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

Lecture 13: Sorting: bubble sort, selection sort, insertion sort

Lecturer: Dr. Andrew Hines *Scribes: Jiaru He , Xuewen Tan*

13.1 Outline

Lecture 13 mainly focuses on sorting including Bubble, Selection, Insertion. Firstly, the concept of sorting is explained and then Andrew highlights these different sorting separately from theory to programming practice. Furthermore, the complexity of different sorting algorithms is demonstrated and compared as well. Lecture 13 will look at the meaning of sorting in general and different sorting in detail by theory explanation, programming practice and complexity comparison.

13.1.1 Sorting

Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. A list of cities could be sorted by population, by area, or by zip code. Sorting a large number of items can take a substantial amount of computing resources. Like searching, the efficiency of a sorting algorithm is related to the number of items being processed. For small collections, a complex sorting method may be more trouble than it is worth. The overhead may be too high. On the other hand, for larger collections, we want to take advantage of as many improvements as possible.

13.1.2 Sorting algorithm

Generic Sorting Algorithm includes:
Input: Sequence n of elements in no particular order.
Output: Sequence rearranged in increasing order of elements Values.
There are many, many sorting algorithms that have been developed and analyzed. This suggests that sorting is an important area of study in computer science. Before getting into specific algorithms, we should think about the operations that can be used to analyze a sorting process. First, it will be necessary to compare two values to see which is smaller (or larger). In order to sort a collection, it will be necessary to have some systematic way to compare values to see if they are out of order. The total number of comparisons will be the most common way to measure a sort procedure. Second, when values are not in the correct position with respect to one another, it may be necessary to exchange them. This exchange is a costly operation and the total number of exchanges will also be important for evaluating the overall efficiency of the algorithm.

13.2 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.

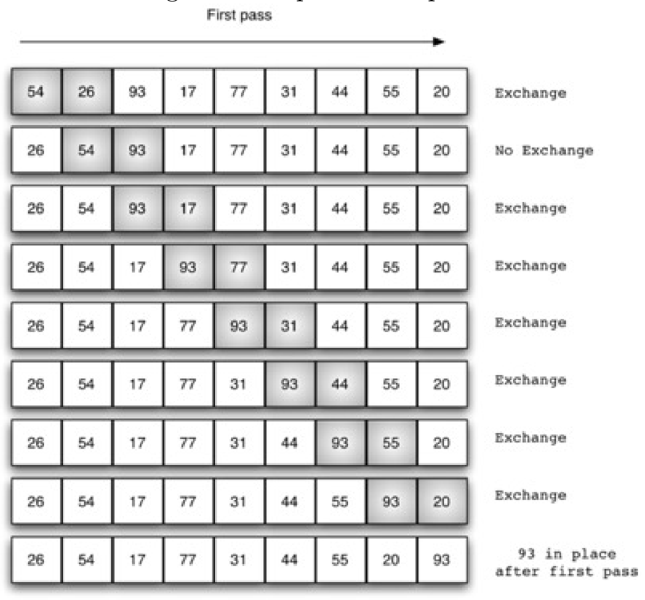

First pass
54 26 93 17 77 31 44 55 20 Exchange
26 54 93 17 77 31 44 55 20 No Exchange
26 54 93 17 77 31 44 55 20 Exchange
26 54 17 93 77 31 44 55 20 Exchange
26 54 17 77 93 31 44 55 20 Exchange
26 54 17 77 31 93 44 55 20 Exchange
26 54 17 77 31 44 93 55 20 Exchange
26 54 17 77 31 44 55 93 20 Exchange
26 54 17 77 31 44 55 20 93 93 in place after first pass

Figure 1 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.

Another example: First Pass: (1 5 4 2 8)$\rightarrow (14528), since 5 > 4$
$(14528) \rightarrow (14258), since 5 > 2$
$(14258) \rightarrow (14258)$
Now, since these elements are already in order (8 ¿ 5), algorithm does not swap them.
Second Pass:
(1 4 2 5 8) $\rightarrow (14258)$
$(14258) \rightarrow (12458), since 4 > 2$
$(12458) \rightarrow (12458)$
$(12458) \rightarrow (12458)$
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.
Third Pass:
$(12458) \rightarrow (12458)$
$(12458) \rightarrow (12458)$
$(12458) \rightarrow (12458)$
$(12458) \rightarrow (12458)$

