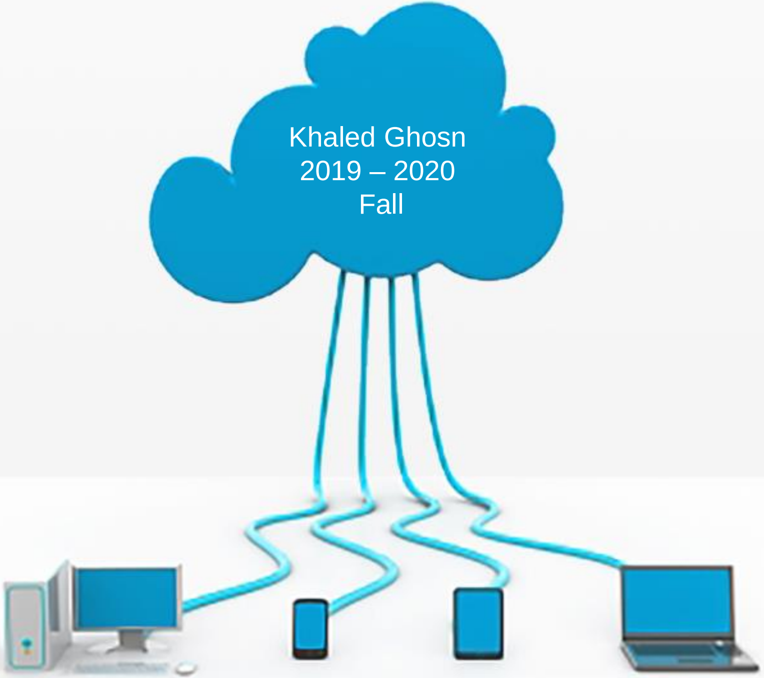
Sorting Algorithms

Selection, Insertion, Quick, Merge



Khaled Ghosn
2019 – 2020
Fall

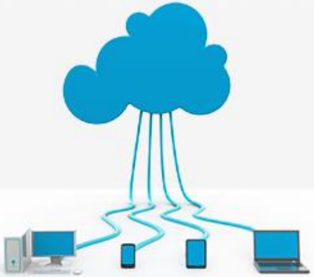
Sorting Algorithms

Comparison

O **(n^2)** in average case:

- Bubble Sort
- Selection Sort
- Insertion Sort

- Insertion sort might be the best for all lists under 5 elements
- In practice, merge sort is faster for lists as small as 50 elements, merge sort needs additional space



Selection Sort

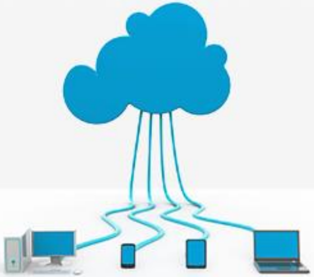
Specification

Idea:

- Find the smallest element in the array
- Exchange it with the element in the first position
- Find the second smallest element and exchange it with the element in the second position
- Continue until the array is sorted

Disadvantage:

- Running time depends only slightly on the amount of order in the file

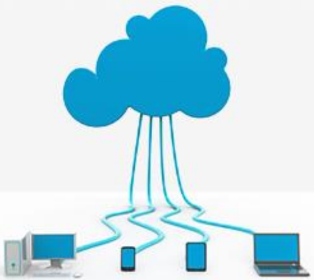


Selection Sort

Example

Starting from smallest to biggest

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8
1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9



Selection Sort

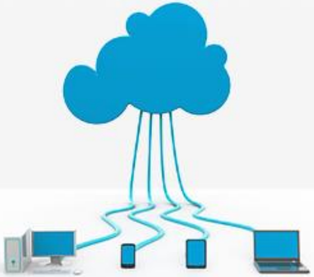
Algorithm Analysis

n ← length[A]
for i ← 1 to n - 1
    do smallest ← i

    for index ← i + 1 to n
        do if A[index] < A[smallest]
            then smallest ← index

    exchange A[i] ↔ A[smallest]

8	4	6	9	2	3	1



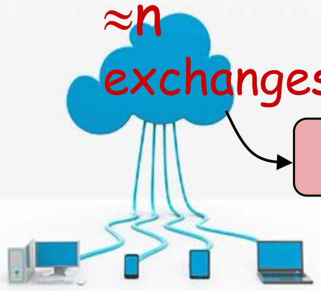
Selection Sort

Algorithm Analysis

Algorithm		cost	times
$n \leftarrow$ length[A]		c_1	1
for $i \leftarrow 1$ **to** $n - 1$		c_2	n
do smallest $\leftarrow i$		c_3	$n-1$
for index $\leftarrow i + 1$ **to** n	$\sum_{j=1}^{n-1}(n-j+1)$	c_4	
do if $A[\text{index}] < A[\text{smallest}]$	$\sum_{j=1}^{n-1}(n-j)$	c_5	
then smallest $\leftarrow$ index	$\sum_{j=1}^{n-1}(n-j)$	c_6	
exchange $A[i] \leftrightarrow A[\text{smallest}]$		c_7	$n-1$

