Sorting

- What is sorting?
  - An algorithm that segregates order items according to specified criterion: Ascending or descending
    - Items: 3 1 6 2 1 3 4 5 9 0
    - Ascending: Items: 0 1 1 2 3 3 4 5 6 9
    - Descending: Items: 9 6 5 4 3 3 3 2 1 1 0

- Sorting Algorithms:
  - There are many, many different types of sorting algorithms, but the primary ones are:
    - Bubble Sort
    - Selection Sort
    - Insertion Sort
    - Merge Sort
    - Shell Sort
    - Heap Sort
    - Quick Sort
    - Radix Sort

- Selection Sort
  - How does it work?
    - Start with the 1st element, scan the entire list to find its smallest element and exchange it with the 1st element
    - Start with the 2nd element, scan the remaining list to find the the smallest among the last (N-1) elements and exchange it with the 2nd element
    - …
  - Example:

- Given the following list: **100, 2, 90, 101, 1, 0, 78**

Find min Value:	100, 2, 90, 101, 1, **0,** 78
Swap:	**0**|, 2, 90, 101, 1, **100**, 78
Find next min Value:	**0**|, 2, 90, 101, **1**, 100, 78
Swap:	**0**, **1**|, 90, 101, **2**, 100, 78
Find next min Value:	**0**, **1**|, 90, 101, **2**, 100, 78
Swap:	**0**, **1**, **2**|, 101, **90**, 100, 78
Find next min Value:	**0**, **1**, **2**|, 101, 90, 100, **78**
Swap:	**0**, **1**, **2**, **78**|, 90, 100, **101**
Find next min Value:	**0**, **1**, **2**, **78**|, **90**, 100, 101
Swap:	**0**, **1**, **2**, **78**|, **90**, 100, 101
Find next min Value:	**0**, **1**, **2**, **78**, **90**|, **100**, 101
Swap:	**0**, **1**, **2**, **78**, **90**|, **100**, 101
Find next min Value:	**0**, **1**, **2**, **78**, **90**, **100**|, **101**
Swap:	**0**, **1**, **2**, **78**, **90**, **100**, **101**|

- Algorithm:

```
SelectionSort (int A[n]) {
int i, j;
for i = 0 to N-2 {                          ∑_{i=0}^{N-2}
        min = i;
        for j = i+1 to N-1 {                ∑_{j=i+1}^{N-1} c
            if (A[j] < A[min])
                min = j;
        }
        swap A[i] and A[min] ;
}
```

- Analysis:

$$T(N) = \sum_{i=0}^{N-2} \sum_{j=i+1}^{N-1} c$$

$$= \sum_{i=0}^{N-2} c(N-1-i)$$

$$= \sum_{i=0}^{N-2} cN-1-i) = \sum_{i=0}^{N-2} cN - \sum_{i=0}^{N-2} c - \sum_{i=0}^{N-2} ci$$