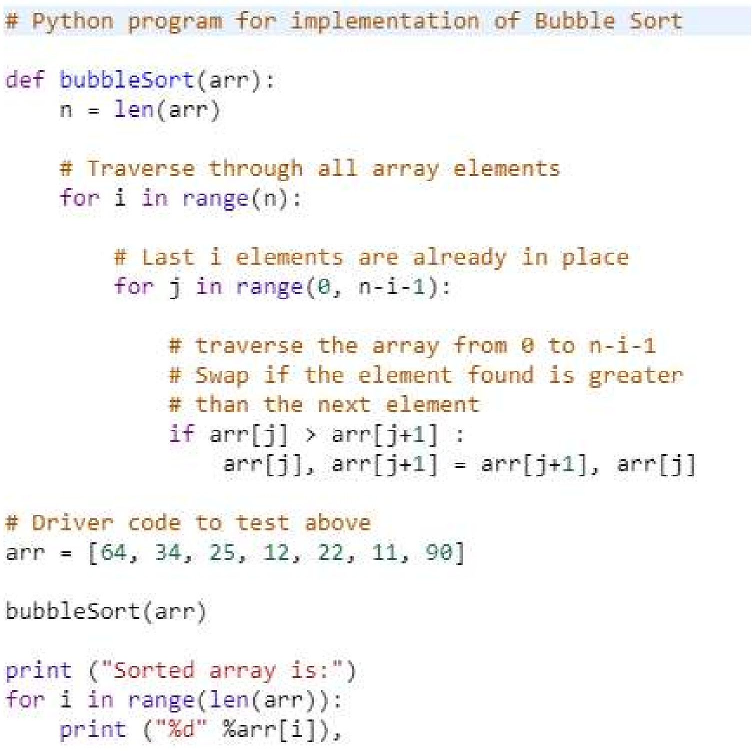
From the example we can find some weakness and the optimizing pseudocode will be presented in next part.

13.2.1 Pseudocode

```
Algorithm bubble_sort
Input:  $A$ an array of n elements
Output:  $A$ is sorted
for $s$ = 0 to n-1 do # Passes
    for $current$ = 0 to n-2 do # Sweeps
        if $A[current] > A[current+1]$ then
          # Comparison Exchange (CE)
          swap $A[current]$ and $A[current+1]$
        endif
    endfor
endfor
```

13.2.2 Python implementation

Before optimization:

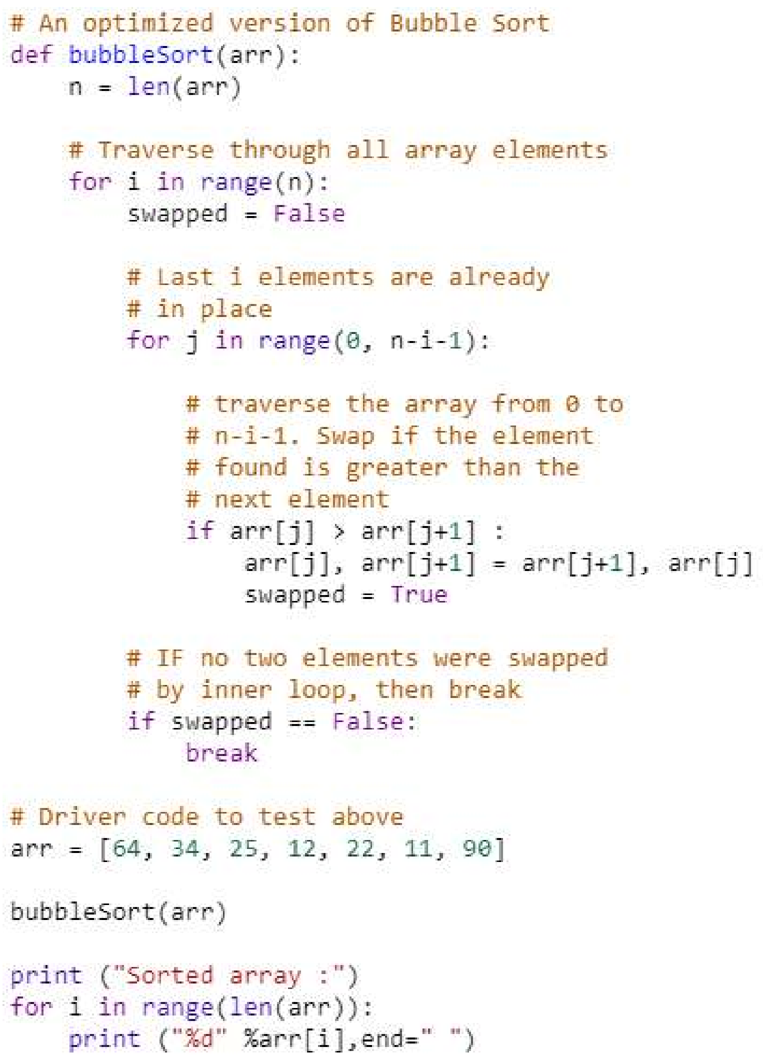


```python
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array is:")
for i in range(len(arr)):
    print ("%d" %arr[i]),
```