$\approx n^2/2$ comparisons

$\approx n$ exchanges

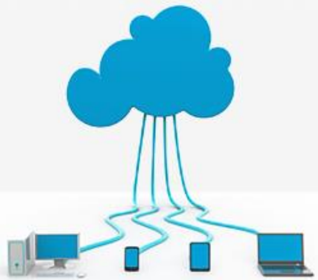


$$T(n) = c_1 + c_2 n + c_3(n-1) + c_4 \sum_{j=1}^{n-1}(n-j+1) + c_5 \sum_{j=1}^{n-1}(n-j) + c_6 \sum_{j=2}^{n-1}(n-j) + c_7(n-1) = \Theta(n^2)$$

Selection Sort

Implementation

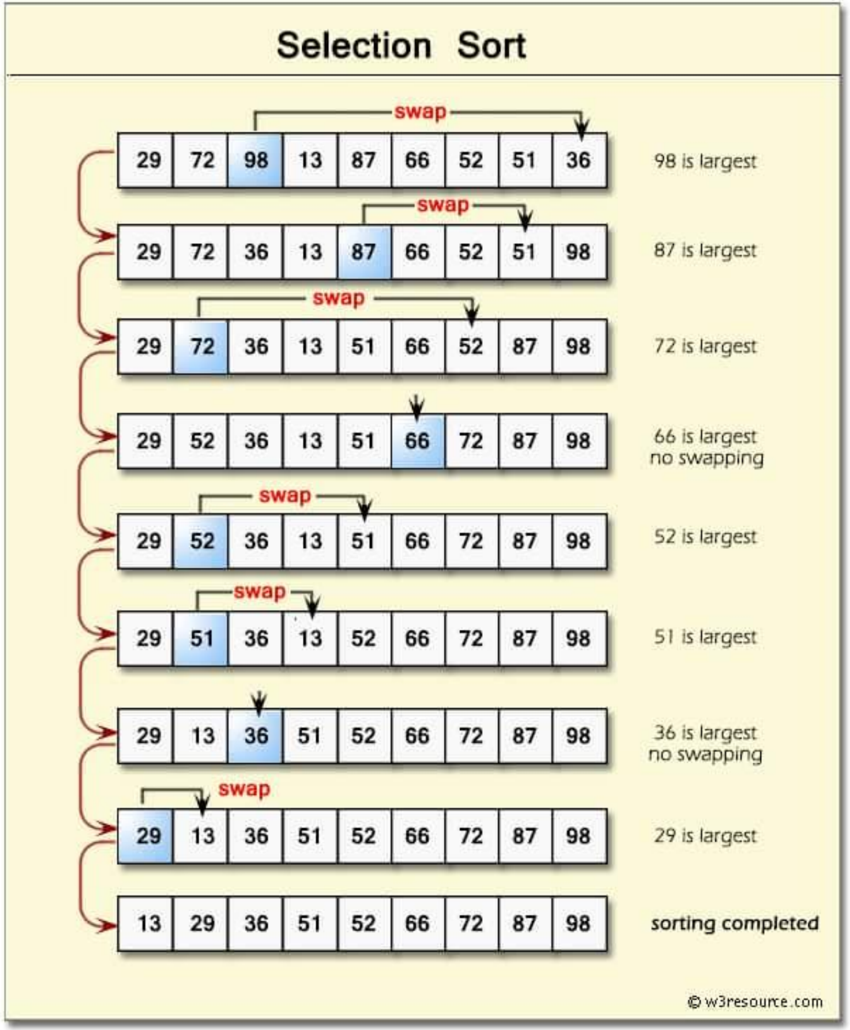
```
public void SelectionSort(int [] data)
{
    int smallest; // index of smallest element

    // loop over data.length - 1 elements
    for ( int i = 0 ; i < data.length - 1 ; i++ )
    {
      smallest = i; // first index of remaining array

      // loop to find index of smallest element
      for ( int index = i + 1 ; index < data.length; index++ )
        if ( data[ index ] < data[ smallest ] )
          smallest = index;

      // swap smallest element into position    Swap(i, smallest)
      int temporary = data[ i ]; // store first in temporary
      data[ i ] = data[ smallest ]; // replace first with second
      data[ smallest ] = temporary; // put temporary in second
    } // end outer for
 } // end method sort
```

