COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 13: Sorting(1)

Lecturer: Dr. Andrew Hines *Scribes: Yao Lu, Jingyuan Feng*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

13.1 Outline

This session introduces sorting algorithms: Bubble, Selection, Insertion (Quick and Merge). It shows how it can be used to make sequence n of elements in no particular order into sequence rearranged in increasing order of elements. The sorting algorithm is fundamental to many real-world applications(e.g. online shopping sort by price/category/colour) While the sorting speed depends on the algorithm and the initial data state.

13.2 Bubble Sort

13.2.1 What is Bubble Sort Algorithm?

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent pairs and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. Bubble sort can be practical if the input is in mostly sorted order with some out-of-order elements nearly in position.

Bubble sort has a worst-case and average complexity of (n2), where n is the number of items being sorted. Most practical sorting algorithms have substantially better worst-case or average complexity, often O(n log n). Even other (n2) sorting algorithms, such as insertion sort, generally run faster than bubble sort, and are no more complex. Therefore, bubble sort is not a practical sorting algorithm.

The only significant advantage that bubble sort has over most other algorithms, even quick sort, but not insertion sort, is that the ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only O(n). By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort share this advantage, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.

Advantages:

- It's a simple algorithm that can be implemented on a computer.
- Efficient way to check if a list is already in order.
- Doesn't use too much memory.

Disadvantages:

- It's an efficient way to sort a list.
- Due to being efficient , the bubble sort algorithm is pretty slow for very large lists of items.

13.2.2 How Does Bubble Sort Work?

Looking at the example in Algorithm 5 below, bubble sort carries out n 1 passes through the list. For each pass, it carries out a sweep of n 1 comparison exchanges, left to right. Bubble sort sorts a sequence (ADT) of values, based on a structured pattern of comparison-exchange (CE) operations : Take value in two adjacent slots in the sequence and if the values are out of order (i.e. the larger before the smaller), then swap them around.

Algorithm 1: Pseudo-code for bubble sort algorithm

```
1 Algorithm bubble sort
  Input  : A an array of n elements
  Output: A is sorted
2  for s = 1 to n-1 do
3     for current = 0 to n-2 do
4        if A[current] >A[current + 1] then
5           swap A[current] and A[current + 1]
6        endif
7     endfor
8  endfor
```

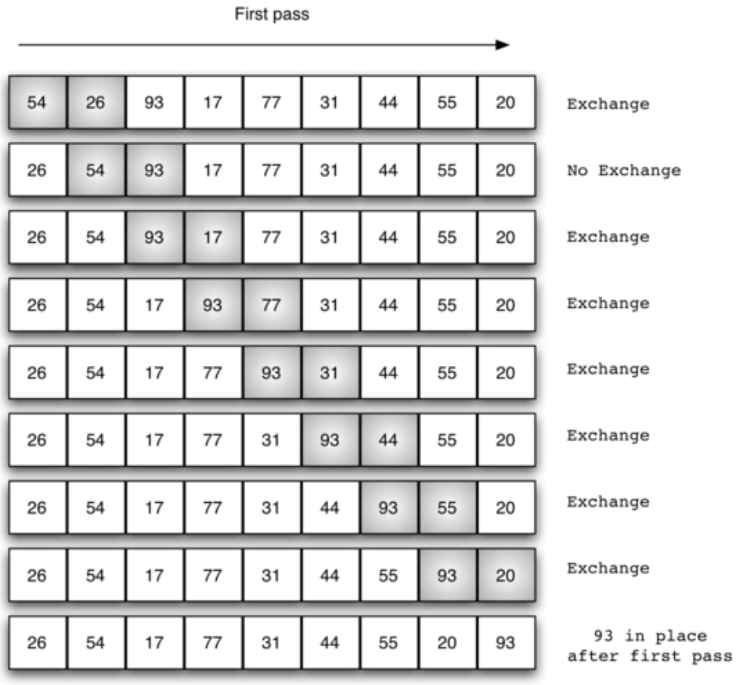

Figure 13.1: Bubble Sort

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item bubbles up to the location where it belongs.