13.2.3 Complexity

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm. In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines (Source: Wikipedia).

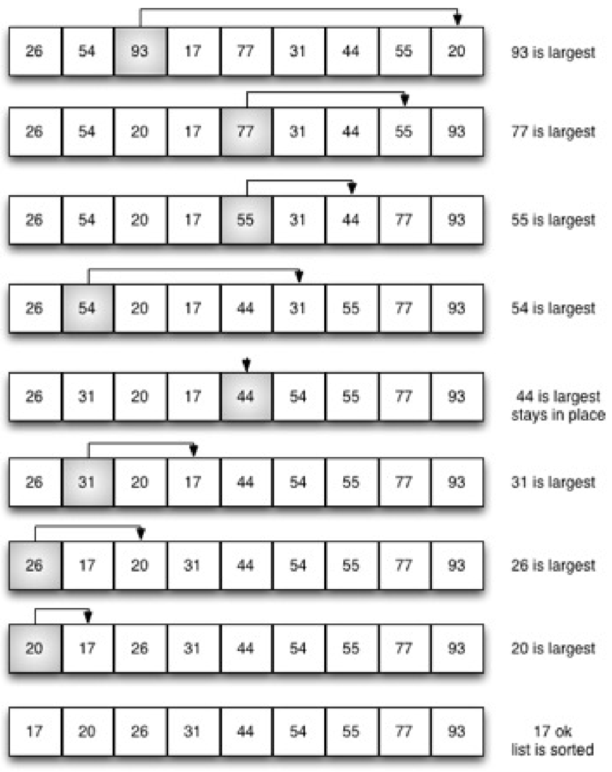
13.2.4 Optimizing Bubble



```python
# An optimized version of Bubble Sort
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already
        # in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to
            # n-i-1. Swap if the element
            # found is greater than the
            # next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True

        # IF no two elements were swapped
        # by inner loop, then break
        if swapped == False:
            break

# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print ("Sorted array :")
for i in range(len(arr)):
    print ("%d" %arr[i],end=" ")
```

13.3 Selection Sort

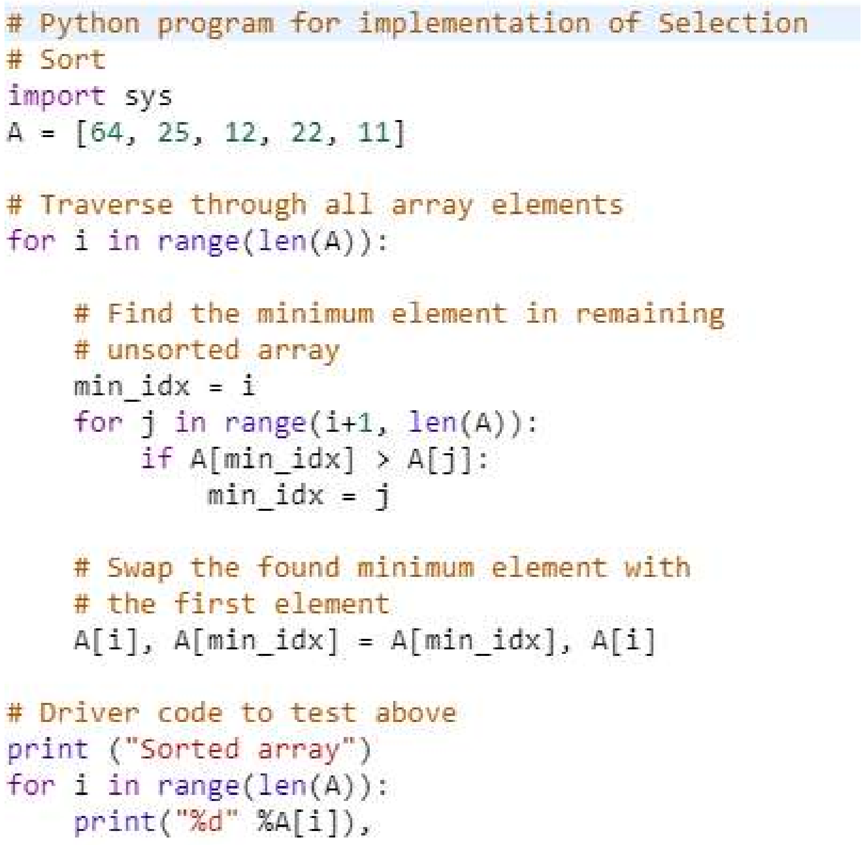
The selection sort improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n1n1 passes to sort n items, since the final item must be in place after the (n1)(n1) st pass.
Figure 2 shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on.



