INTERNAL SORTING ALGORITHMS

```
// DSutil.java -- contains a bunch of utility functions.

import java.util.*;

public class DSutil {  // Swap two objects in an array

    public static void swap(Object[] array, int p1, int p2) {
            Object temp = array[p1];
                   array[p1] = array[p2];
                   array[p2] = temp;
    }

    public static Random value = new Random();// Random class object
    public static int random(int n) {
        return Math.abs(value.nextInt()) % n;
    }

    // Randomly permute the Objects in an array
    public static void permute(Object[] A) {
       for (int i = A.length; i > 0; i--)     // for each i
          swap(A, i-1, DSutil.random(i));     // swap A[i-1] with
    }                                         // a random element
} // end class DSutil

//Elem.java -- Elem interface.  This is just an Object with support for a key field.

interface Elem {              // Interface for generic element type
  public abstract int key(); // Key used for search and ordering
} // end interface Elem

// IElem.java -- Sample implementation for Elem interface.  A record with just an int field.

public class IElem implements Elem {

  private int value;

  public IElem(int v) { value = v; }
  public IElem() {value = 0;}
  public void setkey(int v) { value = v; }

  public String toString() { // Override Object.toString
    return Integer.toString(value);
  }

 public int key() { return value; }
}
```

```
//Sortmain.java
import java.io.*;
public class Sortmain {

/*
```

Insertion sort

`Idea:`

Let a_0, ..., $a_{n\text{-}1}$ be the sequence to be sorted. At the beginning and after each iteration of the algorithm the sequence consists of two parts: the first part a_0, ..., $a_{i\text{-}1}$ is already sorted, the second part a_i, ..., $a_{n\text{-}1}$ is still unsorted ($i \in$ 0, ..., n-1).

In order to insert element a_i into the sorted part, it is compared with $a_{i\text{-}1}$, $a_{i\text{-}2}$ etc. When an element a_j with $a_j \leq a_i$ is found, a_i is inserted behind it. If no such element is found, then a_i is inserted at the beginning of the sequence.

After inserting element a_i the length of the sorted part has increased by one. In the next iteration, a_{i+1} is inserted into the sorted part etc.

While at the beginning the sorted part consists of element a_0 only, at the end it consists of all elements a_0, ..., $a_{n\text{-}1}$.

Example: The following table shows the steps for sorting the sequence 5 7 0 3 4 2 6 1. On the left side the sorted part of the sequence is shown in red. For each iteration, the number of positions the inserted element has moved is shown in brackets. Altogether this amounts to 17 steps.

5	7	0	3	4	2	6	1	(0)
5	7	0	3	4	2	6	1	(0)
0	5	7	3	4	2	6	1	(2)
0	3	5	7	4	2	6	1	(2)
0	3	4	5	7	2	6	1	(2)
0	2	3	4	5	7	6	1	(4)
0	2	3	4	5	6	7	1	(1)
0	1	2	3	4	5	6	7	(6)

```
*/
```

```
static void insertionSort(Elem[] array) {    // Insertion Sort
  for (int i=1; i<array.length; i++)         // Insert i'th record
    for (int j=i; (j>0) && (array[j].key() <  array[j-1].key()); j--)
      DSutil.swap(array, j, j-1);
}
/*
```

Properties

- Stable
- O(1) extra space
- O(n^2) comparisons and swaps
- Adaptive: Θ(n) time when nearly sorted
- Very low overhead

Discussion

Although it is one of the elementary sorting algorithms with O(n^2) worst-case time, **insertion sort** is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (due to its low overhead).

For these reasons, and because it is also stable, insertion sort is often used as the base case (when the problem size is small) for more complex algorithms that have higher overhead such as **quicksort**.

*/

/*

Shell sort

// [D.L. Shell: A High-Speed Sorting Procedure. Communications of the ACM, 2, 7, 30-32 (1959)]

The idea of **Shell sort** is the following:

a. arrange the data sequence in a two-dimensional array
b. sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted. However, the number of sorting operations necessary in each step is limited, due to the presortedness of the sequence obtained in the preceding steps.

Example: Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

```
3 7 9 0 5 1 6       3 3 2 0 5 1 5
8 4 2 0 6 1 5   →   7 4 4 0 6 1 6
7 3 4 9 8 2         8 7 9 9 8 2
```

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:

```
3 3 2       0 0 1
0 5 1       1 2 2
5 7 4       3 3 4
4 0 6   →   4 5 6
1 6 8       5 6 8
7 9 9       7 7 9
8 2         8 9
```

Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to move a little bit to their correct position.

