# Introduction to Sorting

- References:

    - Text book : Chapter 8, Skip Chapter 9: Faster Sorting Methods
    - Previous CSC313 class notes

- Java interface Comparable:

    Define a method compareTo() return integer // recall String class

    Example: x.compareTo(y) returns

    negative (x<y) , 0 (x=y) or positive (x>y)

- Generic type static method

    Recall generic class definition: class A < T > {….}

    If you hava a static method, this is not working:

```
class A <T> {
        static public void f (T a) {….}  // not working!
}
```

    Generic type static method should be:

```
class A {
        static public <T> void f (T a) {….}
}
```

    May call this method:

```
A.f("string");
```

*Bounded Generic Types*: If you want only generic types which has implemented certain interface, such as interface Comparable (which contains compareTo() method), you can use bounded type parameter, syntax is :

```
class A {
        static public <T extends Comparable <T> > void f (T a)
        {….}
 }
```

This means compareTo() only work for T, but not subclasses or super classes of T. For example:

```
class Num implements Comparable<Num>
{
        … // implement compareTo() ….
}

class Num2 extends Num
{
        // do not implement compareTo(), inherit compareTo()….
}
```

Then,

```
Num myNumObj;
        …
A.f(MyNumObj); // it is OK

Num2 myNum2Obj;
        ..
A.f(MyNum2Obj); // it is error, no compareTo()
```

*Wildcards:* To ensure compareTo() work for type T (as long as it or its superclass implements compareTo), use "? Super" wild card as follows:

```
        <T extends Comparable<? Super T>>

        // now,  both  A.f(MyNum2Obj) and A.f(MyNumObj); are OK
        // using compareTo() in Num class
```

- Java static methods that sort an array of "Comparable" objects

  To ensure that generic class T (or its superclasses) must implement Comparable, use

```
        public class SortArray
        {
                public static <T extends Comparable<? Super T>>
                                                        void sort(T[] a, int n)
                {…}
                …
        }
```

## Selection sort

Idea: For each loop i (from 0 to n-1),
search the smallest remaining object (between i to n)
place it in proper location, i.e. i

- Example :

| Index | Initial | 1st loop | 2nd loop | 3rd loop | 4th loop |
|---|---|---|---|---|---|
| 0 | 13 | *-10* | *-10* | *-10* | *-10* |
| 1 | 1 | 1 | *-5* | *-5* | *-5* |
| 2 | 2 | 2 | 2 | *1* | *1* |
| 3 | -5 | -5 | 1 | 2 | *2* |
| 4 | -10 | 13 | 13 | 13 | 13 |

## IterativeAlgorithm:

```
for (index = 0 ; index < n - 1 ; index++)
{
        Find indexOfNextSmalles,
        i.e. the index of the smallest value among a[index], . . . , a [n - 1]

        Interchange the values of a[index] and a[indexOfNextSmallest]
}
```

- Java program:

```
/*******************************************************
 * Class for sorting an array of Comparable objects from smallest to
 * largest.
 *******************************************************/

public class SortArray
{
  /** Task: Sorts the first n objects in an array into ascending order.
   *  @param a  an array of Comparable objects
   *  @param n  an integer > 0 */
  public static <T extends Comparable<? super T>>
                 void selectionSort(T[] a, int n)
  {
          for (int index = 0; index < n - 1; index++)
          {
            int indexOfNextSmallest = getIndexOfSmallest(a, index, n - 1);
            swap(a, index, indexOfNextSmallest);
            // Assertion: a[0] <= a[1] <= . . . <= a[index] <= all other a[i]
          } // end for
  } // end selectionSort
```

```
/** Task: Finds the index of the smallest value in a portion of an
 *      array.
 *  @param a      an array of Comparable objects
 *  @param first  an integer >= 0 and < a.length that is the index of
 *              the first array element to consider
 *  @param last   an integer >= first and < a.length that is the index
 *              of the last array element to consider
 *  @return the index of the smallest value among
 *        a[first], a[first + 1], . . . , a[last] */
private static <T extends Comparable<? super T>>
      int getIndexOfSmallest(T[] a, int first, int last)
{
     T min = a[first];
     int indexOfMin = first;
     for (int index = first + 1; index <= last; index++)
     {
       if (a[index].compareTo(min) < 0)
       {
         min = a[index];
         indexOfMin = index;
         // Assertion: min is the smallest of a[first] through a[index].
       } // end if
     } // end for
     return indexOfMin;
} // end getIndexOfSmallest
```

```
/** Task: Swaps the array elements a[i] and a[j].
  *  @param a  an array of objects
  *  @param i  an integer >= 0 and < a.length
  *  @param j  an integer >= 0 and < a.length */
 private static void swap(Object[] a, int i, int j)
 {
          Object temp = a[i];
          a[i] = a[j];
          a[j] = temp;
 } // end swap
} // end SortArray
```

- The efficiency of Selection sort

  main for loop executes n – 1 times

  For loop i, inner loop executes, getIndexOdSmallest(), n – i - 1 times

  $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 = \mathbf{O(n^2)}$

- **Insertion sort**

  Idea: For each loop i (from 1 to n-1),
  Objects from 0 to i-1 already sorted
  Find proper location within 0..i to insert object i

