```
/*********************************************************************
 * quicksort.c --- Using quicksort and binary search, sort two int
 *                 arrays and search for values within them.
 *********************************************************************/


void quicksort(int array[], int l, int r);
int partition(int array[], int l, int r);
void swap(int array[], int m, int n);
int binarysearch(int array[], int key, int l, int r);


/* Ten random ints in the interval [0, 100). */

int data1[10] = { 34, 14, 85, 18, 80, 81, 55, 10, 41, 56 };

/* Four search keys for data1[].  85 and 41 are in data1[]; 17 and 99
 * are not.
 */

int keys[4] = { 85, 17, 41, 99 };

/* One hundred random ints in the interval [0, 1,000,000). */

int data2[100] =
{
   13202, 421339, 40723, 648473, 93130, 174743, 961310, 163681, 266528,
   809332, 844292, 52042, 528863, 257713, 167976, 428474, 156392, 669761,
   961107, 754483, 34199, 220947, 498637, 987780, 195297, 651326, 523288,
   802797, 676022, 535849, 128108, 205576, 957188, 685184, 854049, 566671,
   859927, 331711, 730352, 642807, 657395, 574644, 211201, 186258, 348709,
   379177, 131085, 505101, 565291, 92192, 775936, 599490, 829491, 274573,
   103622, 24789, 442251, 626911, 343938, 634625, 679112, 472046, 356553,
   152652, 673582, 726954, 719323, 49861, 58666, 966027, 209020, 232413,
   540671, 420222, 935024, 405733, 799399, 66109, 427186, 881042, 674653,
   719475, 996884, 20496, 510400, 100507, 561637, 952652, 727418, 905575,
   103629, 406530, 893974, 460183, 75534, 83908, 187137, 794858, 133770,
   13203
};


/*********************************************************************
 * main()
 *********************************************************************/

int main()
{
   int i;
   int found;
   int passed;
   int failed;

   quicksort(data1, 0, 9);

   /* Perform four binary searches on the sorted data in data1.  Two
    * of the searches will succeed; two will fail.
    */

   found = 0;

   for (i = 0; i < 4; i++)
      if (binarysearch(data1, keys[i], 0, 9) != -1)
         found ++;
```

```
   /* The value of found should be 2. */

   quicksort(data2, 0, 99);

   /* Search for each item already in data2[], checking that the
    * key index returned is correct.
    */

   passed = 1;

   for (i = 0; i < 100; i++)
      if (binarysearch(data2, data2[i], 0, 99) != i)
         passed = 0;

   /* The value of passed should be 1. */

   /* Perform 100 binary searches for keys not in data2[]. */

   failed = 0;

   for (i = 0; i < 100; i++)
      if (binarysearch(data2, data2[i] + 1, 0, 99) == -1)
         failed++;

   /* The value of failed should be 99 (0X63). */

   return 0;
}


/*********************************************************************
 * quicksort()
 *
 * Parameters:
 *
 *    array[] --- the array being sorted.
 *
 *    l and r --- the left and right limit indices for the sort.  The
 *                sub-array of elements from array[l] to array[r] is
 *                sorted.  On initial call l and r should be set to 0
 *                and n-1, respectively, where n is the number of
 *                elements in array[].
 *********************************************************************/

void quicksort(int array[], int l, int r)
{
   int pivot;

   if (l < r)
   {
      pivot = partition(array, l, r);
      quicksort(array, l, pivot);
      quicksort(array, pivot + 1, r);
   }
}


/*********************************************************************
 * partition() --- partition a sub-array around a pivot for quicksort()
 *
 * Parameters:
 *
```

```
 *    array[] --- the array being partitioned.
 *
 *    l and r --- the left and right limit indices for partitioning.
 *                The sub-array of elements from array[l] to array[r]
 *                will be partitioned.
 *
 * Returns the index of the pivot element.  Upon exit, the elements of
 * array[] have been rearranged such that no element in array[l] to
 * array[pivot] is greater than any element in array[pivot+1] to
 * array[r].
 ************************************************************************/

int partition(int array[], int l, int r)
{
   int pivot = array[l];
   l--;
   r++;

   while (1)
   {
      do
         r--;
      while (array[r] > pivot);

      /* array[r] belongs in the left half of array[]. */

      do
         l++;
      while (array[l] < pivot);

      /* array[l] belongs in the right half of array[]. */

      /* Put the two elements into the correct half of array[] by
       * swapping them.
       */

      if (l < r)
         swap(array, l, r);
      else
         return r;
   }
}


/************************************************************************
 * swap() --- swap array[m] and array[n].
 ************************************************************************/

void swap(int array[], int m, int n)
{
   int temp = array[m];
   array[m] = array[n];
   array[n] = temp;
}


/************************************************************************
 * binarysearch()
 *
 * Parameters:
 *
 *    array[] --- the pre-sorted int array to search.
 *
```

```
 *    key --- the value being searched for.
 *
 *    l and r --- the left and right limit indices for the search.
 *                The sub-array of elements from array[l] to array[r]
 *                is being searched for key.  On initial call l and r
 *                should be set to 0 and n-1, respectively, where n is
 *                the number of elements in array[].
 *
 * Returns the index of key in array[].  If key not found, returns -1.
 ************************************************************************/

int binarysearch(int array[], int key, int l, int r)
{
   int mid;

   if (l > r)      /* Key not found in array[]. */
      return -1;

   /* The ARM processor we're using has no divide instruction.
    * A little bit of creativity is necessary here.
    */

   mid = (l + r) / 2;

   if (array[mid] > key)   /* Continue searching first half of array[]. */
      return binarysearch(array, key, l, mid - 1);

      /* Continue searching second half of array[]. */

   else if (array[mid] < key)
      return binarysearch(array, key, mid + 1, r);
   else   /* Found key at array[mid]. */
      return mid;
}
```