Actually, the data sequence is not arranged in a two-dimensional array, but held in a one-dimensional array that is indexed appropriately. For instance, data elements at positions 0, 5, 10, 15 etc. would form the first column of an array with 5 columns. The "columns" obtained by indexing in this way are sorted with Insertion Sort, since this method has a good performance with presorted sequences
*/

```
static void shellSort(Elem[] array) {    // Shell Sort
 int[] cols = {1391376, 463792, 198768, 86961,
                 33936, 13776,   4592,  1968,
                   861,   336,    112,    48,
                    21,     7,      3,     1};
  int h;
  for(int k = 0; k < 16; k ++) {
   h = cols[k];
   for (int i=h; i<array.length; i+=h)
    for (int j=i; (j>=h) && (array[j].key()<array[j-h].key()); j-=h)
      DSutil.swap(array, j, j-h);
  } // end for k
} // end shellSort
/*
```

Theorem: With the h-sequence 1, 3, 7, 15, 31, 63, 127, ..., $2^k - 1$, ... Shellsort needs $O(n \cdot \sqrt{n})$ steps for sorting a sequence of length n . [A. Papernov, G. Stasevic: A Method of Information Sorting in Computer Memories. Problems of Information Transmission 1, 63-75 (1965)]

Theorem: With the h-sequence 1, 2, 3, 4, 6, 8, 9, 12, 16, ..., $2^p 3^q$, ... Shellsort needs $O(n \cdot \log(n)^2)$ steps for sorting a sequence of length n. [V. Pratt: Shellsort and Sorting Networks. Garland, New York ('79)]

Theorem: With the h-sequence 1, 5, 19, 41, 109, 209, ... Shellsort needs $O(n^{4/3})$ steps for sorting a sequence of length n. [R. Sedgewick: Algorithms. Addison-Wesley (1988)]

$$h_s = \begin{cases} 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1 & \text{per } s \text{ pari} \\ 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1 & \text{per } s \text{ dispari} \end{cases}$$

Neither tight upper bounds on time complexity nor the best *h*-sequence are known.

Properties

- Discussion Not stable
- O(1) extra space
- Worst case $O(n^{4/3})$ time, using the Sedgewick's h-sequence.
- Adaptive: O(n·lg(n)) time when nearly sorted

Because of its low overhead, relatively simple implementation, adaptive properties, and sub-quadratic time complexity, **shell sort** may be a viable alternative to the O(n·lg(n)) sorting algorithms for some applications.
*/

/*

Bubblesort

*/

```
static void bubbleSort(Elem[] array) {          // Bubble Sort
  boolean flag = true;
  for (int i=0; flag && i<array.length-1; i++){//Bubble up i'th elem
    flag = false;
    for (int j=0; j<array.length-i-1; j++) {
      if (array[j+1].key() < array[j].key()) {
        DSutil.swap(array, j+1, j);
        flag = true;
      }
    }
  }
}
```

/*

Properties

- Stable
- O(1) extra space
- Worst case O(n^2/2) comparisons and swaps
- Adaptive: Θ(n) when nearly sorted

Discussion

Bubble sort has many of the same properties as **insertion sort**, but has slightly higher overhead.
*/

/*

Selection Sort

```
*/
static void selectionSort(Elem[] array) {// Selection Sort
  for (int i=0; i<array.length-1; i++) { // Select i'th record
    int lowindex = i;                    // Remember its index
    for (int j=i+1; j< array.length; j++) // Find the least value
      if (array[j].key() < array[lowindex].key())
         lowindex = j;                   // Put it in place
    DSutil.swap(array, i, lowindex);
  }
}
/*
```

Properties

- Not stable
- O(1) extra space
- $\Theta(n^2)$ comparisons, $\Theta(n)$ swaps
- Not adaptive

Discussion

From the comparisons presented here, one might conclude that **selection sort** should never be used. It does not adapt to the data in any way, so its runtime is always quadratic.

However, selection sort has the property of minimizing the number of swaps (or write-operations). In applications where the cost of swapping items is high, **selection sort** very well may be the algorithm of choice.