$$= cN(N-2+1)-c(N-2+1)-c\frac{(N-2)(N-1)}{2}$$

$$= cN^2-2cN+cN-cN+2cN-c-\frac{1}{2}cN^2-\frac{3}{2}cN-c$$

$$= \frac{1}{2}cN^2 - \frac{3}{2}cN-2c$$

$$T(N) = O(N^2)$$

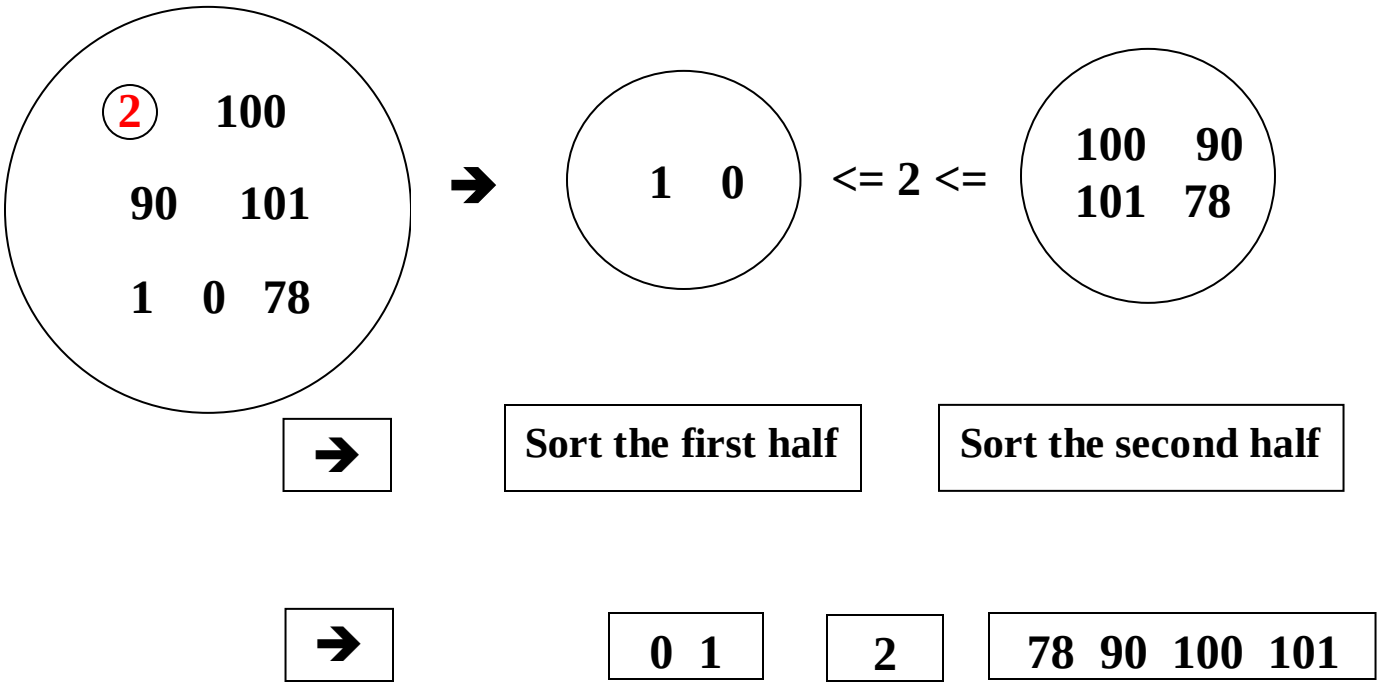
- Bubble Sort
  - Bubble scan the array from left to right, comparing a pair of adjacent values.
  - Bubble the largest element:
    - Any time it finds a larger element before a smaller element, it swaps the two, moving the largest element to the end of the array.
  - Then it repeats the process for the unsorted portion of the array until the whole array is sorted.
  - Example:
    - Given the following items:

- First Pass: **100, 2,** 90, 101, 1, 0, 78 Swap: 2, **100, 90**, 101, 1, 0, 78 Swap: 2, 90, **100, 101**, 1, 0, 78 Swap: 2, 90, 100, **101, 1**, 0, 78 Swap: 2, 90, 100, 1, **101, 0**, 78 Swap: 2, 90, 100, 1, 0, **101, 78** Swap: 2, 90, 100, 1, 0, 78, **101**	- Second Pass: **2, 90**, 100, 1, 0, 78, **101** 2, **90, 100**, 1, 0, 78, **101** 2, 90, **100, 1**, 0, 78, **101** Swap: 2, 90, 1, **100, 0**, 78, **101** Swap: 2, 90, 1, 0, **100, 78**, **101** Swap: 2, 90, 1, 0, 78, **100**, **101**
- Third Pass: **2, 90**, 1, 0, 78, **100**, **101** 2, **90, 1**, 0, 78, **100**, **101** Swap: 2, 1, **90, 0**, 78, **100**, **101** Swap: 2, 1, 0, **90, 78**, **100**, **101** Swap: 2, 1, 0, 78, **90**, **100**, **101**	- Fourth Pass: **2, 1**, 0, 78, **90**, **100**, **101** Swap:1, **2, 0**, 78, **90**, **100**, **101** Swap:1, 0, **2, 78**, **90**, **100**, **101** 1, 0, 2, **78**, **90**, **100**, **101**
- Fifth Pass: **1, 0**, 2, **78**, **90**, **100**, **101** Swap: 0, **1, 2**, **78**, **90**, **100**, **101** 0, 1, **2, 78**, **90**, **100**, **101** 0, 1, **2, 78**, **90**, **100**, **101**	- Sixth Pass: **0, 1, 2, 78**, **90**, **100**, **101** 0, **1, 2, 78**, **90**, **100**, **101**

  - Algorithm

```
BubbleSort (int A[n]) {
int i, j;
for(i = 0; i < n; i++)
        for(j = 0; j < n-1; j++)
                if(A[j] > A[j+1] )
                        Swap A[j] and A[j+1];
}
```

- **Quicksort**
  - **Approach**:
    - **Uses Divide and Conquer strategy**:
      - If the input is small, solve directly and exit.
      - Split the input into 2or more parts.
      - Solve the problem for those smaller.
      - Combine the solutions.
  - **In Quicksort:**
    - Partition the input set into **<u>two sets</u>** using a chosen value called **Pivot**



  - **Advantages**:
    - Sorts in place
    - Average time complexity: O(n lg n)
    - Very efficient in practice

  - **Disadvantage**:
    - Worst time complexity: $O(n^2)$
    - When data is already sorted

- Divide and Conquer Algorithm:
  - **Divide:** the array $A_{[}p..r_{]}$ into two parts:
  - $A_{[}p..q_{]}$ and $A_{[}q+1..r_{]}$ such that each element in $A_{[}p..q_{]}$ is less than or equal to each element in $A_{[}q+1..r_{]}$ where q is found by the **partitioning function**
  - **Conquer**:$A_{[}p..q_{]}$ and $A_{[}q+1..r_{]}$ are sorted recursively by calling Quicksort procedure.
  - **Combine**: No work needed since the subarrays are sorted in place already.

