TEMASEK JUNIOR COLLEGE

Temasek Junior College

JC H2 Computing

Problem Solving & Algorithm Design 14A – Introduction to Sorting algorithms

1 Introduction to Sorting algorithms

Sorting is the operation of arranging the records of a table into some **sequential order** according to an **ordering criterion**.

9	JOYCE TAN WAN LIN	65
1	NG HUI TING JACQUELINE	45
3	NUR RAMIZAH BTE RAMLI	77
6	TANG KUAN YEE	67
7	KAW TECK LIN	30
5	KUNG GUANGJUN	90
2	LOE CHUAN YUN	71
4	OH JIEYI JOEL TIMOTHY	68
10	PANG YINGXIANG BONNER	88
8	TAY KAI ZHONG	56
-1		999

Data file: COMPUTING_SCORES

- The sort is performed according to the **key value** of each record.
- Depending on the makeup of the key, records can sorted either **numerically** or, more generally, **alphanumerically**. In numerical sorting, the records are **arranged in ascending or descending order according numerical value of the key**.
- In most cases, the sorting criterion/ordering relations considers only a part of a record (i.e. record key) e.g. A record is made up of: Class register number, Name, Exam_score
- Sort records based on record key: Exam_score
- It is more efficient to sort records based on a key instead of the entire record.
- In general, a key can be any sequence of characters, and the ordering imposed by sorting depends on the collating sequence associated with the particular character set which is being used.

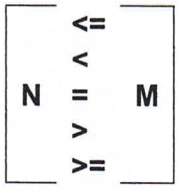
1.1 Operations involved in sorting

In general simple sorting algorithms perform 2 basic operations:

1. Compare operation,
2. Swap operation.

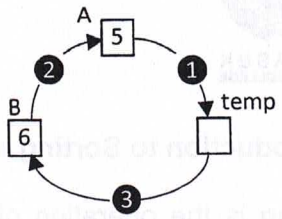
These operations repeats over and over again until the data is entirely and correctly sorted based on the sorting criterion/ordering relation.

1.1.1 The Comparison Operation



- Determines the order of an element.
- For instance,
  - Sorting a sequence of numbers from smallest to largest, the comparison operation determine to place the number with the least value first.
  - Sorting a sequence of characters alphabetically, the comparison operation determine to place 'a' before 'b', 'b' before 'c', and so on. [Internally using value of code, e.g. **ASCII code**]
- Sorting algorithm must usually perform many comparisons in order to perform the operation correctly.

1.1.2 The Swap Operation



- Assisting computer moves/order elements during the sorting process.
- Every swap operation performed repeatedly takes us a step closer towards a correctly sorted output.
- SWAP(A,B): operation requires copy operations
  - Assign value of A to temp.
  - Assign value of B to A.
  - Assign value of temp to B.

1.2 Uses of Sorting

- We need sorting because
  - The data in sorted order is required
  - We need to perform binary search or index a database
  - It is the initialization step of many algorithms.

- **Examples**
  - Order components [file names, dates, size, etc] in a **computer folder**
  - **Electronic mailbox**: allows users to sort their emails in different ways (by date received, by subject line, by sender, etc).
  - **Telephone Directory**: stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

1.3 Advantages of Sorting

- If data is stored and sorted in a pre-defined order, data searching can be optimized. [Why?]
- By sorting data, information can be made more readable.
- Processing of data can be performed in a defined order.

1.4 Sorting can be classified as 2 types:

1.4.1 In-place sort:

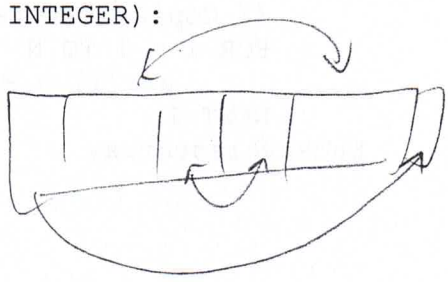
- Performed when the number of elements is small enough to fit into the main memory.
- In order to produce the desired output, modification to the data set/sequence only requires only small and constant extra space.
- Sorting is done by modifying the order of the elements within the sequence.
- Insertion sort, Bubble sort, Quick sort

An in-place algorithm is an algorithm that does not need an extra space and produces an output in the same memory that contains the data by transforming the input 'in-place'. However, a small constant extra space used for variables is allowed.

