INTERNAL SORTING

C++ IMPLEMENTATION

```
// File: Book.h
// From the software distribution accompanying the textbook
// "A Practical Introduction to Data Structures and Algorithm Analysis,
// Third Edition" by Clifford A. Shaffer, Prentice Hall, 2007.
// Source code Copyright (C) 2006 by Clifford A. Shaffer.
#ifndef _____BOOK_H_____
#define _____BOOK_H_____
#include <time.h>  // Used by timing functions
#include <iostream>
#include <stdlib.h>
using namespace std;

namespace sorting {

// A collection of various macros, constants, and small functions
// used for the software examples.

inline bool EVEN(int x) { return (x % 2) == 0; } // Return true iff x is even
inline bool ODD(int x) { return (x & 1) != 0; } // Return true iff x is odd

// Swap two elements in a generic array
template<class Elem>
inline void swap(Elem A[], int i, int j) {
  Elem temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
// Random number generator functions
inline void Randomize() { srand((unsigned)time( NULL )); } // Seed the generator
inline int Random(int n) { return rand() % (n); } // Return a value in [0,n-1]

#define THRESHOLD 9
template<class Elem> void print_array(Elem a[], int n) {
    int k;
    cout << "\n[ ";
    for (k= 0; k < n-1; k++)
          cout << a[k] << ", ";
    cout << a[k] << " ]\n";
}

template<class Elem> void copy_array(Elem dest[], Elem source[], int n) {
    for (int k= 0; k < n; dest[k] = source[k], k++) ;
}

class Int { // Your basic int type as an object.
private:
  int val;
public:
  Int(int input=0) { val = input; }
  // The following is for those times when we actually
  //   need to get a value, rather than compare objects.
  int key() const { return val; }
  // Overload = to support Int foo = 5 syntax
  Int operator= (int input) { val = input; }
};
} #endif
```

```
// File: Compare.h
#ifndef _____COMPARE_H_____
#define _____COMPARE_H_____
#include <string.h>
namespace sorting {   //Some definitions for Comparator classes

class getintKey { // Get the key from an int
public:
  static int key(int x) { return x; }
};

class getIntKey { // Get the key from an Int object
public:
  static int key(Int x) { return x.key(); }
};

class getIntsKey { // Get the key from a pointer to an Int object
public:
  static int key(Int* x) { return x->key(); }
};

class IntIntCompare {
public:
  static bool lt(Int x, Int y) { return x.key() < y.key(); }
  static bool eq(Int x, Int y) { return x.key() == y.key(); }
  static bool gt(Int x, Int y) { return x.key() > y.key(); }
};

class IntsIntsCompare {
public:
  static bool lt(Int* x, Int* y) { return x->key() < y->key(); }
  static bool eq(Int* x, Int* y) { return x->key() == y->key(); }
  static bool gt(Int* x, Int* y) { return x->key() > y->key(); }
};

class intintCompare {
public:
  static bool lt(int x, int y) { return x < y; }
  static bool eq(int x, int y) { return x == y; }
  static bool gt(int x, int y) { return x > y; }
};

class CCCompare { // Compare two character strings
public:
  static bool lt(char* x, char* y)
    { return strcmp(x, y) < 0; }
  static bool eq(char* x, char* y)
    { return strcmp(x, y) == 0; }
  static bool gt(char* x, char* y)
    { return strcmp(x, y) > 0; }
};

// Get the key for a character string, the key is just the string itself
class getCKey {
public:
  static char* key(char* x) { return x; }
};
} #endif
```

