, 3, 2, 4, 1]

  [14, 10, **8, 16**, 7, 8, 3, 2, 4, 1]
- We continue until 16 reaches the last position.

  [14, 10, 8, **16, 7**, 8, 3, 2, 4, 1]

  [14, 10, 8, **7, 16**, 8, 3, 2, 4, 1]

  --

  [14, 10, 8, 7, **16, 8**, 3, 2, 4, 1]

  [14, 10, 8, 7, **8, 16**, 3, 2, 4, 1]

  --

  [14, 10, 8, 7, 8, **16, 3**, 2, 4, 1]

  [14, 10, 8, 7, 8, **3, 16**, 2, 4, 1]

  --

  [14, 10, 8, 7, 8, 3, **16, 2**, 4, 1]

  [14, 10, 8, 7, 8, 3, **2, 16**, 4, 1]

--

[14, 10, 8, 7, 8, 3, 2, **16, 4**, 1]

[14, 10, 8, 7, 8, 3, 2, **4, 16**, 1]

--

[14, 10, 8, 7, 8, 3, 2, 4, **16, 1**]

[14, 10, 8, 7, 8, 3, 2, 4, **1, 16**]

So now the largest number has occupied its place in the last position. But the other numbers are still not in the right order. Therefore, we have to repeat the steps starting from the beginning again. We start from the pair (14, 10). This will repeat the **pair-wise** comparison and move 14 to the second last position. Similarly, we can see how 10 and 8 will reach their final position. Here, we no longer show the pair-wise comparison and swapping for brevity.

- [10, 8, 7, 8, 3, 2, 4, 1, **14, 16**]
- [8, 7, 8, 3, 2, 4, 1, **10, 14, 16**]
- [7, 8, 3, 2, 4, 1, **8, 10, 14, 16**]

At this point, we will start comparing the pair (7, 8). **But since 7 is not greater than 8, there is no swap**.

- [**7, 8**, 3, 2, 4, 1, **8, 10, 14, 16**]

Afterward, we will compare the pair (8, 3).

- [7, **8, 3**, 2, 4, 1, **8, 10, 14, 16**]

Since 8 is greater than 3, **there will be a swap**, and these **pair-wise swaps** will continue to push 8 to its final position.

- [7, 3, 2, 4, 1, **8, 8, 10, 14, 16**]

Now we begin again with the pair (7, 3) and push 7 to its final position. The detailed pair-wise swapping is not shown below, but the final arrangement at the end of this iteration will be as the one below.

- [3, 2, 4, 1, **7, 8, 8, 10, 14, 16**]

At this point, we will start a new iteration by comparing the pair (3, 2). Since 3 is greater than 2, there will be a swap. The next comparison will fall on the pair (3, 4), i.e. [2, **3, 4**, 1, 7, 8, 8, 10, 14, 16]. But since 3 is less than 4, there is no swap happing. And the rest of the comparison will push 4 to its final position.

- [2, 3, 1, **4, 7, 8, 8, 10, 14, 16**]

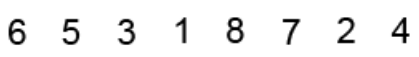
A similar situation occurs. There is no swap between (2, 3), but then it will swap (3, 1).

- [2, 1, **3, 4, 7, 8, 8, 10, 14, 16**]

In the last iteration, it wil swap (2, 1).

- [**1, 2, 3, 4, 7, 8, 8, 10, 14, 16**]

Here's an animated example from Wikipedia,sorting a different sequence of numbers.



Design of Algorithm

After we work on the test cases, we can now write down the steps in pseudocode. Several things to note from the above test cases:

- There are two kind of iterations:
  1. the *inner* iteration is when we compare the pairs (a, b) and do a swap if $a>b$,
  2. the *outer* iteration is when repeat the inner iteration pass starting from the first pair again.
- The number of *inner* iteration is the $n-1$, where n is the number of elements in the list. This is because the inner iteration

compares a pair. So if there is n elements, there will be $n-1$ pairs to compare.
- The number of *outer* iteration is also $n-1$. You can refer back to the case above that there were 9 *outer* iterations for the 10 elements.

Let's write down what we did in the above case. Note that in this algorithm we chose not to return the sorted list as a new object, but rather *sort the list in place*. This means that the input list is modified and the sorted list exists in the object pointed by the input list. The advantage of this is that the list need not be duplicated and the memory is saved as we deal only with one list.

```
Bubble Sort
Version: 1
Input: array
Output: None, sort in place
Steps:
1. n = length of array
2. For outer_index from 1 to n-1, do:
    2.1 For inner_index from 1 to n-1, do:
        2.1.1 first_number = array[inner_index - 1]
        2.1.2 second_number = array[inner_index]
        2.1.3 if first_number > second_number, do:
            2.1.3.1 swap(first_number, second_number)
```

Optimised Bubble Sort

We can optimized bubble sort algorithm by noting the following:

- If the sequence is already in order, we can reduce the next *outer* iteration. For example, if the input is

  [16, 1, 2, 3, 4, 5]

  The first iteration will push 16 to the last position.