[In-Place Algorithm] Example

```
//To reverse the items of an array and store the reversed results
//back into itself
Procedure RevereseArray(Arr AS ARRAY: INIEGER, N: INTEGER):
      //N is the size of array
      DECLARE TEMP: INTEGER
      FOR I = 1 TO int(N / 2)
            #swaps_values(Arr[I], Arr[N - (I - 1)])
            TEMP ← Arr[I]
            Arr[I] ← Arr[N - I + 1]
            Arr[N - (I - 1)] ← TEMP
      NEXT I
EndRevereseArray

//main program
arr ←  {1, 2, 3, 7, 8, 9}
N ←  len(arr)
Print(arr)
RevereseArray(arr, N);
Print("Reversed array is")
Print(arr)
```



1.4.2 Not-In-place sort:

- When all elements that needs to be sorted cannot be placed in memory at a time, therefore additional memory is required in order to perform the sorting.
- External Sorting is used for massive amount of data.
- Typical Merged sort.

[NOT In-Place Algorithm] Example

```
//To reverse the items of an array to a new array in function
//and store the reversed results back into itself
Procedure ReverereseArray(arr, N): //N is the size of array
      // Creates a copy of the array and store its reversed elements
      Rev = new Array of [1..N]: INTEGER

      FOR I = 1 TO N
            Rev[N - I + 1] = arr[I]
      NEXT I
      // Copies reversed elements back to arr[] as results
      FOR I = 1 TO N
            arr[I] = Rev[I]
      NEXT I
EndRevereseArray
```

2 Bubble Sort

The strategy is to start at the beginning of the list and compare pairs of data items as it moves down to the end. Each time the items in the pair are out of order, the algorithm swaps them. This process has the effect of bubbling the largest items to the end of the list. The algorithm then repeats the process from the beginning of the list and goes to the next to last item, and so on, until it begins with the last item. At that point, the list is sorted.

In each **pass** of the bubble sort [**General illustration**]:

1. The item in the first position is compared with the item in the second position. If they are out of order, they are exchanged; if they are in order, they are left alone.
2. The item now in the second position is compared with the item in the third position. If they are out of order, they are exchanged; if they are in order, they are left alone.
3. The item now in the third position is compared with the item in the fourth position....
4. The steps in the pass continue until each item has been checked.

After the first complete **pass**, the last item will be in the correct position. After the second, the last two items will be in the correct position, and so on. After **N - 1 passes**, each of N items will be in the correct position (the final pass determines the position of the final two items).

Example 1 Bubble Sort Version 1 algorithm

The accompanying table shows a bubble sort to sort five numbers in ascending order.

Position of array elements		**0 1 2 3 4**
Original Order		**4 7 3 1 6**
		Result
First Pass		
a.	4 is compared with 7, no exchange	4 7 3 1 6
b.	7 is compared with 3, exchange	4 3* 7* 1 6
c.	7 is compared with 1, exchange	4 3 1* 7* 6
d.	7 is compared with 6, exchange	4 3 1 6* 7*
Second Pass		
a.	4 is compared with 3, exchange	3* 4* 1 6 7
b.	4 is compared with 1, exchange	3 1* 4* 6 7
c.	4 is compared with 6, no exchange	3 1 4 6 7
d.	6 is compared with 7, no exchange^	3 1 4 6 7
Third Pass		
a.	3 is compared with 1, exchange	1* 3* 4 6 7
b.	3 is compared with 4, no exchange	1 3 4 6 7
c.	4 is compared with 6, no exchange^	1 3 4 6 7
d.	6 is compared with 7, no exchange^	1 3 4 6 7
Fourth Pass		
a.	1 is compared with 3, no exchange	1 3 4 6 7
b.	3 is compared with 4, no exchange^	1 3 4 6 7
c.	4 is compared with 6, no exchange^	1 3 4 6 7
d.	6 is compared with 7, no exchange^	1 3 4 6 7

^ An efficient form of the bubble sort omits these steps because the items they compare

Example 1 shows a trace of the bubbling process through a list of five items. This process makes four passes through a nested loop to bubble the largest item down to the end of the list. Once again, the items just swapped are marked with asterisks.