*/

/*

Mergesort

The **Mergesort** algorithm is based on a **divide and conquer** strategy. First, the sequence to be sorted is decomposed into two halves (*Divide*). Each half is sorted independently (*Conquer*). Then the two sorted halves are merged to a sorted sequence (*Combine*)
*/

```
static void mergeSort(Elem[] array) {

  Elem[] temp = new Elem[array.length];
  auxMergeSort(array, temp, 0, array.length-1);
}

static void auxMergeSort(Elem[] array, Elem[] temp,
                         int left, int right) {

  int mid = (left+right)/2;         // Select midpoint

  if (left == right) return;        // List has one element

  auxmergesort(array, temp, left, mid);// Mergesort first half

  auxmergesort(array, temp, mid+1, right); // Mergesort second half

  for (int i=left; i<=right; i++)  // Copy the sub-array to temp
    temp[i] = array[i];

  // Do the merge operation back to array
  int i1 = left;
  int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)             //Left sublist exhausted – copy the
      array[curr] = temp[i2++];//remaining el.s in the right sublist

    else if (i2 > right)         // Right sublist exhausted
      array[curr] = temp[i1++];

    else if (temp[i1].key() < temp[i2].key()) // Get smaller value
      array[curr] = temp[i1++];

    else array[curr] = temp[i2++];
  }
} // end auxMergeSort
```

Source code examples based on "A Practical Introduction to Data Structures and Algorithm Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer.

/*

Properties

- Stable
- Θ(n) extra space
- Θ(n·lg(n)) time
- Not adaptive

Discussion

If using Θ(n) extra space is of no concern, then merge sort is an excellent choice. It is simple to implement, and it is the only stable O(n·lg(n)) sorting algorithm.

There do exist linear time **in-place** merge algorithms (for the last step of the algorithm), but they are extremely complex. The complexity is justified for applications such as **external sorting** where the extra space is not available, but one of the important properties (stability) may be lost.

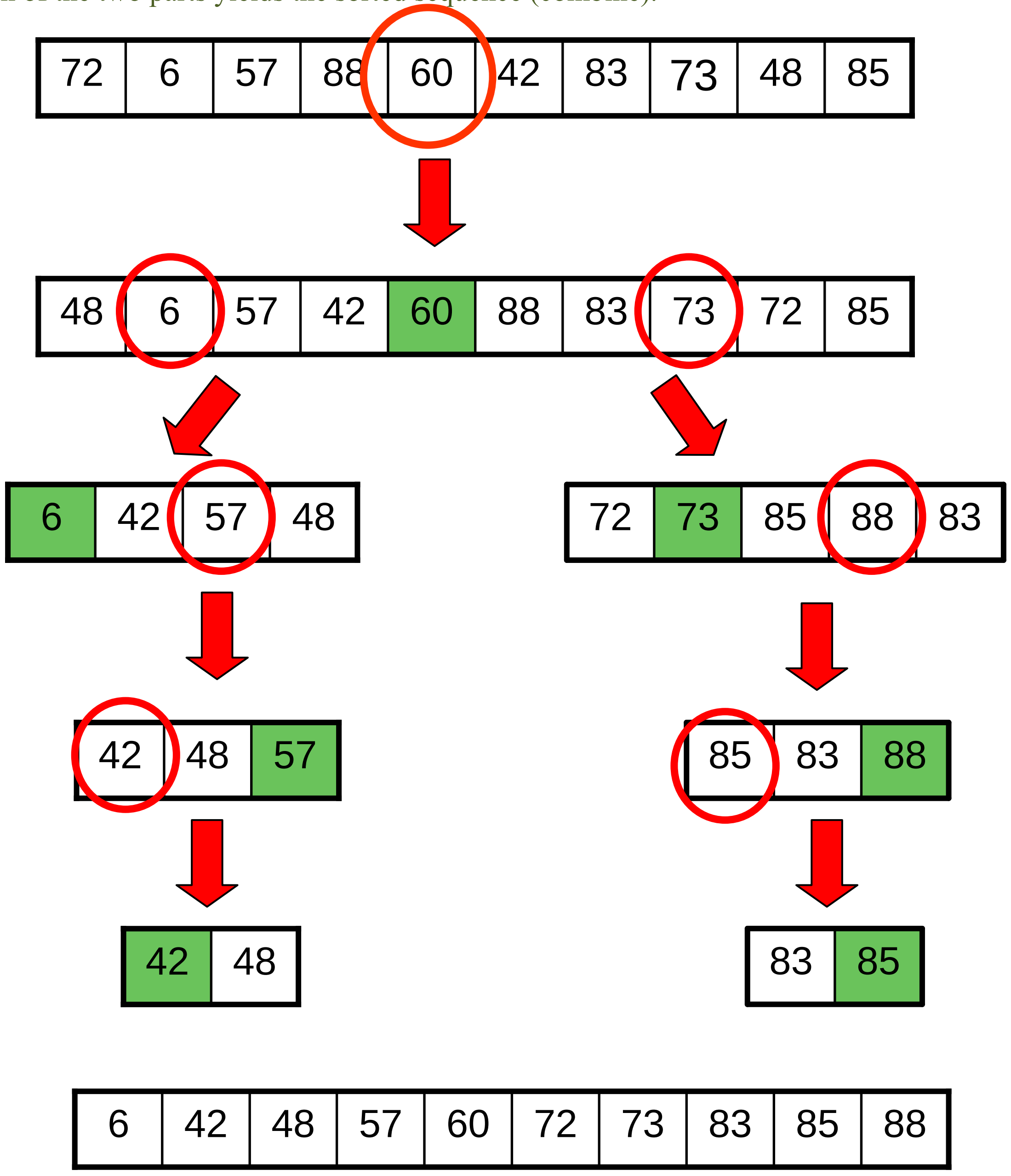
When sorting linked lists, merge sort requires only Θ(lg(n)) extra space for recursion. Given its many other advantages, merge sort is an excellent choice for sorting linked lists.