- **Partitioning function**:
  - **Example**:

100, 2, 90, 101, 1, 0, 78, 300, 200

100	2	90	101	1	0	78	300	200
100	2	90	101	1	0	78	300	200
100	2	90	101	1	0	78	300	200
100	2	78	101	1	0	90	300	200
100	2	78	90	1	0	101	300	200
100	2	78	90	1	0	101	300	200
100	2	78	90	1	0	101	300	200
100	2	78	90	1	0	101	300	200

0	2	78	90	1	100	101	300	200

- int function(type A[1..n], int p,r);
  int i,j; type x;
  begin
  x = A[p]; i=p; j=r+1;
  while TRUE do
  repeat
  j = j-1;
  until (A[j]≤x);
  repeat
  i = i+1;
  until (A[i]≥x);
  if (i<j) then
  swap(A[i],A[j]);
  else
  A[p] = A[j];
  A[j] = x;
  return(j);
  endif;
  Endwhile
  end;

- **Complexity**:
  - partitioning takes O(n) where n is the number of elements to be sorted.

- **Quicksort procedure:**

```
procedure Quicksort(type A[1..n], int p,r);
int q;
Begin
      if (p<q)
      then begin
                  q = partition(A[1..n], p,r);
                  Quicksort(A[1..n], p,q);
                  Quicksort(A[1..n], q+1,r);
            end;
      endif;
end;
```

  - Complexity:
    - The recurrence relation of Quicksort is:

$$T(n) = \begin{cases} 1 & n = 1 \\ 2 * T(\frac{n}{2}) + n & n > 1 \end{cases}$$

===> T(n) = O(nlogn)

  - What is the time complexity when the **array is already sorted**?

- **MergeSort**
    - **Motivation**:
        - Worst case time complexity is O(nlogn)
    - **Merging**:
        - Two sorted lists:

1	5	8	9	15	18	20	30

3	7	11	17	19	25	35	40

1	3	5	7	8	9	11	15	17	18	19	20	25	30	35	40

        - **Complexity**:
            - The time complexity to merge two sorted lists of m and n elements is:

O(max(n,m)

    - **Algorithm**:

Procedure mergesort(int L,H);
/* A[L..H] is a global array containing H-L+1 ≥ 0 */

```
/* values which represent the elements to be sorted */
int MID;
Begin
      If (L<H) then
            begin
                  MID = (L+H)/2;
                  mergesort(L,MID);
                  mergesort(MID+1,H);
                  merge(L,MID,H);
            end;
      endif;
End;
```

- Complexity:
  - The recurrence relation of mergesort is:

$$\mathbf{T(n)} = \begin{cases} \mathbf{1} & \mathbf{n = 1} \\ \mathbf{2 * T(\frac{n}{2}) + n} & \mathbf{n > 1} \end{cases}$$

===> T(n) = O(nlogn)

- **Stable Sorting**:
  - A "Stable" sorting algorithm leaves the order of equal elements unchanged.
  - Examples:
    - Selection sort is not stable
    - Bubble sort is stable
    - Quicksort is not stable
    - Mergesort is stable
    - Heapsort is not stable

- **Comparing Sorting Algorithms:**

	Worst Case	Average Case
Selection Sort	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Mergesort	O(nlogn)	O(nlogn)
Quicksort	$O(n^2)$	O(nlogn)
Heapsort	O(nlogn)	O(nlogn)

- Code Example

```
public class Sort {

   public static int partition(int arr[], int left, int right)
   {
   }

   public static void quickSort(int arr[], int left, int right) {
         int index = partition(arr, left, right);
         if (left < index - 1)
               quickSort(arr, left, index - 1);
         if (index < right)
               quickSort(arr, index, right);
   }

   public static void bubbleSort(int a[], int n)
   {
   }

   public static void selectionSort(int a[], int n)
   {
   }


}
```

- **Programming Assignment:**
  - Design and implement the missing operations in the sorting class above:
    - partition function for quicksort
    - Bubble sort
    - Selection sort
  - Measure the time for each of your sorting algorithm.
  - Use an array size of 1000 randomly generated integers.
  - Test your implementation.