2.1 Bubble Sort algorithms

2.1.1 Version 1. In the algorithm:

`Arr` = array to be sorted, index of first array element is **0**
`number_of_elements` = number of elements in the array
`temp` = temporary area for holding an array element which is being switched
`I` = index for outer loop
`J` = index for inner loop

Assume that the contents of Array and `number_of_elements` have already been established.

```
Bubble_Sort [Version 1]
   Set I to number_of_elements
   DOWHILE ( I > 1 )          // Minimum 2 elements. Do I - 1 passes
     Set J to 1               // Start from 2nd element
     DOWHILE ( J < I )        //Do I - 1 times
        IF Arr[J] < Arr[J - 1] THEN   //Compare adjacent elements
           temp      ← Arr[J]         //Swap
           Arr[J]    ← Arr[J - 1]
           Arr[J - 1]← temp
        ENDIF
        J ← J + 1                     //Shift right one position
     ENDDO
     I ← I - 1                        //Next pass
   ENDDO
END
```

2.1.2 Version 2 Modified version: to keep track of the number of swaps it performs.

NoMoreSwaps = flag to record if the elements have been switched in the current pass

```
Bubble_Sort [Version 2] // With Efficient factor NoMoreSwaps
   Set I to number_of_elements
   Set NoMoreSwaps to TRUE
   DOWHILE ( I > 1 )          // Minimum 2 elements. Do I - 1 passes
     Set J to 1               // Start from 2nd element
     Set NoMoreSwaps to FALSE //Default: No swapping of any elements
     DOWHILE ( J < I )        //Do I - 1 times
        IF Arr[J] < Arr[J - 1] THEN   //Compare adjacent elements
           temp      ← Arr[J]         //Swap
           Arr[J]    ← Arr[J - 1]
           Arr[J - 1] ← temp
           NoMoreSwaps ← TRUE         //At least 1 swap in the pass
        ENDIF
        J ← J + 1                     //Shift right one position
     ENDDO
     IF NOT NoMoreSwaps THEN      //If NoMoreSwaps remain as FALSE
        RETURN                    //Sorting ended.
     ENDIF
     I ← I - 1                    //Next pass
   ENDDO
END
```

Example 2 Using Version 2 algorithm

The accompanying table shows a bubble sort to sort five numbers in ascending order.

Position of array elements		**0 1 2 3 4**
Original Order		**1 3 4 6 7**
		Result
	NoMoreSwaps	
	TRUE	
First Pass	**FALSE**	
a.	1 is compared with 3, no exchange	1 3 4 6 7
b.	3 is compared with 4, no exchange	1 3 4 6 7
c.	4 is compared with 6, no exchange	1 3 4 6 7
d.	6 is compared with 7, no exchange	1 3 4 6 7
	FALSE	

2.2 Complexity [Analysis] of Bubble Sort

2.2.1 Method 1. To calculate the complexity of the bubble sort algorithm, it is useful to determine **how many comparisons each loop performs**.

For each element in the array, bubble sort does $N - 1$ comparisons.
In big O notation, bubble sort performs $O(N)$ comparisons.

Because the array contains n elements, it has an $O(N)$ number of elements. In other words, bubble sort performs $O(N)$ operations on an $O(N)$ number of elements, leading to a total running time of $O(N^2)$.

2.2.2 Method 2. Another way to analyze the complexity of bubble sort is by determining the recurrence relation that represents it.

When `I = 2`, **one** comparison is made by the program. When `I = 3`, **two** comparisons are made, and so on. Thus, we can conclude that when `I = N`, **`N - 1`** comparisons are made. Hence, in an array of length `N`, it does

$$1 + 2 + 3 + 4 + \cdots + (N - 2) + (N - 1) \text{ comparisons.}$$
$$= N\ (N - 1)\ /\ 2$$
$$= \tfrac{1}{2}N^2 - \tfrac{1}{2}N$$

As expected, the algorithm's complexity is $O(N^2)$.