***/**

/*

Quicksort

Quicksort is one of the fastest and simplest sorting algorithms. It works recursively by a **divide-and-conquer** strategy.
First, the sequence to be sorted is partitioned into two parts, such that all elements of the first part are less than or equal to all elements of the second part (divide).
Then the two parts are sorted separately by recursive application of the same procedure (conquer).
Recombination of the two parts yields the sorted sequence (combine).
*/



Source code examples based on "A Practical Introduction to Data Structures and Algorithm Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer.

```
static void quickSort(Elem[] array) {
  quicksort(array, 0, array.length-1);
}

static int findpivot(Elem[] array, int i, int j) {
     return (i+j)/2;
}

static void quicksort(Elem[] array, int i, int j) {


  int pivotindex = findpivot(array, i, j);  // Pick a pivot

  DSutil.swap(array, pivotindex, j);        // Stick pivot at end
  // the pivot element was moved in the last right position: j.

        // k will be the first position in the right sub-array
        // that is the original position of the pivot
  int k = partition(array, i-1, j, array[j].key());
  DSutil.swap(array, k, j);                 // Put pivot in place

  if ((k-i) > 1) quicksort(array, i, k-1);  // Sort left partition
  if ((j-k) > 1) quicksort(array, k+1, j);  // Sort right partition
}

// Partition the sub-array Elem[] A
// -- the routine returns the first index of the right partition
// -- the "left" and "right" parameters stands for the (first-1) and the (last+1)
// -- index of the current sub-array Elem[] A, respectively.
static int partition(Elem[] A, int left, int right, int pivot) {
  do { // Move the bounds inward until they meet

    // Move left bound to the right
    while (left < right && A[++left].key() <= pivot) ;

    // Move right bound
    while (right!=0 && A[--right].key()>pivot );

    DSutil.swap(A, left, right);      // Swap out-of-place values

  } while (left < right);             // Stop when they cross
  DSutil.swap(A, left, right);        // Reverse last, wasted swap
  return left;         // Return first position in right partition
}
```

Source code examples based on "A Practical Introduction to Data Structures and Algorithm Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer.

/*
Properties

- Not stable
- O(lg(n)) extra space
- O(n^2) time, but typically O(n·lg(n)) time
- Not adaptive

Time complexity of quickSort: Pivot computation O(1) --- Partitioning: O(n)

