Sorting

Bubble Sort :fixed number of passes

```
public static void bubbleSort1(int[] x) {
    int n = x.length;
   for (int pass=1; pass < n; pass++) { // count how many times
  // This next loop becomes shorter and shorter
        for (int i=0; i < n-pass; i++) {
              if (x[i] > x[i+1]) {
                  // exchange elements
                     int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp;
                }
           }
    }
}
```

Bubble Sort -- stop when no exchanges

```
public static void bubbleSort2(int[] x) {
   boolean doMore = true;
    while (doMore) {
        doMore = false; // assume this is last pass over array
       for (int i=0; i<x.length-1; i++) {
         if (x[i] > x[i+1]) { // exchange elements
           int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp;
           doMore = true; // after an exchange, must look again
           }
       }
     }
}
```

Bubble Sort –
stop when no exchanges, shorter range each time

```
public static void bubbleSort3(int[] x) {
 int n = x.length;
  boolean doMore = true;
    while (doMore) {
          n--; doMore = false;
       // assume this is our last pass over the array
      for (int i=0; i<n; i++) {
        if (x[i] > x[i+1]) { // exchange elements
          int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp;
           doMore = true; // after an exchange, must look again
        } } } }
```

Bubble Sort -- Sort only necessary range

```
public static void bubbleSort4(int[] x) {
   int newLowest = 0; // index of first comparison
   int newHighest = x.length-1; // index of last comparison
   while (newLowest < newHighest) {
          int highest = newHighest;
           int lowest = newLowest;
           newLowest = x.length; // start higher than any legal index
         for (int i=lowest; i<highest; i++) {
              if (x[i] > x[i+1]) { // exchange elements
                int temp = x[i]; x[i] = x[i+1]; x[i+1] = temp;
              if (i<newLowest) {
                   newLowest = i-1;
                     if (newLowest < 0) { newLowest = 0; }
               } else if (i>newHighest) { newHighest = i+1; }
            } } } }
```

Simple selection sort

```
public static void selectionSort1(int[] x) {
   for (int i=0; i<x.length-1; i++) {
      for (int j=i+1; j<x.length; j++) {
            if (x[i] > x[j]) { //... Exchange elements
             int temp = x[i]; x[i] = x[j]; x[j] = temp;
             }
       }
     }
}
```

selection sort –(More efficient)
Move every value only once

```
public static void selectionSort2(int[] x) {
    for (int i=0; i<x.length-1; i++) {
     int minIndex = i; // Index of smallest remaining value.
         for (int j=i+1; j<x.length; j++) {
             if (x[minIndex] > x[j])
                 minIndex = j; // Remember index of new minimum
          }
       if (minIndex != i) {
         // Exchange current element with smallest remaining.
        int temp = x[i]; x[i] = x[minIndex];     x[minIndex] = temp;
       }
    }
}
```

Searching

- Binary Search – non-recursive
- Binary Search - recursive

Non-Recursive Binary search

```
public static int binarySearch(int[] sorted, int key) {
  int first = 0 , upto = Sorted.length ;
  while (first < upto) {
     int mid = (first + upto) / 2; // Compute mid point.
       if (key < sorted[mid])
            upto = mid; // repeat search in bottom half.
       else
         if (key > sorted[mid])     first = mid + 1; // Repeat search in top half.
          else      return mid; // Found it !  return position
       }
    return -1; // Failed to find key , -1 indicates the failure
}
```

Non-Recursive Binary Search search for string

```
public static int binarySearch(String[] sorted, String key) {
     int first = 0;
   int upto = sorted.length;
      while (first < upto) {
         int mid = (first + upto) / 2; // Compute mid point.
         if (key.compareTo(sorted[mid]) < 0) {
            upto = mid; // repeat search in bottom half. }
         else if (key.compareTo(sorted[mid]) > 0) {
                first = mid + 1; // Repeat search in top half.
         } else { return mid; // Found it. return position }
       }
     return -1; // Failed to find key
}
```

Non-Recursive Generic Binary search

```
public static int binarySearch(Comparable[] sorted, Comparable key)
{
   int first = 0 ,  upto = Sorted.length ;
    while (first < upto) {
    int mid = (first + upto) / 2; // Compute mid point.
      if (key .compareTo(sorted[mid])  < 0 )
           upto = mid; // repeat search in bottom half.
      else if (key .compareTo( sorted[mid])  > 0 )
                  first = mid + 1; // Repeat search in top half.
        else      return mid; // Found it !  return position
      }
   return -1; // Failed to find key , -1 indicates the failure
 }
```

Recursive Binary Search (Generic version)

```
public static int binarySearch(Comparable[] sorted, int first, int upto,
    Comparable key)
{
   if  (first > upto)  return -1 ;  // not found
    int mid = (first + upto) / 2; // Compute mid point.
     if (key.compareTo(sorted[mid]) < 0)
           upto = mid-1;
        else if (key.compareTo(sorted[mid]) > 0)
              first = mid + 1;
        } else { return mid; // Found it. return position }
      return binarySearch( Sorted, first, upto, key) ;
}
```