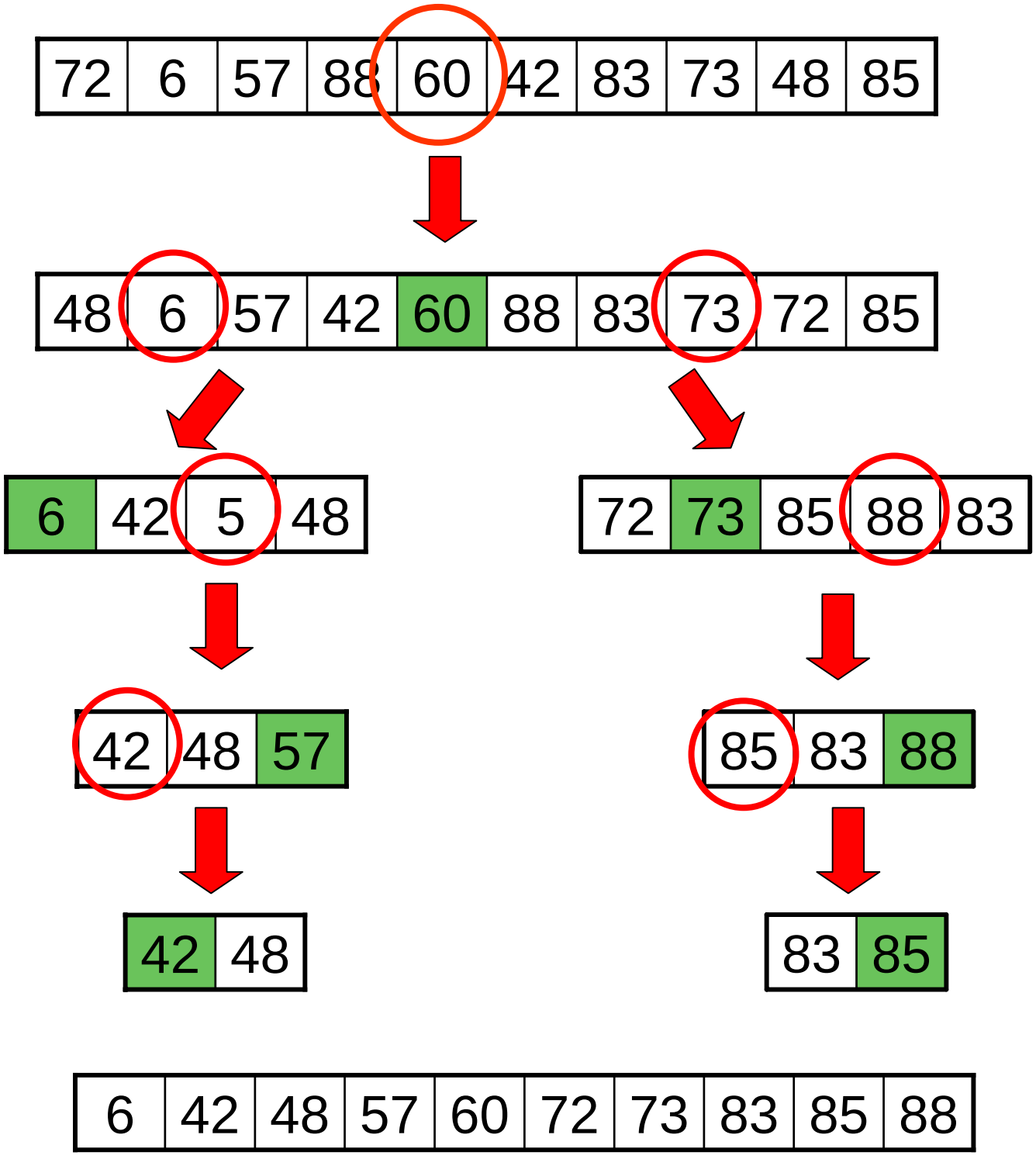
// File: Sorting.h

```
// From the software distribution accompanying the textbook
// "A Practical Introduction to Data Structures and Algorithm Analysis,
// Third Edition" by Clifford A. Shaffer, Prentice Hall, 2007.
// Source code Copyright (C) 2006 by Clifford A. Shaffer.
#include <iostream>
using namespace std;
#include "book.h"
#include "compare.h"
namespace sorting {
// -----------------------------------------------------------------------
// Insertion sort implementation
template <class Elem, class Comp>
void insertionsort(Elem A[], int n) {   // Insertion Sort

  for (int i=1; i < n; i++)       // Insert i'th record
    for (int j=i; (j>0) && (Comp::lt(A[j], A[j-1])); j--)
      swap(A, j, j-1);

} // end insertionsort

template <class Elem, class Comp>void isort(Elem* array, int n)
{insertionsort<Elem, Comp>(array, n);} // end isort

// -----------------------------------------------------------------------
// Bubble sort implementation
template <class Elem, class Comp>
void bubblesort(Elem A[], int n) {   // Bubble Sort

  for (int i=0; i < n-1; i++)     // Bubble up i'th record
    for (int j=n-1; j > i; j--)
      if (Comp::lt(A[j], A[j-1]))
        swap(A, j, j-1);

} // end bubblesort


template <class Elem, class Comp>
void bsort(Elem* array, int n) {
  bubblesort<Elem, Comp>(array, n);
} // end bsort


template <class Elem, class Comp>
void bubblesort2(Elem A[], int n) { // Bubble Sort variant

  for (int i=0; i<n-1; i++)     // Bubble up i'th record
    for (int j=0; j < n-1-i; j++)
      if (Comp::gt(A[j], A[j+1]))
        swap(A, j, j-1);

} // end bubblesort2

template <class Elem, class Comp>
void bsort2(Elem* array, int n) {
  bubblesort2<Elem, Comp>(array, n);
} // end bsort2
```

```
// ---------------------------------------------------------------------------
// Selection sort implementation
template <class Elem, class Comp>
void selectionsort(Elem A[], int n) { // Selection Sort

  for (int i=0; i < n-1; i++) { // Select i'th record
    int lowindex = i;           // Remember its index
    for (int j=n-1; j>i; j--)   // Find the least value
      if (Comp::lt(A[j], A[lowindex]))
          lowindex = j;           // Put it in place
    swap(A, i, lowindex);
  }

} // end selectionsort


template <class Elem, class Comp>
void ssort(Elem* array, int n) {
  selectionsort<Elem, Comp>(array, n);
} // end ssort
```