Figure 1 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are n1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

At the start of the second pass, the largest value is now in place. There are n1 items left to sort, meaning that there will be n2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be n1. After completing the n1 passes, the smallest item must be in the correct position with no further processing required. The code below shows the complete bubbleSort function. It takes the list as a parameter, and modifies it by exchanging items as necessary.

The exchange operation, sometimes called a swap, is slightly different in Python than in most other programming languages. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location). A code fragment such as

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

will exchange the ith and jth items in the list. Without the temporary storage, one of the values would be overwritten.

In Python, it is possible to perform simultaneous assignment. The statement a,b=b,a will result in two assignment statements being done at the same time (see Figure 2). Using simultaneous assignment, the

exchange operation can be done in one statement.

The code below perform the exchange of the i and (i+1)th items using the threestep procedure described earlier. Note that we could also have used the simultaneous assignment to swap the items.

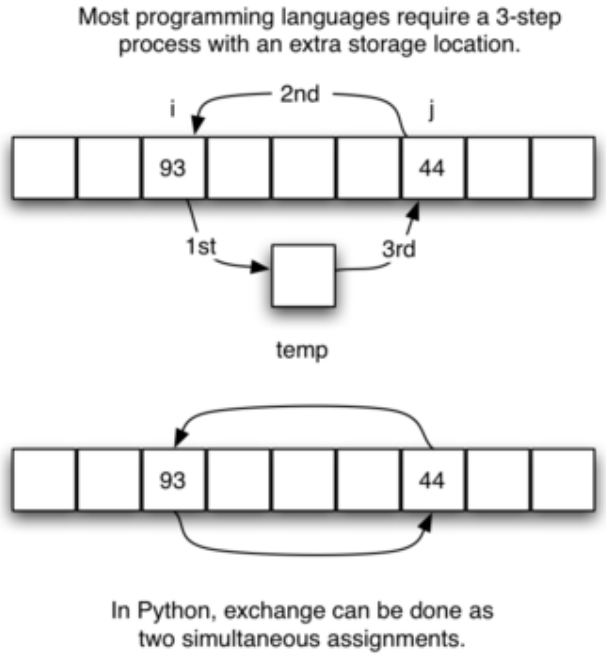

Figure 13.2: Swap items in Python

The following code example shows the complete bubble sort function working on the list shown above.

```
1 def bubbleSort(alist):
2     n = len(alist)
3     for s in range(1,n):
4         for i in range(0,n-1):
5             if alist[i]>alist[i+1]:
6                 temp = alist[i]
7                 alist[i] = alist[i+1]
8                 alist[i+1] = temp
9
10 alist = [27, 13, 44, 15, 12, 99, 63, 57]
11 bubbleSort(alist)
12 print(alist)
13
```

Figure 13.3: Python implementation example for Bubble Sort

13.2.3 Optimising Bubble Sort (1)

When the array is sorted, we can stop. In the example below, we are sorted after pass 4 has completed.

- List: 27, 13, 44, 15, 12, 99, 63, 57
- Pass 1: 13 27 15 12 44 63 57 **99**
- Pass 2: 13 15 12 27 44 57 **63 99**
- Pass 3: 13 12 15 27 44 **57 63 99**
- Pass 4: 12 13 15 27 **44 57 63 99**
- Pass 5: 12 13 15 **27 44 57 63 99**
- Pass 6: 12 13 **15 27 44 57 63 99**
- End: 12, 13, 15, 27, 44, 57, 63, 99

