```
/*
 * SortingAlgorithms.cc - demonstration of various sorting methods
 *
 * Copyright (c) 2001, 2003, 2013 - Russell C. Bjork
 *
 */

#include <string>
#include <iostream>
using namespace std;

/*****************************************************************************
 * AUXILIARY ROUTINES NEEDED BY VARIOUS SORTING ALGORITHMS
 *****************************************************************************/

// Exchange two items

void exchange(string & item1, string & item2)
{
    string temp = item1;
    item1 = item2;
    item2 = temp;
}

// Recursive auxiliary to quick sort - fully-defined below.
void quickSortAux(int lo, int hi, string key[]);

// Auxiliary to heap sort - fully-defined below.
void percolateDown(int size, int hole, string key[]);

// Recursive auxiliary method for merge sort - fully-defined below.
void mergeSortAux(int lo, int hi, string key[]);

/*****************************************************************************
 * Implementations of various different sorting algorithms.
 *
 * Parameters:
 *
 *   n - the number of items to sort
 *   key[] - the array of keys to sort
 *    (0-origin indexing - we sort key[0] .. key[n-1])
 *
 *****************************************************************************/
```

```
/*****************************************************************************
 * SIMPLE INSERTION SORT
 *****************************************************************************/

void insertionSort(int n, string key[])
{
    for (int i = 1; i < n; i ++)
    {
        // Invariant: key [] is a permutation of the original array
        // key[0] .. key[i-1] is sorted

        // On each pass, we add key[i] to the set of already sorted keys

        string temp = key[i];

        // Find the right place to insert temp relative to already sorted keys.
        // Items that belong after temp are moved up to make room for it.
        // j ends up referring to the slot where temp belongs.

        int j;
        for (j = i; j > 0 && key[j - 1] > temp; j --)
            key[j] = key[j - 1];

        // Keys in slots 0 .. j - 1 are <= temp.  Keys in slots j + 1 .. i
        // are > temp

        key[j] = temp;
    }
}

/*****************************************************************************
 * BUBBLE SORT (BASIC FORM - W/O IMPROVEMENTS)
 *****************************************************************************/

void bubbleSort(int n, string key[])
{
    for (int i = 1; i < n; i ++)
    {
        // Invariant: key [] is a permutation of the original array
        // key[n - i + 1] .. key[n - 1] is sorted, and all keys in
        // key[n - i + 1] .. key[n - 1] are >= all keys in key[0] .. key[n - i]

        for (int j = 0; j < n - i; j ++)
        {
            if (key[j] > key[j + 1])
                exchange(key[j], key[j + 1]);
        }
    }
}
```

```
/*****************************************************************************
 * QUICK SORT (BASIC FORM - W/O IMPROVEMENTS)
 *****************************************************************************/

void quickSort(int n, string key[])
{
    quickSortAux(0, n - 1, key);
}

// Recursive auxiliary that does the work of quicksort - parameters
// are lower and upper indices of key[] to be sorted

void quickSortAux(int lo, int hi, string key[])
{
    if (lo >= hi)
        ; // Base case: an array of 0 or 1 items is already sorted - do nothing
    else
    {
        string pivot = key[lo];
        int i = lo, j = hi + 1;

        // Look for a pair of items such that key[i] occurs before key[j] in
        // the original data (i.e. i < j), but key[i] belongs after the pivot
        // and key[j] belongs before it.  Switch these two items; stop when
        // i reaches or passes j.  When we finish, j will be the last slot
        // whose key belongs at or before the pivot

        while (i < j)
        {
            do
                i ++;
            while (i < hi && key[i] <= pivot);

            do
                j --;
            while (j > lo && key[j] >= pivot);

            if (i < j)
                exchange(key[i], key[j]);
        }

        // Put pivot in its correct place relative to other keys

        key[lo] = key[j];
        key[j] = pivot;

        // Sort each subarray recursively

        quickSortAux(lo, j - 1, key);
        quickSortAux(j + 1, hi, key);
    }
}
```

```
/*****************************************************************************
 * SIMPLE SELECTION SORT
 *****************************************************************************/

void selectionSort(int n, string key[])
{
    for (int i = 0; i < n - 1; i ++)
    {
        // Invariant: key [] is a permutation of the original array
        // key[0] .. key[i-1] is sorted, and all keys in key[0]..key[i-1]
        // are <= all keys in key[i] .. key[n-1]

        // On each pass, we identify the key that belongs in slot i - which
        // will always be the smallest of the remaining keys

        // smallest will be the index of the smallest key in key[i] ..key [n-1]

        int smallest = i;
        for (int j = i + 1; j < n; j ++)

            if (key[j] < key[smallest])
                smallest = j;

        // Put key[smallest] in its proper place - slot i

        exchange(key[smallest], key[i]);
    }
}
```

```
/*****************************************************************************
 * HEAP SORT
 *****************************************************************************/

// A tricky problem with implementing the heapsort in C++ is that the heap
// structure assumes elements of the heap are stored in slots 1 .. n, but
// C++ wants array subscripts to range from 0 .. n - 1.  We handle this by
// using a macro to effectively make Key an array with 1-origin indexing such
// that Key[1] corresponds to key[0] .. Key[n] corresponds to key[n-1]

#define Key (key-1)

void heapSort(int n, string key[])
{
    // Turn the array into a maxheap

    for (int i = n / 2; i > 0; i --)
        percolateDown(n, i, Key);

    // Convert the maxheap into a sorted array

    for (int i = n; i > 0; i --)
    {
        // Put the correct key in slot n by exchanging with the
        // key in slot 1; then percolate the key in slot 1 down.