  [1, 2, 3, 4, 5, **16**]

  In the second iteration, no swap is made since all the numbers are already in the correct order. However, with the above algorithm, the outer iteration will still repeat for $n-1$ times. We can conclude therefore that if no swap is made in one pass of *outer* iteration, the sequence is already in order, and we can stop the *outer* iteration. We can then modify the pseudocode as follows.

  ```
  Bubble Sort
  Version: 2
  Input: array
  Output: None, sort in place
  Steps:
  1. n = length of array
  2. swapped = True
  3. As long as swapped is True, do:
      3.1 swapped = False
      3.2 For inner_index from 1 to n-1, do:
          3.2.1 first_number = array[inner_index - 1]
          3.2.2 second_number = array[inner_index]
          3.2.3 if first_number > second_number, do:
              3.2.3.1 swap(first_number, second_number)
              3.2.3.2 swapped = True
  ```

Let's compare this with version 1.

- Notice that the number of *outer* iteration is much less in version two as compared to version one.

- The n-th pass of the *outer* iteration will place the n-th largest number to its final position. For example, in the given input below,

  [16, 14, 10, 8, 7, 8, 3, 2, 4, 1]

  In the 1-st *outer* pass, the 1-st largest number (i.e. 16) will be placed to its final position.

  [14, 10, 8, 7, 8, 3, 2, 4, 1, **16**]

  This means that we can reduce the number of *inner* iteration. Instead of comparing $n-1$ pairs for each *inner* iteration, we can reduce the number of *inner* iteration after each pass of *outer* iteration is done. For example, in the above example, we have 10 elements. In the first *outer* iteration, we have 9 comparisons in the *inner* iteration. In the next *outer* iteration, we can simply do 8

elements. In the first outer iteration, we have 9 comparisons in the inner iteration. In the next outer iteration, we can simply do 8 comparisons instead of 9. In the third *outer* iteration, we can do just 7 comparisons instead of 9, and so on.

We can re-write the pseudocode as follows.

```
Bubble Sort
Version: 3
Input: array
Output: None, sort in place
Steps:
1. n = length of array
2. swapped = True
3. As long as swapped is True, do:
    3.1 swapped = False
    3.2 For inner_index from 1 to n-1, do:
        3.2.1 first_number = array[inner_index - 1]
        3.2.2 second_number = array[inner_index]
        3.2.3 if first_number > second_number, do:
            3.2.3.1 swap(first_number, second_number)
            3.2.3.2 swapped = True
    3.3 n = n -1
```

The additional step is 3.3 which is to reduce the number of n. With this, Step 3.2 will decrease by one in the next *outer* iteration.

- It can happen that more than one elements are place in their final positions in one *outer* iteration pass. This means that we don't have to decrease the number of *inner* iteration by 1 on each step of *outer* iteration. We can record down, at which position was the last swap, and on the next *outer* iteration, we can do comparison up to that last position. To illustrate this, consider the following input.

  [1000, 1, 4, 7, 3, 10, 100]

  In the first *outer* pass, we push 1000 to its final position. This means we have $n-1$ comparisons and swaps.

  [1, 4, 7, 3, 10, 100, 1000]

  In the second *outer* pass, we first compare the pair (1, 4), but no swap happens. Similarly with (4, 7). Now, when comparing the pair (7, 3), we do a swap to result in.

  [1, 4, **3, 7**, 10, 100, 1000]

  When we have a swap, we will record down our position. In this case, it is at the position of the **4th element in the list, or index 3** (if our index starts from 0). Subsequently, we compare (7, 10), but no swap happens. Similarly with (10, 100). We stop at this point because the second iteration only compares up to the second last element in the above algorithm.

  Now, if we follow the previous algorithm, the next *outer* pass will compare up to the third last element, which is 10. However, we note that the last five elements are already ordered. We know this because in the last pass, the last swap happens at (7, 3) to (3, 7) pair. In this case, we just need to run our *inner* iteration up to this position, i.e. **4th position, or index 3**.