Il Quicksort utilizza una strategia ***divide et impera*** per ordinare una lista. I passi sono:

- Scegliere un elemento **pivot** nella lista.
- Riordinare la lista in modo che tutti **gli elementi minori del pivot lo precedano e quelli maggiori lo seguano**. L'algoritmo ad ogni passo pone quindi almeno un elemento (il pivot) nella sua posizione finale. Questo passo viene chiamato ***partitioning***.
- **Riordinare ricorsivamente le due sottoliste** alla sinistra e alla destra del pivot. Se una delle sottoliste è vuota o contiene un elemento la ricorsione termina.



```
// ---------------------------------------------------------------------
// Quicksort implementation




// Simple findpivot: Pick middle value in array
//                   ("the median of threen would be better!")
template <class Elem>
inline int findpivot(Elem A[], int i, int j) { return (i+j)/2; }







// Partition the array
// -- 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 A[], respectively
// -- the pivotal parameter is passed by reference for efficiency reasons
template <class Elem, class Comp>
inline int partition(Elem A[], int left, int right, Elem& pivot) {

  do { // Move the bounds inward until they meet

       // Move the "left index" toward the right direction
         // and compare wrt the pivot
         while (Comp::lt(A[++left], pivot));

         // If the "left index" results to be less than the "right index"
         // then Move the "right index" toward the left direction
         // and compare compare wrt the pivot
         while ((left < right) && Comp::gt(A[--right], pivot));

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

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

```
// quicksort core function: Basic quicksort
template <class Elem, class Comp>
void quicksort(Elem A[], int i, int j) { // Quicksort

      // Don't sort array of 0 or 1 Elem
      if (j <= i) return;

      // Find the index of the pivotal value
      int pivotindex = findpivot(A, i, j);

      // Put the pivot at end of the array [i,j]
      // excluding it from subsequent computation
      swap(A, pivotindex, j);

      // Partition the array [i,j-1]:
      // parameter passing conventions are that
      // the current "left" and "right" indeces of the subarray
      // must be equal to the (first-1) index and the (last+1) index respectively
      int k = partition<Elem,Comp>(A, i-1, j, A[j]);

      // Swap the pivotal element at index j, with the 1st element of
      // the right partition
      swap(A, k, j);

      // Sort the two remaining sub-arrays:
    quicksort<Elem,Comp>(A, i, k-1);
    quicksort<Elem,Comp>(A, k+1, j);

} // end quicksort


template <class Elem, class Comp>
void qsort(Elem* array, int n) {
  quicksort<Elem, Comp>(array, 0, n-1);
} // end qsort
```

Calcolo di Complessità:

- Ricerca del pivot: O(1)
- Partitioning: O(n)
- Costo complessivo
  - Caso ottimo: due liste di uguale dimensione

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

  - Caso pessimo: liste sbilanciate al massimo

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

  - Caso medio

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

$$T(n) = cn + \frac{2}{n}\sum_{k=0}^{n-1} T\left(k\right)$$

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

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

$(n+1)T(n+1) - nT(n)$ [sottraggo $nT(n)$]

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

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

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

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

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

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

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

La serie armonica è maggiorata da logn
Complessivamente quicksort ha complessità:

T(n) = O(n logn)

```
//---------------------------------------------------------------------
// Optimized Quicksort: No function calls, no recursion, and no
// sorting of sublists with length <= THRESHOLD (see book.h);
// final call to inssort
template <class Elem, class Comp>
void quicksort2(Elem array[], int i, int j) {
  static int stack[200];
  int top = -1;
  Elem pivot;
  int pivotindex, left, right;
  int n = j-i+1;

  stack[++top] = i;
  stack[++top] = j;

  while (top > 0) {
    // Pop stack
    j = stack[top--];
    i = stack[top--];