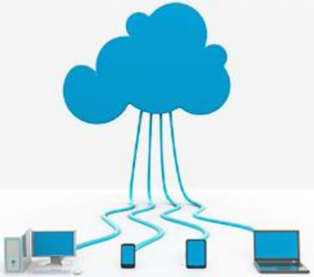


Selection Sort

Example

Starting from biggest to smallest

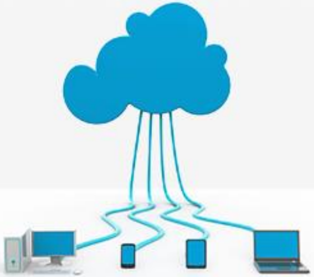




Insertion Sort

Specification

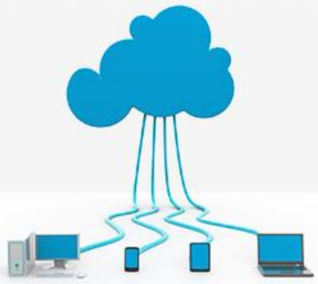
- More efficient than bubble sort, because in insertion sort the comparisons of elements are less as compare to bubble sort.
- In insertion sorting algorithm compare the value until all the prior elements are lesser than compared value is not found.
- This mean that the all previous values are lesser than compared value.
- Insertion sort is a good choice for small values and for nearly-sorted values.
- There are more efficient algorithms such as quick sort, heap sort, or merge sort for large values.



Insertion Sort

Implementation

```
public void InsertionSort (int [] data)
    {
        int insert; // temporary variable to hold element to insert

        // loop over data.length - 1 elements
        for ( int next = 1; next < data.length; next++ )
        {
            // store value in current element
            insert = data[ next ];

            // initialize location to place element
            int moveItem = next;

            // search for place to put current element
            while ( moveItem > 0 && data[ moveItem - 1 ] > insert )
            {
                // shift element right one slot
                data[ moveItem ] = data[ moveItem - 1 ];
                moveItem--;
            } // end while

            data[ moveItem ] = insert; // place inserted element
        } // end for
} // end method sort
```



Quick-Sort

Randomized Quick-Sort

- we desire some way of getting close to the best-case running time for quick-sort
- The way to get close to the best-case running time: the pivot to divide the input sequence S almost equally
- If this outcome were to occur, then it would result in a running time that is asymptotically the same as the best-case running time
- That is, having pivots close to the “middle” of the set of elements leads to an $O(n \log n)$ running time for quick-sort

Quick-Sort

Randomized Quick-Sort

Picking Pivots at Random:

- pick as the pivot a ***random element*** of the input sequence
- instead of picking the pivot as the first or last element of S, we pick an element of S at random as the pivot, keeping the rest of the algorithm unchanged
- this variation of quick-sort is called ***randomized quick-sort***
- the expected running time is $O(n \log n)$

Quick-Sort

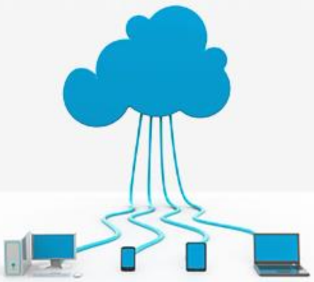
Quicksort for Small Arrays

- For very small arrays ($N<= 20$), quicksort does not perform as well as insertion sort
- A good cutoff range is $N=10$
- Switching to insertion sort for small arrays can save about 15% in the running time

Quick-Sort

Algorithm

1. Get the pivot element from the middle of the list
2. Divide into two lists:
   a) If the current value from the left list is smaller then the pivot element then get the next element from the left list
   b) If the current value from the right list is larger then the pivot element then get the next element from the right list
   c) If we have found a values in the left list which is larger then the pivot element and if we have found a value in the right list which is smaller then the pivot element then we **exchange** the values
      - Increase left pointer and decrease right pointer
3. Recursively sort two sub parts



Merge Sort

Quick sort faster than Merge sort

Both quicksort and merge-sort take O(N log N) in the average case.

Why is quicksort faster than merge-sort?

- The inner loop consists of an increment/decrement (by 1, which is fast), a test and a jump.
- There is no extra juggling as in merge-sort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

inner loop

Merge Sort

Comparisons of Merge-sort and Quicksort

- Both run in ***O* (n log n)**
- Compared with Quicksort, Merge-sort has less number of comparisons but larger number of moving elements
- In Java, an element comparison is expensive but moving elements is cheap. Therefore, Merge-sort is used in the standard Java library for generic sorting

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)