  We can modify our pseudocode as follows.

```
Bubble Sort
Version: 4
Input: array
Output: None, sort in place
Steps:
1. n = length of array
2. swapped = True
3. As long as swapped is True, do:
    3.1 swapped = False
    3.2 new_n = 0
    3.3 For inner_index from 1 to n-1, do:
        3.3.1 first_number = array[inner_index - 1]
        3.3.2 second_number = array[inner_index]
        3.3.3 if first_number > second_number, do:
            3.3.3.1 swap(first_number, second_number)
            3.3.3.2 swapped = True
            3.3.3.3 new_n = inner_index
    3.4 n = new_n
```

  In the above pseudocode, we set record down the position of the element on the last swap (step 3.3.3.3), and we assign this as the new ending position for the next *outer* pass (step 3.4).

Insertion Sort

Insertion sort is another algorithm that solves the same problem. Let's start by looking at the same test case.

Test Case

Given the following input:

```
# Python Code
numbers = [16, 14, 10, 8, 7, 8, 3, 2, 4, 1]
```

We want to write some steps that sort the numbers such that the output will be:

```
[1, 2, 3, 4, 7, 8, 8, 10, 14, 16]
```

- We start from the *second* element in the list, i.e. 14.

  [16, **14**, 10, 8, 7, 8, 3, 2, 4, 1]
- We then compare that number with the one on the left. If it is smaller, then we will swap. Since $14<16$, we do a swap.

  [**14, 16**, 10, 8, 7, 8, 3, 2, 4, 1]
- Since 14 has reached its place, we now move to the *third* element in the list, i.e. 10. Since $10 < 16$, we swap (16, 10).

  [14, **16, 10**, 8, 7, 8, 3, 2, 4, 1]

  [14, **10, 16**, 8, 7, 8, 3, 2, 4, 1]

  Now we continue comparing 10 with the one on its left, i.e. 14. Since $10<14$, we swap (14, 10).

  [**14, 10**, 16, 8, 7, 8, 3, 2, 4, 1]

  [**10, 14**, 16, 8, 7, 8, 3, 2, 4, 1]

  Now 10 has reached its position.
- We now move to the *fourth* element, i.e. 8, and compare it with the number on its left. Since $8 < 16$, we swap (16, 8). We then continue swapping until 8 reaches its position.

  [10, 14, **16, 8**, 7, 8, 3, 2, 4, 1]

  [10, 14, **8, 16**, 7, 8, 3, 2, 4, 1]

  [10, **14, 8**, 16, 7, 8, 3, 2, 4, 1]

  [10, **8, 14**, 16, 7, 8, 3, 2, 4, 1]

  [**10, 8**, 14, 16, 7, 8, 3, 2, 4, 1]

  [**8, 10**, 14, 16, 7, 8, 3, 2, 4, 1]
- We now move to the *fifth* element, i.e. 7. We then have the same swapping all the way until 7 reaches its place.

  [8, 10, 14, **16, 7**, 8, 3, 2, 4, 1]

  [8, 10, 14, **7, 16**, 8, 3, 2, 4, 1]

  [8, 10, **14, 7**, 16, 8, 3, 2, 4, 1]

  [8, 10, **7, 14**, 16, 8, 3, 2, 4, 1]

  [8, **10, 7**, 14, 16, 8, 3, 2, 4, 1]

  [8, **7, 10**, 14, 16, 8, 3, 2, 4, 1]

  [**8, 7**, 10, 14, 16, 8, 3, 2, 4, 1]

  [**7, 8**, 10, 14, 16, 8, 3, 2, 4, 1]
- We now move to the *sixth* element, i.e. 8. We will continue swapping until 8 encounters another 8 in the 2nd element. At this point, the swapping will stop.

[7, 8, 10, 14, **16, 8**, 3, 2, 4, 1]

[7, 8, 10, 14, **8, 16**, 3, 2, 4, 1]

[7, 8, 10, **14, 8**, 16, 3, 2, 4, 1]

[7, 8, 10, **8, 14**, 16, 3, 2, 4, 1]

[7, 8, **10, 8**, 14, 16, 3, 2, 4, 1]

[7, 8, **8, 10**, 14, 16, 3, 2, 4, 1]

[7, **8, 8**, 10, 14, 16, 3, 2, 4, 1]

no swapping occurs

- We can now move to the *seventh* element, i.e. 3. We will not show the swapping steps, and only show the final position of the sevent element.

  [**3**, 7, 8, 8, 10, 14, 16, 2, 4, 1]
- We do the same with the *eight* element, i.e. 2.

  [**2**, 3, 7, 8, 8, 10, 14, 16, 4, 1]
- Similary with the *nineth* element, i.e. 4. However, this element will stop swapping when it sees the number lower than itself, so it will stop when it sees 3.

  [2, 3, **4**, 7, 8, 8, 10, 14, 16, 1]
- Lastly, the *tenth* element, i.e. 1, will move all the way to the first position.