        // We use i as the size of the heap, because after an item is in
        // place it is no longer considered part of the heap

        exchange(Key[1], Key[i]);
        percolateDown(i-1, 1, Key);
    }
}

// Auxiliary to heap sort - based on Weiss, page 218.  Parameters are current
// size of the heap, the key to potentially percolate down, and the heap itself.

void percolateDown(int size, int hole, string key [])
{
    int child;
    string temp = key [hole];

    for ( ; hole * 2 <= size; hole = child)
    {
        child = hole * 2;
        if (child < size && key [child + 1] > key [child] )
            child ++;
        if (key [child] > temp)
            key [hole] = key [child];
        else
            break;
    }

    key [hole] = temp;
}
```

```
/*****************************************************************************
 * MERGE SORT
 *****************************************************************************/

void mergeSort(int n, string key[])
{
    mergeSortAux(0, n, key);
}

// Recursive auxiliary that does the work of merge sort - parameters
// are lower and 1 more than upper limits of indices of key[] to be sorted

void mergeSortAux(int lo, int hi, string key[])
{
    int numberOfItems = hi - lo;
    int lowerSize = numberOfItems / 2;
    int upperSize = numberOfItems - lowerSize;  // May differ if # is odd

    if (numberOfItems <= 1)
    {
        // Base case: an array of 0 or 1 items is already sorted - do nothing
    }
    else
    {
        // Partition into two halves in temporary arrays; sort each half
        // recursively; merge the two halves

        string lower[lowerSize];
        string upper[upperSize];

        for (int i = 0; i < lowerSize; i ++)
            lower[i] = key[lo + i];
        for (int i = 0; i < upperSize; i ++)
            upper[i] = key[lo + lowerSize + i];

        mergeSortAux(0, lowerSize, lower);
        mergeSortAux(0, upperSize, upper);

        // Merge the two partitions back into original array

        for (int i = lo, lowerIndex = 0, upperIndex = 0; i < hi; i ++)
        {
            if (lowerIndex >= lowerSize)
                key[i] = upper[upperIndex ++];
            else if (upperIndex >= upperSize)
                key[i] = lower[lowerIndex ++];
            else if (lower[lowerIndex] <= upper[upperIndex])
                key[i] = lower[lowerIndex ++];
            else
                key[i] = upper[upperIndex ++];
        }
    }
}
```

```
/*****************************************************************************
 * TEST DRIVER CODE
 *****************************************************************************/

/* Tester for the various sorts */

int N = 26;
const char * initialData[] = { "PENGUIN", "KANGAROO", "DOG", "QUAIL", "XERUS",
                               "HIPPO", "LLAMA", "AARDVARK", "CAT", "IGUANA",
                               "SNAKE", "RACCOON", "TURTLE", "ELEPHANT", "MOUSE",
                               "JACKAL", "UNICORN", "BUFFALO", "NEWT", "YAK",
                               "VIXEN", "FOX", "ZEBRA", "GOPHER",
                               "WATERBUFFALO", "OSPREY"
                     };

void testASort(string name, void (* sort) (int, string []))
{
    string data [N];

    // Copy the initial data fresh for each sort, so each sort starts out
    // with the same raw data

    for (int i = 0; i < N; i ++)
        data[i] = initialData[i];

    // Do the sort

    cout << "Test of " << name << " produces" << endl;

    sort(N, data);

    // Print out the results

    for (int i = 0; i < N; i ++)
        cout << data[i] << endl;

    cout << endl;
}

int main(int argc, char * argv[])
{
    testASort("Insertion Sort", insertionSort);
    testASort("Bubble Sort", bubbleSort);
    testASort("Quick Sort", quickSort);
    testASort("Selection Sort", selectionSort);
    testASort("Heap Sort", heapSort);
    testASort("Merge Sort", mergeSort);
}
```