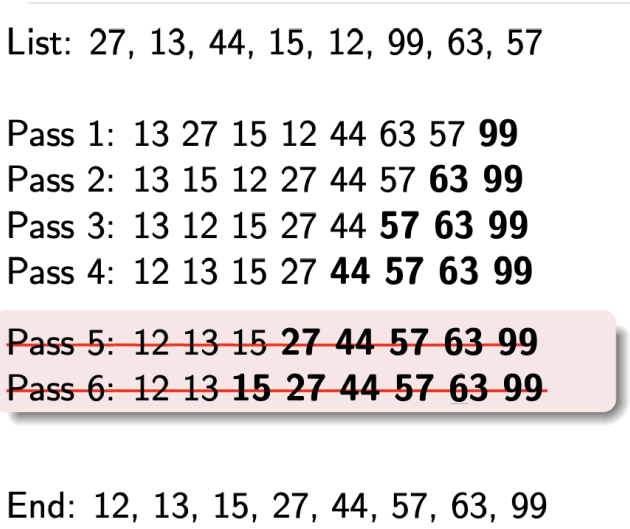

Figure 13.4: Optimised Bubble Sort (1)

Algorithm 2: Pseudo-code for bubble sort algorithm optimization

```
1  Algorithm bubble sort
   Input  : A an array of n elements
   Output: A is sorted
2  for s = 1 to n-1 do
3     swapped ⟸ False
4     for current = 0 to n-2 do
5        if A[current] >A[current + 1] then
6           swap A[current] and A[current + 1]
7           swapped ⟸ True
8        endif
9     endfor
10    if not swapped then
11       finish
12    endif
13 endfor
```

13.2.4 Optimising Bubble Sort (2)

After the ith pass the last (i-1) items are sorted: no need to keep evaluating them each pass.

- List: 27, 13, 44, 15, 12, 99, 63, 57
- Pass 1: 13 27 15 12 44 63 57 **99**
- Pass 2: 13 15 12 27 44 57 **63 99**
- Pass 3: 13 12 15 27 44 **57 63 99**
- Pass 4: 12 13 15 27 **44 57 63 99**
- Pass 5: 12 13 15 **27 44 57 63 99**
- Pass 6: 12 13 **15 27 44 57 63 99**
- End: 12, 13, 15, 27, 44, 57, 63, 99

Algorithm 3: Pseudo-code for bubble sort algorithm optimization

```
1  Algorithm bubble sort
   Input  : A an array of n elements
   Output: A is sorted
2  for s = 1 to n-1 do
3     swapped ⟸ False
4     for current = 0 to n - s - 2 do
5        if A[current] >A[current + 1] then
6           swap A[current] and A[current + 1]
7           swapped ⟸ True
8        endif
9     endfor
10    if not swapped then
11       finish
12    endif
13 endfor
```

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These wasted exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop. The code below shows this modification, which is often referred to as the short bubble.

```python
def shortBubbleSort(alist):
    exchanges = True
    n = len(alist)
    for s in range(0,n):
        while exchanges:
           exchanges = False
           for i in range(0,n-s-1):
               if alist[i]>alist[i+1]:
                   exchanges = True
                   temp = alist[i]
                   alist[i] = alist[i+1]
                   alist[i+1] = temp


alist=[27, 13, 44, 15, 12, 99, 63, 57]
shortBubbleSort(alist)
print(alist)

```

Figure 13.5: Python implementation example for Optimised bubble sort (2)

13.3 Selection Sort

13.3.1 What is Selection Sort?

As the name suggests, selection sort means picking or selecting the smallest (or the biggest in descending sort) element from the list and placing it in the sorted portion of the list. The first element is initially considered to be the minimum element and then compared with other elements. During these comparisons, if a smaller element is emerge then it will be considered as the newest minimum. After completion of one full round, the smallest element found is swapped with the first element. This process continues until all the elements are sorted.

13.3.2 How Does Selection Sort Work?

Step1: Selection sorting algorithm first compare the first two elements of the array and exchange them if necessary, this means if you want to sort the array elements in ascending order, if the first element is larger than the second one, you need to exchange them, but if the first elements are less than seconds, just leaving the elements where they are. Then, if necessary, compare and exchange the first element and the third element again. Continue this process till the first and last elements of the array are compared. This completes the first step of selecting the sort.