- Best case: always merge 2 sub-sequences with the same length

$$T(n) = 2T\left(\frac{n}{2}\right) + cn \Rightarrow \Theta(n \log n)$$

- Worst case: always merge 2 sub-sequences, where one of them is of length 1

$$T(n) = T(n-1) + T(1) + cn \Rightarrow \Theta(n^2)$$

- Average case: $\begin{cases} T(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}\left(T(k) + T(n-k-1)\right) \\ T(1) = d \end{cases}$ $T(n) = cn + \frac{2}{n}\sum_{k=0}^{n-1} T(k)$

$$nT(n) = cn^2 + 2\sum_{k=0}^{n-1} T(k) \text{ [moltiplico entrambi i membri per n]}$$

$$(n+1)T(n+1) = c(n+1)^2 + 2\sum_{k=0}^{n} T(k) \quad \text{[n+1]}$$

$$(n+1)T(n+1) - nT(n) \text{ [sottraggo } nT(n)\text{]}$$

$$(n+1)T(n+1) - nT(n) = c(n+1)^2 + 2\sum_{k=0}^{n} T(k) - cn^2 - 2\sum_{k=0}^{n-1} T(k) =$$

$$= c(2n+1) + 2T(n)$$

$$\Rightarrow T(n+1) = \frac{c(2n+1)}{n+1} + T(n)\frac{n+2}{n+1}$$

$$T(n+1) = \frac{c(2n+1)}{n+1} + T(n)\frac{n+2}{n+1} \le 2c + T(n)\frac{n+2}{n+1}$$

$$T(n) \le 2c + T(n-1)\frac{n+1}{n} = 2c + \frac{n+1}{n}\left(2c + T(n-2)\frac{n}{n-1}\right) =$$

$$= 2c + \frac{n+1}{n}\left(2c + \frac{n}{n-1}\left(2c + T(n-3)\frac{n-1}{n-2}\right)\right) =$$

$$= 2c\left\{1 + (n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \ldots + \frac{1}{2}T(1)\right)\right\}$$

The Harmonic series *H(n)= 1/n+1/(n-1)+…* is always bound by

ln n < H(n) < 1 + ln n, therefore

T(n) = O(n logn) */

Source code examples based on "A Practical Introduction to Data Structures and Algorithm Analysis" by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer.

/*

Discussion

When carefully implemented, **quicksort** has low overhead. When a stable sort is not needed, **quicksort** is a decent general-purpose sort.

The time complexity ranges between n lg(n) and n^2 depending on the key distribution and the pivot choices. When the key distribution has many different values, choosing a pivot randomly guarantees O(n lg(n)) time with very high probability. However, when there are O(1) unique keys, the standard 2-way partitioning quicksort exhibits its worst case $O(n^2)$ time complexity no matter the pivot choices.

The algorithm requires O(lg(n)) extra space for the recursion stack in the worst case if recursion is performed on only the *smaller* sub-array; the second recursive call, being tail recursion, may be done with iteration. If both sub-arrays are sorted recursively, then the worst case space requirement grows to O(n).

*/

```
static void print_array(Elem[] A) {
  System.err.print("\n [ "+A[i]+", ");
  for (int i=1; i<A.length-1; i++) System.err.print(A[i]+", ");
  System.err.print(A[i] + " ]\n ");
}

static void copy_array(Elem[] dest, Elem[] source) {
  for (int i=0; i< source.length; i++) dest[i]=source[i];
}
public static void main(String[] args) {
  final int ARRAYSIZE = 15;
  Elem[] a1 = new Elem[ARRAYSIZE];
  Elem[] a2 = new Elem[ARRAYSIZE];
  Elem[] a3 = new Elem[ARRAYSIZE];
  Elem[] a4 = new Elem[ARRAYSIZE];
  Elem[] a5 = new Elem[ARRAYSIZE];
  Elem[] a6 = new Elem[ARRAYSIZE];

  for (int i=0; i<ARRAYSIZE; i++)
    a1[i] = new IElem(DSutil.random(100));  // Random

  copy_array(a2,a1); copy_array(a3,a1); copy_array(a4,a1);
  copy_array(a5,a1); copy_array(a6,a1);

  System.err.print("original : \n ");print_array(a1);
  insertionSort(a1);
  System.err.print("insertionSort : \n ");print_array(a1);

  shellSort(a2);
  System.err.print("shellSort : \n ");print_array(a2);

  bubbleSort(a3);
  System.err.print("bubbleSort : \n ");print_array(a3);

  selectionSort(a4);
  System.err.print("selectionSort : \n ");print_array(a4);

  mergeSort(a5);
  System.err.print("mergeSort : \n ");print_array(a5);

  quickSort(a6);
  System.err.print("quickSort : \n ");print_array(a6);
}
}
```