13.3.1 Pseudocode

```
Algorithm selection_sort
Input:  $A$ an array of n elements
Output:  $A$ is sorted
for $j$ = 0 to n-2 do # For each element in the array
    min ← j # Find the min and swap
    for $i$ = j +1 to n -1 do
        if $A[min] > A[i]$ then
            min ← i
        endif
    endfor
    swap a[min], a[j]
endfor
```

13.3.2 Python implementation



```
# Python program for implementation of Selection
# Sort
import sys
A = [64, 25, 12, 22, 11]

# Traverse through all array elements
for i in range(len(A)):

    # Find the minimum element in remaining
    # unsorted array
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j

    # Swap the found minimum element with
    # the first element
    A[i], A[min_idx] = A[min_idx], A[i]

# Driver code to test above
print ("Sorted array")
for i in range(len(A)):
    print("%d" %A[i]),
```

13.3.3 Complexity

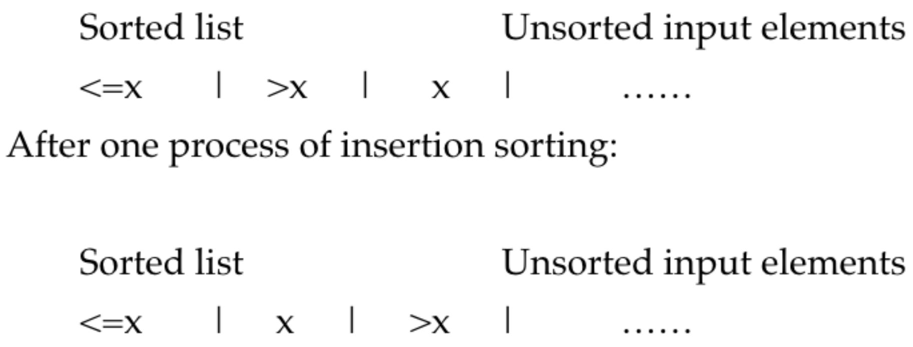
Time Complexity: O(n2) as there are two nested loops. You may see that the selection sort makes the same number of comparisons as the bubble sort and is therefore also $O(n^2)$.
However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies.

13.4 Insertion sort

Insertion sorting is a simple and efficient sorting using comparison. This algorithm has similar idea with selection sorting, which increase the sorted section at the start of the array, takes the next item and inserts it into the correct position in the list being sorted. This process is repeated until all input elements have been traversed.

13.4.1 Implementation of algorithm:

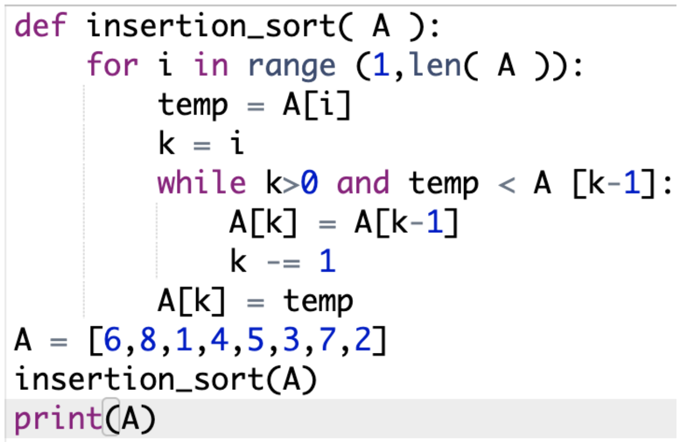
Insertion sorting removes an element from the input array of data and inserts it into the correct position in the list being sorted. This process is repeated until all input elements have been traversed.
illustration:



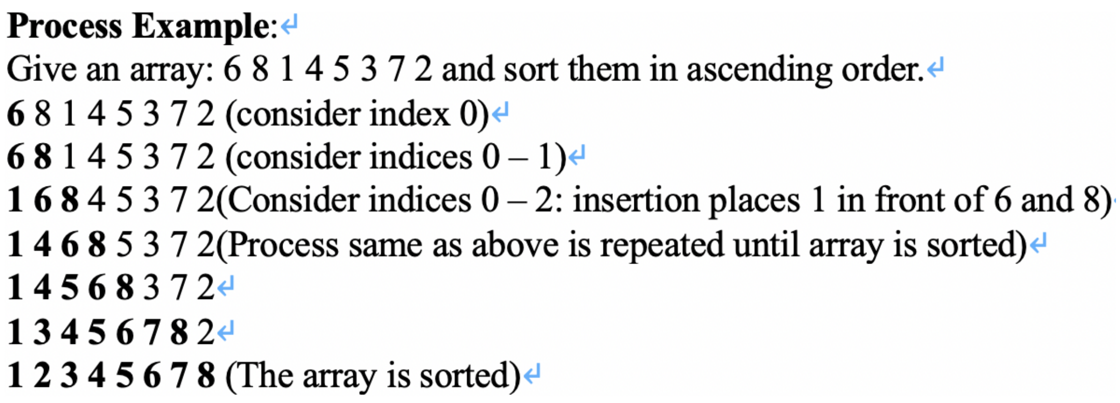
13.4.2 Pseudocode

```
Algorithm insertion_sort
Input:  $A$ an array of n elements
Output:  $A$ is sorted
for j = 1 to n-1 do # For each element in the array
    i ← j # Push right until element inserted
    while i > 0 and A[i-1] > A[i] do
        swap a[i] and a[i-1]
        i ← i - 1
    endwhile
endfor
```

13.4.3 Python implementation



```
def insertion_sort( A ):
    for i in range (1,len( A )):
        temp = A[i]
        k = i
        while k>0 and temp < A [k-1]:
            A[k] = A[k-1]
            k -= 1
        A[k] = temp
A = [6,8,1,4,5,3,7,2]
insertion_sort(A)
print(A)
```



Process Example:
Give an array: 6 8 1 4 5 3 7 2 and sort them in ascending order.
6 8 1 4 5 3 7 2 (consider index 0)
6 8 1 4 5 3 7 2 (consider indices 0 – 1)
1 6 8 4 5 3 7 2(Consider indices 0 – 2: insertion places 1 in front of 6 and 8)
1 4 6 8 5 3 7 2(Process same as above is repeated until array is sorted)
1 4 5 6 8 3 7 2
1 3 4 5 6 7 8 2
1 2 3 4 5 6 7 8 (The array is sorted)

13.4.4 Complexity analysis

Worst case: Worst case occurs when the inner loop has to move all elements A[1],,A[i-1] (which happens when A[i] = key is smaller than all other elements in the inner loop, this takes O(i-1) time.

$$T(n) = O(1) + O(2) + O(3) + \cdots + O(n-1)$$
$$= O(1 + 2 + 3 + \cdots + n - 1) = O(\frac{n(n-1)}{2}) \approx O(n^2)$$

Average case: For the average case, the inner loop will insert A[i] in the middle of A[1],,A[i-1], which takes O(i/2) time.

$$T(n) = \sum_{i=1}^{n} O(\frac{i}{2}) \approx O(n^2)$$

13.4.5 Performance of algorithm:

If every element is greater than or equal to every element to its left, the running time of insertion sort is O(n). This situation occurs if the array starts with all elements sorted.

Worst case complexity: O(n^2)
Best case complexity: O(n)
Average case complexity: O(n^2)

13.4.6 Merge sorting and Quick sorting

These two sorting algorithms both belong to divide and conquer strategy.

13.4.7 Reference

https://www.geeksforgeeks.org/
http://interactivepython.org/runestone/static/pythonds/index.html/
Miller, B.N. and Ranum, D.L., 2011. Problem solving with algorithms and data structures using python Second Edition. Franklin, Beedle Associates Inc..
Karumanchi, N., 2015. Data Structure and Algorithmic Thinking with Python: Data Structure and Algorithmic Puzzles. CareerMonk Publications.