2.2.3 Best Case Time Complexity

If the numbers are already sorted in ascending order, in Version 2 the algorithm will determine in the first iteration that no number pairs need to be swapped and will then terminate immediately.

The algorithm must perform `N - 1` comparisons; therefore:

Conclusion: The best-case time complexity of Bubble Sort is: $O(N)$

2.2.4 Worse Case Time Complexity

Example 3 If the numbers are already in descending order, `Arr[6, 5, 4, 3, 2, 1]`

In the first iteration, there are 5 comparison and exchange operations.
In the second iteration, there are 4 comparison and exchange operations.

In the last iteration, there is 1 comparison and exchange operation.
`total = 5 + 4 + 3 + 2 + 1 = 15` comparison and exchange operations.

For `n` elements in an array, perform
`(N-1) + (N-1) + ... + 3 + 2 + 1` comparison and exchange operations.
`= N (N - 1) / 2`

Conclusion: The worst-case time complexity of Bubble Sort is: $O(N^2)$

2.2.5 Average Case Time Complexity

In the average case, roughly about half as many exchange operations as in the worst case since about half of the elements are in the correct position compared to the neighboring element. So the number of exchange operations is:

$$\frac{1}{4}\ (N^2 - N)$$

It becomes even more complicated with the number of comparison operations, which amounts to:

$\frac{1}{2}\ (N^2 - N \times ln(N) - (\gamma + ln(2) - 1) \times N) + O(\sqrt{N})$ **[not in syllabus]**

In both terms, the highest power of n is again N^2; therefore:

Conclusion: The average time complexity of Bubble Sort case is: $O(N^2)$

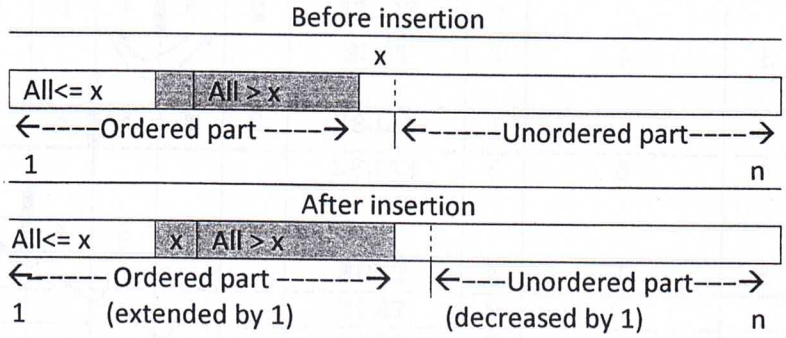
Bubble Sort Time Complexity		
Best case	Average case	Worse case
$O(N)$	$O(N^2)$	$O(N^2)$

The modified version of bubble sort performs better for lists that are already sorted. But the modified bubble sort can still perform poorly if many items are out of order in the list.

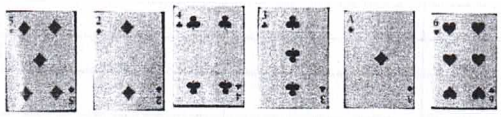
3 Insertion sort

The insertion sort, attempts to exploit the partial ordering of the list in a different way.

In the algorithm, the array is scanned until an out-of-order element is found. The scan is then temporarily halted while a backward scan is made to find the correct position to insert the out-of-order element. Elements bypassed during this backward scan are moved up one position to make room for the element being inserted.



Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.



The following algorithm sorts an integer array into ascending order using an insertion sort method.