- Example :

| Index | Initial | 1st loop | 2nd loop | 3rd loop | 4th loop |
|---|---|---|---|---|---|
| 0 | *13* | *1* | *1* | *-5* | *-10* |
| 1 | 1 | *13* | *2* | *1* | *-5* |
| 2 | 2 | 2 | *13* | *2* | *1* |
| 3 | -5 | -5 | -5 | *13* | *2* |
| 4 | -10 | -10 | -10 | -10 | *13* |

```
Algorithm insertionSort (a, first, last)
   // Sorts the array elements a[first] through a[last] iteratively.

    for (unsorted = first + 1 through last)
    {
           firstUnsorted = a [unsorted]
           insertInOrder (firstUnsorted, a, first, unsorted - 1)
    }

Algorithm insertInOrder (element, a, begin, end)
   // Inserts element into the sorted array elements a[begin] through a[end].

    index = end
    while ((index >= begin) and (element < a [index]))
    {
       a [index + 1] = a [index]  // make room
       index--
    }
    // Assertion: a[index + 1] is available.
    a [index + 1] = element // insert
```

- The efficiency of insertion sort

  main for loop executes n – 1 times

  For loop i, inner loop executes, insertInOrder(),
  At most i times

  $1 + 2 + \ldots + (n - 2) + (n - 1) = n(n - 1)/2 = \mathbf{O(n^2)}$

| | | |
|---|---|---|
| Best case running time : | $O(n)$ | // input a sorted list |
| Average case running time: | $O(n^2)$ | |
| Worst Case Running time : | $O(n^2)$ | |

- **Bubble sort**

  Algorithm

```
// compare adjacent keys from the remaining keys in A[0, n-j-1]
// and exchange them if they are out of order
// one record, A[n-pass], is in place in every pass

sorted = false
for (pass=1; (pass < n) && !sorted; pass++)
{
    sorted = true
    for (k=0; k<n-pass; k++)
            if (A[k].key > A[k+1].key)
            {
                    SWAP(A[k],A[k+1])
                    sorted = false
            }
}
```

- Example :

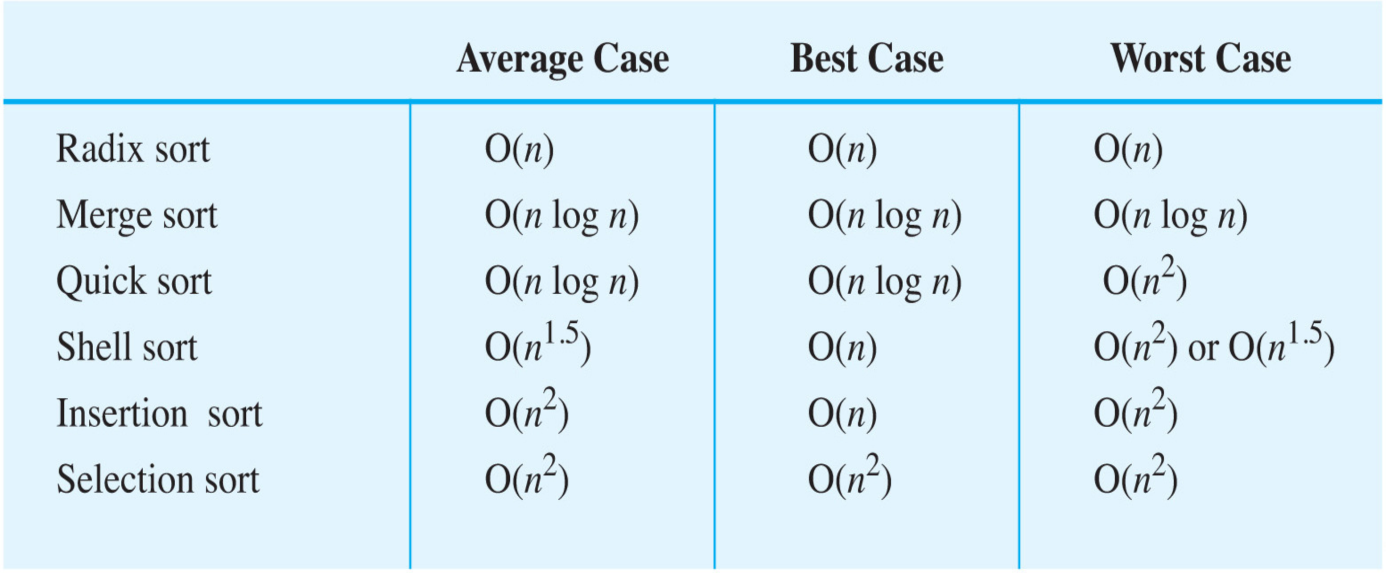
| Index | Initial | 1st loop | 2nd loop | 3rd loop | 4th loop |
|---|---|---|---|---|---|
| 0 | 13 | 1 | 1 | -5 | -10 |
| 1 | 1 | 2 | -5 | -10 | *-5* |
| 2 | 2 | -5 | -10 | *1* | *1* |
| 3 | -5 | -10 | *2* | *2* | *2* |
| 4 | -10 | *13* | *13* | *13* | *13* |

- Analysis

| | | |
|---|---|---|
| Best case running time : | O(n) | // input a sorted list |
| Average Case Running time : | O($n^2$) | // sort half of the list |

**We skip Chapter 9 Faster Sorting Methods in this course.**
**Those methods will be covered in CSC340 or CSC510**

**Faster methods include : Merge Sort, Quick Sort and Radix Sort**



| | Average Case | Best Case | Worst Case |
|---|---|---|---|
| Radix sort | O($n$) | O($n$) | O($n$) |
| Merge sort | O($n \log n$) | O($n \log n$) | O($n \log n$) |
| Quick sort | O($n \log n$) | O($n \log n$) | O($n^2$) |
| Shell sort | O($n^{1.5}$) | O($n$) | O($n^2$) or O($n^{1.5}$) |
| Insertion sort | O($n^2$) | O($n$) | O($n^2$) |
| Selection sort | O($n^2$) | O($n^2$) | O($n^2$) |