Step 2: If there are n elements to be sorted, the above process should be repeated n-1 times to obtain the desired outcome. For a better performance, the comparison starts with the second element in the second step, because after the first step, the smallest element will appear at the first position, and if the desired result is in descending order, the largest element will be the first one.

Step 3: Similarly, in the third step, the comparison starts with the third element, and so on.

The figure below clearly explains the working of selection sort algorithm:

List: 27, 13, 44, 15, 12, 99, 63, 57

27	13	44	15	12	99	63	57
12	13	44	15	27	99	63	57
12	**13**	44	15	27	99	63	57
12	**13**	**15**	44	27	99	63	57
12	**13**	**15**	**27**	44	99	63	57
12	**13**	**15**	**27**	**44**	99	63	57
12	**13**	**15**	**27**	**44**	**57**	63	99

End: **12, 13, 15, 27, 44, 57, 63, 99**

Figure 13.6: Selection Sort

13.3.3 A Real Life Example

Like mentioned before, sorting algorithms are very useful. For example, when we arrange the names in a phone book, when we plan the dates to travel or when we sort items in different order during online shopping.

Suppose you have a collection of music on one of your play lists. For each artist, you have a play count. What you will do if you want to sort this list from most to least played, so that you can rank your favorite artists? And how can you do it? By implementing selection sort, one way is to go through the whole list and find the artist with the highest play count then add that artist to a new list. Do it again to find the next-most-played artist. Keep doing this procedure until you end up with a sorted list:

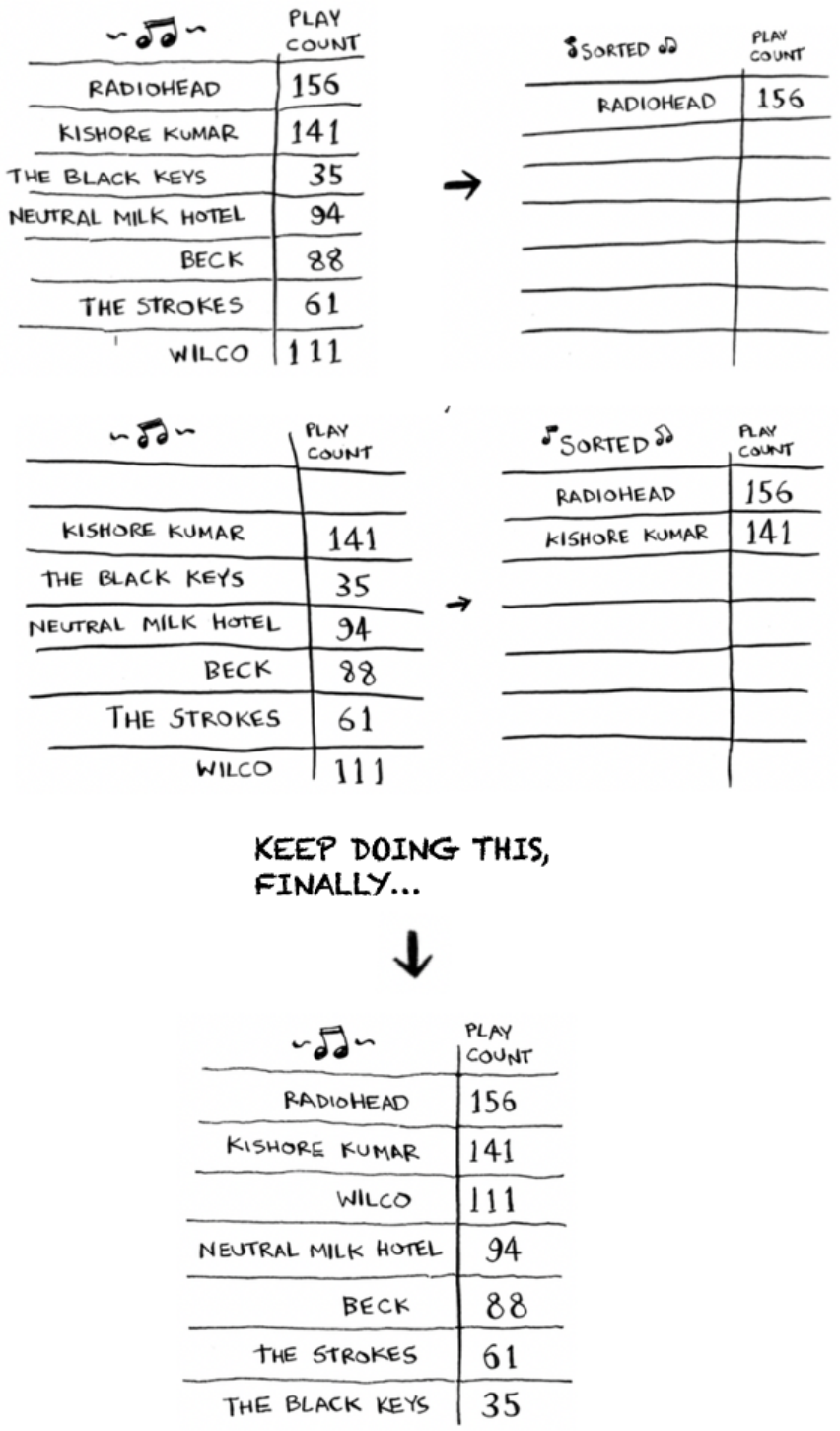

Figure 13.7: Play list example (1)

As you just saw, to find the most-played artist, you have to check each element in the play list. When put

it on our sorting algorithm hats to see how long this will take to run, you need to remember that you touch every element in a list once. i.e. running simple search over the list of artists looks at each artist once. To find the artist with the highest play count, you have to check each item in the list. This takes O(n) time, so you have an operation that takes O(n) time, and you have to do that n times like the figure showed below:

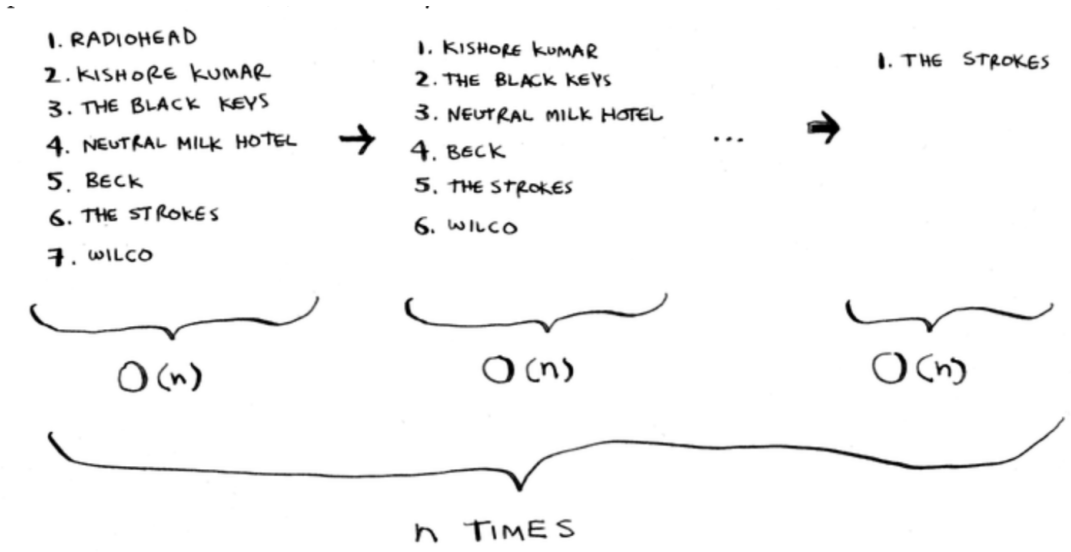

Figure 13.8: Play list example (2)

Obviously, this takes O(n * n) time.