    // Findpivot
    pivotindex = (i+j)/2;
    pivot = array[pivotindex];
    swap(array, pivotindex, j); // stick pivot at end

    // Partition
    left = i-1;
    right = j;
    do {
      while (Comp::lt(array[++left], pivot));
      while ((left < right) && Comp::gt(array[--right], pivot));
      swap(array, left, right);
    } while (left < right);
    swap(array, left, right);      // Undo final swap
    swap(array, left, j);          // Put pivot value in place

    // Load up stack.  "left" is pivot point.
    if ((left-1-i) > THRESHOLD) {
      stack[++top] = i;
      stack[++top] = left-1;
    }
    if ((j-left-1) > THRESHOLD) {
      stack[++top] = left+1;
      stack[++top] = j;
    }
  } // end while

  inssort(array, n);
} // end quicksort2


template <class Elem, class Comp>
void qsort2(Elem* array, int n) {
  quicksort2<Elem, Comp>(array, 0, n-1);
} // end qsort2
```

```
//-----------------------------------------------------------------------
// Basic mergesort implementation
template <class Elem, class Comp>
void mergesort(Elem A[], Elem temp[], int left, int right) {

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

  int mid = (left+right)/2;

  mergesort<Elem,Comp>(A, temp, left, mid);

  mergesort<Elem,Comp>(A, temp, mid+1, right);

  // Copy subarray A[left, right] to temp[left, right]
  for (int i=left; i<=right; temp[i] = A[i], i++) ;

  //
  // Do the merge operation back to A
  //
  int i1 = left;    // index for the 1st portion: temp[left,mid]
  int i2 = mid + 1; // index for the 2nd portion: temp[mid+1,right]

  for (int curr=left; curr<=right; curr++) {

    if (i1 == mid+1)        // Left sublist exhausted,
      A[curr] = temp[i2++]; // next instructions will eventually copy
                            // the remaining velues of temp pointed by i2

    else if (i2 > right)    // Right sublist exhausted
      A[curr] = temp[i1++]; // next instructions will eventually copy
                            // the remaining velues of temp pointed by i1

    else if (Comp::lt(temp[i1], temp[i2]))
          A[curr] = temp[i1++];

    else
          A[curr] = temp[i2++];
  }

} // end mergesort

template <class Elem, class Comp>
void msort(Elem* array, int n) {
  static Elem* temp = NULL;
  if (temp == NULL) temp = new Elem[n];  // Declare temp array
  mergesort<Elem,Comp>(array, temp, 0, n-1);
} // end msort



}
```

//File: main.cpp

```
#include <iostream>
using namespace std;
#include "book.h"
#include "compare.h"
#include "sorting.h"
using namespace sorting;


int ELEMSIZE = 32003;
int ARRAYSIZE = 10;


int main() {

  int* a1 = new int[ARRAYSIZE];
  int* a2 = new int[ARRAYSIZE];
  int* a3 = new int[ARRAYSIZE];
  int* a4 = new int[ARRAYSIZE];
  int* a5 = new int[ARRAYSIZE];

  Randomize();for (int i=0; i < ARRAYSIZE; a1[i] = Random(ELEMSIZE), i++);

  copy_array<int>(a2,a1,ARRAYSIZE); copy_array<int>(a3,a1,ARRAYSIZE);
  copy_array<int>(a4,a1,ARRAYSIZE); copy_array<int>(a5,a1,ARRAYSIZE);

  print_array<int>(a1, ARRAYSIZE);
  print_array<int>(a2, ARRAYSIZE);
  print_array<int>(a3, ARRAYSIZE);
  print_array<int>(a4, ARRAYSIZE);
  print_array<int>(a5, ARRAYSIZE);


  isort<int, intintCompare>(a1,ARRAYSIZE);
  bsort<int, intintCompare>(a2,ARRAYSIZE);
  ssort<int, intintCompare>(a3,ARRAYSIZE);
  qsort<int, intintCompare>(a4,ARRAYSIZE);
  msort<int, intintCompare>(a5,ARRAYSIZE);

  print_array<int>(a1, ARRAYSIZE);
  print_array<int>(a2, ARRAYSIZE);
  print_array<int>(a3, ARRAYSIZE);
  print_array<int>(a4, ARRAYSIZE);
  print_array<int>(a5, ARRAYSIZE);


  return(0);
}
```

JAVA IMPLEMENTATION

// DSutil.java
// Source code example for "A Practical Introduction to Data Structures and Algorithm Analysis"
// by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