In the algorithm:
`Arr` = array to be sorted, index of first element is **1**.
`N` = number of elements in the array
`itemToInsert` = temporary area for holding an array element while correct position is being searched
`I` = current position of the element
`J` = index for inner loop

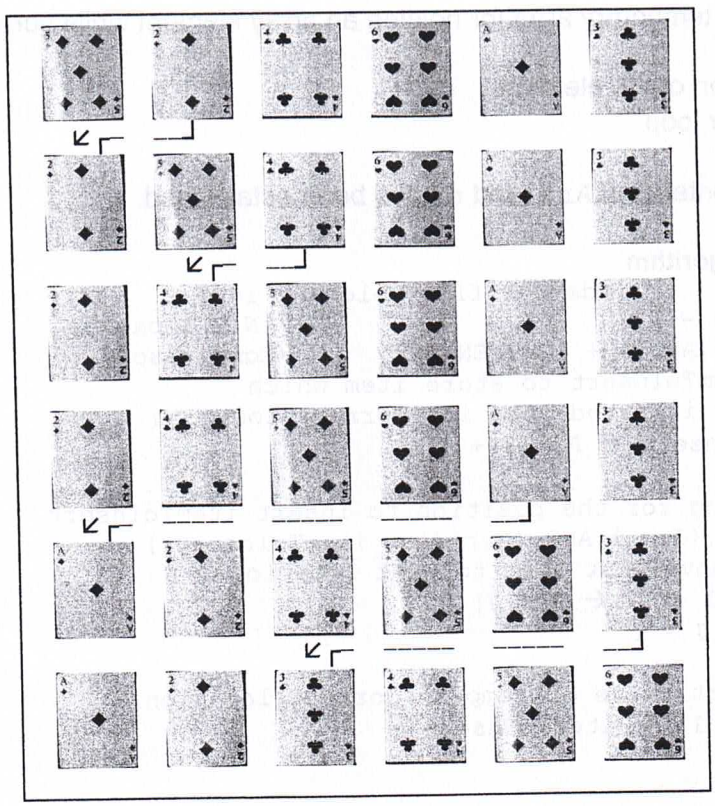
Assume that the contents of `Arr` and `n` have been established.

Insertion_Sort // algorithm

```
  Set I to 1        // index of first element is 1
  DOWHILE i ≤ (N - 1)                    // N - 1 passes
     IF Arr[I] > Arr[I + 1] THEN         //Comparison
     //use a itemToInsert to store item which
     // is to be inserted into its correct location
         itemToInsert ← Arr[I + 1]
         J ← I
         //looking for the position to insert itemToInsert
         DOWHILE (J ≥ 1 AND (Arr[J] > itemToInsert))
             // move list item to next location
            Arr[J + 1] ← Arr[J]
            J ← J - 1
         ENDDO
         // insert value of temp in correct location
         Arr[J + 1] ← itemToInsert
     ENDIF
     I ← I +1
  ENDDO
END
```

Example 4 Tracing with Insertion Sort

N	N-1	I+1	itemToInsert	J	J ≥ 1 AND Arr[J] > itemToInsert	[1]	[2]	[3]	[4]	[5]	[6]
6	5										
Arr	I = 1	2	2	1	TRUE	5	2	4	6	1	3
				0	FALSE	**2**	5	4	6	1	3
	I = 2	3	4	2	TRUE						
				1	FALSE	2	**4**	5	6	1	3
	I = 3	4	6	3	FALSE						
						2	4	5	6	1	3
	I = 4	5	1	4	TRUE						
				3	TRUE						
				2	TRUE						
				1	TRUE						
				0	FALSE						
						1	2	4	5	6	3
	I = 5	6	3	5							
				4							
				3							
SORTED						1	2	**3**	4	5	6



3.1 Analysis of Insertion Sort

Once again, analysis focuses on the nested loop. The outer loop executes $N - 1$ times. In the worst case, when all the data are out of order, the inner loop iterates once on the first pass through the outer loop, twice on the second pass, and so on, for a total of $\frac{1}{2} N^2 - \frac{1}{2} N$ times. Thus, the worst-case behavior of insertion sort is $O(N^2)$. The more items in the list that are in order, the better insertion sort gets until, in the best case of a sorted list, the sort's behavior is linear, $O(N)$. In the average case, however, insertion sort is still quadratic.

3.2 Insertion Sort using recursion

Base Case: If array size is 1 or smaller, return.

- Recursively sort first N - 1 elements.
- Insert last element at its correct position in sorted array.