13.3.4 Time Complexity

- The estimate running time for selection sort is T(n) = 10(n)2 + 2n.
- The best-case, average-case and worst-case complexity are all O(n2).

13.3.5 Analysis Of Selection Sort

Regardless of the condition, the time complexity of selection sort is O(n2), so its runtime is always quadratic. Thus, some suggest that selection sort should never be used since it does not adapt to the data in any way.

However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort may be the appropriate algorithm to choose.

In addition, selection sorting algorithm is also easy to use, and works best with a small number of elements. If performance is matters to the program, selection sort should not be used to sort large numbers of elements.

13.3.6 Pseudo-code for Selection Sort

Algorithm 4: Pseudo-code for selection sort

```
1  function selection_sort (array);
   Input  : A an array of n elements
   Output: A is sorted
2  for j = 0 to n-2 do
3  |  min ← j
4  |  for i = j +1 to n-1 do
5  |  |  if A[min] > A[i] then
6  |  |  |  min ← i
7  |  |  endif
8  |  endfor
9  |  swap a[min], a[j]
10 endfor
```

13.3.7 Python Implementation: Selection Sort

```
def selection_sort(alist):
    n = len(alist)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if alist[j] < alist[min_index]:
                min_index = j
        if min_index != i:
            alist[i], alist[min_index] = alist[min_index], alist[i]
    print(alist)

selection_sort([5,2,6,9,1])
```

```
[1, 2, 5, 6, 9]
```

Figure 13.9: Python implementation example for selection sort

13.4 Insertion Sort

13.4.1 What is Insertion Sort?

In insertion sort, an element gets compared and inserted into the correct position in the list. Similar to selection sort, we started regard the first element of the list as the sorted portion and the rest to be unsorted. As the algorithm proceeds, the sorted portion will increase and the unsorted portion will keeping shrink. In another words, to apply this sort, the first element is considered as the sorted portion and the other elements of the list to be unsorted. Then compare each element from the unsorted portion with the element(s) in the sorted portion and insert it in the correct position in the sorted part if necessary. The most important things is in insertion sort algorithm, the sorted portion of the list remain sorted at all times.

13.4.2 How does Insertion Sort Work?

So, how does insertion sort works suppose you want to arrange the elements in ascending order?

Step 1: Compare the second element of the array with the element that appears in front of it (at the beginning, just the first element). If the first element is bigger than the second element, the second element will be inserted into the position of the first element. Otherwise, just keep them where they are. After this step, the first two elements of the array are sorted.

Step 2: Then compare the third element of the array with the element before it (the first and second elements). If the third element is smaller than the first element, it will be inserted in the position of the first element. If the third element is larger than the first element but smaller than the second element, it will be inserted into the position where the second element is. If the third element is larger than first two elements, keep it as it is. After this step, the first three elements of the array are sorted.

Step 3: Similarly, the third step compares the fourth element of the array with the elements preceding it (the first, second, and third elements). Apply the same procedures repeatedly and insert the element into the correct position when necessary until the array are completely sorted.

The figure below shows the working process of insertion sort algorithm:

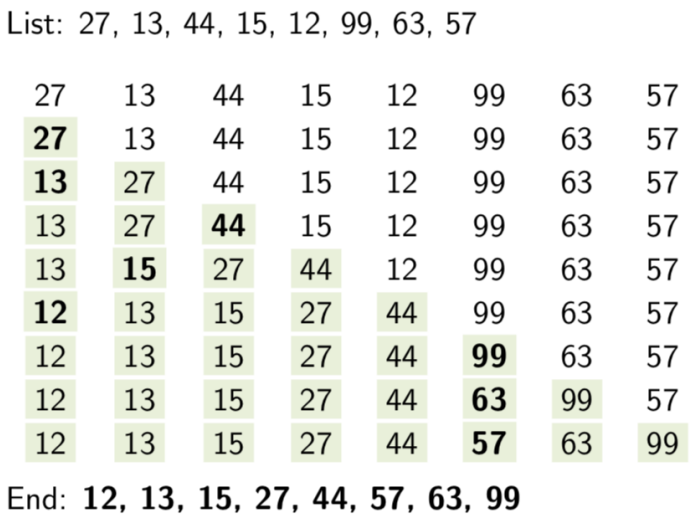

Figure 13.10: Insertion Sort

13.4.3 Time Complexity

- The estimate running time for insertion sort is T(n) = 10(n)2 + 2n.
- The best-case complexity is O(n).
- The average-case and worst-case complexity is O(n2).

13.4.4 Analysis of Insertion Sorting Algorithms

As the number of elements in the sorted portion grows, the new element from the unsorted portion have to compare with all the elements in the sorted portion before insertion.

Insertion Sort works best with small number of elements. Similar to bubble sort and selection sort, the average case and worst case run time complexity of insertion sort is O(n2). Even though the best case time complexity for both bubble sort and insertion sort are O(n), insertion sort is considered better than bubble sort. To compare with selection sort, the inner loop of insertion sort can be terminated earlier when the element is already in the correct position. Thus, the time complexity of insertion sort can reach the level of O(n) in the best case.

13.4.5 Pseudo-code for Insertion Sort

Algorithm 5: Pseudo-code for insertion sort

```
1 function insertion_sort (array);
  Input  : A an array of n elements
  Output: A is sorted
2 for j = 1 to n-1 do
3 |   i ← j
4 |   while i > 0 and A[i - 1]>A[i] do
5 |   |   swap a[i] and a[i - 1]
6 |   |   i ← i-1
7 |   endwhile
8 endfor
```

13.4.6 Python Implementation: Insertion Sort

```
def insertion_sort(alist):
    n = len(alist)
    for i in range(1, n):
        current = alist[i]
        while i >= 0 and alist[i-1] > current:
            alist[i-1], alist[i] = alist[i], current
    print(alist)

insert_sort([5,2,6,9,1])
```

```
[9, 6, 5, 2, 1]
```

Figure 13.11: Python implementation example for insertion sort

13.5 Summary

Sorting algorithm is everywhere and there are many different implementations. It is one of the most important and well studied problem in the field of computer science. Many advanced algorithms for a wide range of problems rely on sorting as a subroutine.

This lecture focuses on three basic kinds of sorting algorithms - bubble sort, selection sort and insertion

sort. They are the foundation of some other sorting algorithms. In general, they all have same complexity: O(n2). Each could have the fastest runtime depending on the data.

The table below shows the complexity of all the sorting algorithms discussed in this lecture:

Sorting Algorithm	Best-case	Average-case	Worst-case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Optimised Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$

Table 13.1: Complexity for All the Sorting Algorithm Mentioned

The chosen of algorithm to use highly depend on the status of the initial dataset. Because all of them show different tendency when running on large datasets. In other words, they all work in different ways in different situation and on different dataset.

13.6 To Be Continued...

- Quick Sort
- Merge Sort

References

[1] Data Structures and Algorithms in Python
`Goodrich, M., Tamassia, R. and Goldwasser, M. (2013). Data structures and algorithms in Python. Hoboken, N.J.: Wiley.`

[2] Grokking Algorithms: An illustrated guide for programmers and other curious people
`Aditya, B. (2016).Manning Publications Co..`

[3] Python Data Structures Tutorial
`https://www.pythoncentral.io/series/python-data-structures-tutorial/`

[4] Sorting Algorithms Animations
`https://www.toptal.com/developers/sorting-algorithms`

[5] Time Complexities of all Sorting Algorithms
`https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/`