```
import java.util.*;

// A bunch of utility functions.
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;
  }

  // Randomly permute the Objects in an array
  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

  // Create a random number function to return values
  // uniformly distributed within the range 0 to n-1.
  static private Random value = new Random();// Random class object
    static int random(int n) { // My own function
        return Math.abs(value.nextInt()) % n;
    }
  }
```

//Elem.java
// Source code example for "A Practical Introduction to Data Structures and Algorithm Analysis"
// by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

// 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
} // interface Elem
```

```
// IElem.java
// Source code example for "A Practical Introduction to Data Structures and Algorithm Analysis"
// by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

// 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 int key() { return value; }
  public void setkey(int v) { value = v; }

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

}
```

```
//Sortmain.java
// Source code example for "A Practical Introduction to Data Structures and Algorithm Analysis"
// by Clifford A. Shaffer, Prentice Hall, 1998. Copyright 1998 by Clifford A. Shaffer

import java.io.*;

public class Sortmain {

static int THRESHOLD = 9;
static int ARRAYSIZE = 10;

//-----------------------------------------------------------------
static void isort(Elem[] array) {
  insertionsort(array);
}

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);
}
```

```
//-----------------------------------------------------------------
static void bsort(Elem[] array) {
  bubblesort(array);
}

static void bubblesort(Elem[] array) {    // Bubble Sort
  for (int i=0; i<array.length-1; i++)    // Bubble up i'th record
    for (int j=array.length-1; j>i; j--)
      if (array[j].key() < array[j-1].key())
        DSutil.swap(array, j, j-1);
}
```

```
//-----------------------------------------------------------------
static void ssort(Elem[] array) {
  selectionsort(array);
}

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=array.length-1; j>i; j--) // Find the least value
      if (array[j].key() < array[lowindex].key())
     lowindex = j;                       // Put it in place
    DSutil.swap(array, i, lowindex);
  }
}
```

```
//-----------------------------------------------------------------
static void msort(Elem[] array) {
  Elem[] temp = new Elem[ARRAYSIZE];
  mergesort(array, temp, 0, array.length-1);
}

static void mergesort(Elem[] array, Elem[] temp, int l, int r) {
  int mid = (l+r)/2;                // Select midpoint
  if (l == r) return;               // List has one element
  mergesort(array, temp, l, mid);   // Mergesort first half
  mergesort(array, temp, mid+1, r); // Mergesort second half
  for (int i=l; i<=r; i++)          // Copy subarray to temp
    temp[i] = array[i];
  // Do the merge operation back to array
  int i1 = l; int i2 = mid + 1;
  for (int curr=l; curr<=r; curr++) {
    if (i1 == mid+1)                // Left sublist exhausted
      array[curr] = temp[i2++];
    else if (i2 > r)                // 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 mergesort
```

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

static void quicksort(Elem[] array, int i, int j) { // Quicksort
  int pivotindex = findpivot(array, i, j);  // Pick a pivot
  DSutil.swap(array, pivotindex, j);        // Stick pivot at end
  // k will be the first position in the right subarray
  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
}

static int partition(Elem[] array, int l, int r, int pivot) {
  do {                // Move the bounds inward until they meet
    while (array[++l].key() < pivot); // Move left bound right
    while ((r!=0) && (array[--r].key()>pivot)); // Move right bound
    DSutil.swap(array, l, r);         // Swap out-of-place values
  } while (l < r);                    // Stop when they cross
  DSutil.swap(array, l, r);           // Reverse last, wasted swap
  return l;           // Return first position in right partition
}

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


static void print_array(Elem[] array) {
  int i = 0;
  System.out. print("\n [ "+array[i]+", ");
  for (i=1; i< array.length-1; i++)
      System.out. print(array[i] + ", ");
  System.out. print(array[i] + " ]\n ");
}

static void copy_array(Elem[] dest, Elem[] source) {
  for (int i=0; i< source.length; i++)
      dest[i]=source[i];
}
```

```
// Main routine for sorting class driver
// This is the version for running timings
public static void main(String[] args) {
  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];

  for (int i=0; i<ARRAYSIZE; i++)
      a1[i] = new IElem(DSutil.random(32000));  // Random

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

  isort(a1);  print_array(a1);
  bsort(a2);  print_array(a2);
  ssort(a3);  print_array(a3);
  qsort(a4);  print_array(a4);
  msort(a5);  print_array(a5);
}

}
```