  [**1**, 2, 3, 4, 7, 8, 8, 10, 14, 16]

Design of Algorithm

Looking at the above case, we can try to write down our algorithm in pseudocode. Several things to note:

- There are two iterations in the steps above:
  1. *outer* iteration is moving from the *second* element to the last element in the list. What the outer iteration does is to place that n-th element into its position.
  2. *inner* iteration is swapping the n-th element until either:
     - it reaches the most left position, or
     - the number on its left is smaller
- The number of *outer* iteration is fixed, which is $n-1$. This is because it starts from the second element to the last element. So if there are n elements, it will repeat itself $n-1$ times.
- The *outer* iteration starts from the second element, which is index 1.
- The number of *inner* iteration is not fixed since it depends on the two cases stated above. The maximum number of iteration is when the number reaches the most left position. In this case for element at position i, it will repeat i times. If it sees a number that is smaller than itself, the number of iteration for the element at position i will be less than i.

Let's write it down.

```
Insertion Sort
Version: 1
Input: array
Output: None, sort in place
Steps:
1. n = length of array
2. For outer_index in Range(from 1 to n-1), do:
    2.1 inner_index = outer_index  # start with the i-th element
    2.2 As long as (inner_index > 0) AND (array[inner_index] < array[inner_index - 1]), do:
        2.2.1 swap(array[inner_index - 1], array[inner_index])
        2.2.2 inner_index = inner_index - 1  # move to the left
```

Optimised Insertion Sort

We can improve the algorithm slightly by reducing the number of assignment in the inner loop. This means that instead of swapping and assigning elements in the *inner* iteration, we only assign the element once it finds its final position. To do this, we store the element we are going to move into a temporary variable.

element we are going to move into a temporary variable.

- Let's illustrate this when the *outer* iteration is moving the *sixth* element, i.e. 8.

  [7, 8, 10, 14, 16, **8**, 3, 2, 4, 1]
- Instead of swapping (16, 8) pair, we store 8 into a temporary variable. Then we compare the temporary variable with 16. Since $8 < 16$, we simply shift 16 to the right. We indicate the position to be replaced with an underscore below. Since no swap is being done, the old value remains after the shift.

  [7, 8, 10, 14, **16**, `_8_`, 3, 2, 4, 1] , temporary = 8

  [7, 8, 10, 14, `_16_`, **16**, 3, 2, 4, 1] , temporary = 8
- We then now compare, the temporary variable with 14. Since $ 8 < 14$, we shift 14 to the right.

  [7, 8, 10, **14**, `_16_`, 16, 3, 2, 4, 1] , temporary = 8

  [7, 8, 10, `_14_`, **14**, 16, 3, 2, 4, 1] , temporary = 8
- We do the same with 10.

  [7, 8, **10**, `_14_`, 14, 16, 3, 2, 4, 1] , temporary = 8

  [7, 8, `_10_`, **10**, 14, 16, 3, 2, 4, 1] , temporary = 8
- Now, we compare the temporary variable with 8. But since 8 is not less than the value in the temporary variable, we do not swap, and store the temporary value to element on the right of it. We can then stop the *inner* iteration, and move to the next pass of *outer* iteration.

  [7, **8**, `_10_`, 10, 14, 16, 3, 2, 4, 1] , temporary = 8

  [7, 8, **8**, 10, 14, 16, 3, 2, 4, 1] , temporary = 8

This is an animation example for a different sequence of number from Wikipedia.

We can then modify the pseudocode as follows.

```
Insertion Sort
Version: 2
Input: array
Output: None, sort in place
Steps:
1. n = length of array
2. For outer_index in Range(from 1 to n-1), do:
    2.1 inner_index = outer_index  # start with the i-th element
    2.2 temporary = array[inner_index]
    2.3 As long as (inner_index > 0) AND (temporary < array[inner_index - 1]), do:
        2.3.1 array[inner_index] = array[inner_index - 1]) # shift to the right
        2.3.2 inner_index = inner_index - 1  # move to the left
    2.4 array[inner_index] = temporary # save temporary to its final position
```

Note:

- The additional step is on 2.2 where we store the i-th element to a temporary variable.
- Step 2.3 condition is modified to compare the temporary variable with the element pointed by `inner_index - 1`.
- Step 2.3.1 now only have one assignment instead of two.
- We only assign the temporary to the final position in step 2.4.
- Note that the position stored in 2.4 is `inner_index` instead of `inner_index - 1`. The reason is that in the last iteration, we have executed 2.3.2 which reduces the index by one.

we have executed 2.3.2 which reduces the index by one.
- Assignment from temporary variable to the array only happens at the end of *inner* iteration.

References

- https://www.geeksforgeeks.org/bubble-sort/
- https://en.wikipedia.org/wiki/Bubble_sort
- https://www.geeksforgeeks.org/insertion-sort/
- https://en.wikipedia.org/wiki/Insertion_sort