Sort array `Arr[1 . . N]`. Assumes `N` ≤ length of **`Arr`**

```
Insertion_Sort(Arr, N)              // With_Recursion
1     IF N > 1 THEN
2           Insertion_Sort(Arr, N - 1)
3           Key ← Arr[N]
4           I ← N - 1
5           DOWHILE I > 0 and Arr[I] > Key
6                 Arr[I+1] ← Arr[I]
7                 I ← I - 1
8           ENDDO
9           Arr[I +1] ← Key
10    ELSE // N = 1
11          RETURN
12    ENDIF
END
```

	Calling `Insertion_Sort ()` //with recursion	N	N-1	Key = Arr[N]	I	Arr [1]	[2]	[3]	[4]	[5]	[6]
						5	2	4	6	1	3
1st	recursive_insertion_sort([5,2,4,6,1,3], 6)	6	5								
2nd	| recursive_insertion_sort([5,2,4,6,1,3], 5)	5	4								
3rd	| | recursive_insertion_sort([5,2,4,6,1,3], 4)	4	3								
4th	| | | recursive_insertion_sort([5,2,4,6,1,3], 3)	3	2								
5th	| | | | recursive_insertion_sort([5,2,4,6,1,3], 2)	2	1								
6th	| | | | | recursive_insertion_sort([5,2,4,6,1,3], 1)	1									
5th	Return recursive_insertion_sort([5,2,4,6,1,3], 2) | | | | puts 2 in the right position between it's ORDERED left values [5] → [2,5]	2		2	1	2	5				
					0			4			
4th	Return recursive_insertion_sort([2,5,4,6,1,3], 3) | | | puts 4 in the right position between it's ORDERED left values [2,5] → [2,4,5]	3		4	2			5			
					1	2	4	5	6		
3rd	Return recursive_insertion_sort([2,4,5,6,1,3], 4) | | puts 6 in the right position between it's ORDERED left values [2,4,5] → [2,4,5,6]	4		6	3	2	4	5	6	1	
2nd	Return recursive_insertion_sort([1,2,4,5,6,3], 5) | puts 1 in the right position between it's ORDERED left values [2,4,5,6] → [1,2,4,5,6]	5		1	4					6	
					3				5		
					2			4			
					1		2				
					0	1					3
1st	Return recursive_insertion_sort([1,2,3,4,5,6], 6) puts 3 in the right position between it's ORDERED left values [1,2,4,5,6] → [1,2,3,4,5,6]	6		3	5						6
					4					5	
					3				4		
	0, 1, 2, 3, 4, 6, 8, 9 best -8 nos 7 comparisons n-1 O(n)					1	2	3			
	4, 6, 1, 8, 9, 2, 0, 3 random average										
	9, 8, 6, 4, 4, 2, 1, 0 worst										

3.3 Analysis of Insertion Sort with recursion

Calling the same function recursively for `(N - 1)` elements and iterating for all the elements less than the current index in each call, so Time complexity is $O(N * N)$.

However, insertion sort saves programming effort as it is easy to write and maintain. It works reasonably well for small list and best when the list is already partially sorted. In fact this is the best sorting method if the list is already in order, in this case it do only `N - 1` comparison and nothing else.

In the **best case**, each of the **N** items is in its proper place and thus Insertion Sort takes linear time, or $\mathbf{O(N)}$.

In the **average** and **worst case**, each of the N items must be transposed a linear number of positions, thus Insertion Sort requires $\mathbf{O(N^2)}$ quadratic time.

Insertion Sort Time Complexity			
Best case	Average case	Worse case	Concept
$O(N)$	$O(N^2)$	$O(N^2)$	*Array*

Tutorial 14A [Bubble and Insertion Sorts]

1 Describe how bubble sort and insertion sort algorithms operate. [4]

2 Explain the role that the number of data exchanges plays in the analysis of bubble sort. What role, if any, does the size of the data objects play?

3 Explain why the modified bubble sort still exhibits $O(N^2)$ behavior on the average.

4 Explain why insertion sort works well on partially sorted lists.