```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

/* Exchange items at positions @i and @j in array @a. */
static void exchange(int a[], int i, int j)
{
    int v = a[i];
    a[i] = a[j];
    a[j] = v;
}

/*
 * Selection sort:
 *  Iterate @n times. At the end of iteration @k the first @k items in the
 *  list are in their *final* sorted positions.
 * Space: O(1)
 * Time:  O(n^2)
 */
static void selection_sort(int a[], int n)
{
    int i, j, min;

    for ( i = 0; i < n; i++ )
    {
        min = i;
        for ( j = i+1; j < n; j++ )
            if ( a[j] < a[min] )
                min = j;
        exchange(a, i, min);
    }
}

/*
 * Insertion sort:
 *  Iterate @n times. At the end of iteration @k the original first @k
 *  items in the list are in sorted order. The remaining @n-@k items are
 *  untouched.
 * Space: O(1)
 * Time:  O(n^2)
 */
static void insertion_sort(int a[], int n)
{
    int i, j, v;

    for ( i = 0; i < n; i++ )
    {
        v = a[i];
        for ( j = i-1; (j >= 0) && (a[j] > v); j-- )
            a[j+1] = a[j]; /* no need for exchange() */
        a[j+1] = v;
    }
}

/*
 * Bubble sort:
 *  Iterate @n-1 times. On each iteration compare each adjacent pair of items
 *  in the list (implies another @n-1 iterations of an inner loop). Exchange
 *  items if they are out of order.
 * Space: O(1)
 * Time:  O(n^2)
```



```c
 */
static void bubble_sort(int a[], int n)
{
    int i, j;

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

/*
 * Shell sort:
 *  The final iteration is a pure insertion sort, but preceding iterations
 *  ensure that the list is already partially sorted so the last iteration
 *  is not quadratic in time.
 * Space: O(1)
 * Time:  O(n^1.5)
 */
static void shell_sort(int a[], int n)
{
    int i, j, h, v;

    h = 1;
    while ( h <= n/9 )
        h = 3*h+1;

    for ( ; h > 0; h /= 3 )
    {
        for ( i = h; i < n; i++ )
        {
            v = a[i];
            for ( j = i-h; (j >= 0) && (a[j] > v); j -= h )
                a[j+h] = a[j];
            a[j+h] = v;
        }
    }
}

/*
 * Quicksort:
 *  Also know as partition-exchange sort. Select a pivot value, partition the
 *  array about the pivot, then recursively sort the two partitions.
 * Space: O(log(n))   average; O(n)   worst
 * Time:  O(n*log(n)) average; O(n^2) worst
 */
static void __quick_sort(int a[], int l, int r)
{
    int i, j, k, pivot;

    if ( l >= r )
        return;

    i = l-1;
    j = r;
    pivot = a[r];

    while ( i < j )
    {
        while ( a[++i] < pivot )
            continue;
        while ( (a[--j] > pivot) && (i < j) )
```

```c
            continue;
        if ( i < j )
            exchange(a, i, j);
    }

    a[r] = a[i];
    a[i] = pivot;

    __quick_sort(a, l, i-1);
    __quick_sort(a, i+1, r);
}

static void quick_sort(int a[], int n)
{
    __quick_sort(a, 0, n-1);
}

/*
 * Merge sort:
 *  Recursively mergesort the two halves of the list, then copy the sorted
 *  first half to a temporary holding array. Then merge the (sorted) temporary
 *  array and the (sorted) second half of the original array into the original
 *  array.
 * Space: O(n)
 * Time:  O(n*log(n))
 */
static void __merge_sort(int a[], int l, int r)
{
    int m, i, j, k, *tmp;

    if ( l >= r )
        return;

    m = (l+r)/2;

    __merge_sort(a, l, m);
    __merge_sort(a, m+1, r);

    tmp = malloc((m-l+1)*sizeof(int));
    memcpy(tmp, &a[l], (m-l+1)*sizeof(int));

    i = l;
    j = m+1;
    for ( k = l; k <= r; k++ )
    {
        if ( i > m )
            break;
        if ( j > r )
        {
            memcpy(&a[k], &tmp[i-l], (r-k+1)*sizeof(int));
            break;
        }
        if ( tmp[i-l] < a[j] )
            a[k] = tmp[i++-l];
        else
            a[k] = a[j++];
    }

    free(tmp);
}

static void merge_sort(int a[], int n)
```



```c
{
    __merge_sort(a, 0, n-1);
}

/*
 * Heapify:
 *  Given an array representation of a heap, sink the value at index @p to
 *  its correct location. Assumes that the two subheap children of index @p
 *  already obey the heap property.
 *
 *  This function builds a "max heap": the item at the root of the heap is
 *  the largest item in the array.
 */
static void __heapify(int a[], int p, int n)
{
    int l, r;

    for ( ; ; )
    {
        l = 2*p+1;
        r = 2*p+2;

        if ( l >= n )
            break;

        if ( r >= n )
        {
            if ( a[l] > a[p] )
                exchange(a, l, p);
            break;
        }

        if ( (a[l] > a[p]) && (a[l] > a[r]) )
        {
            exchange(a, l, p);
            p = l;
        }
        else if ( a[r] > a[p] )
        {
            exchange(a, r, p);
            p = r;
        }
        else
            break;
    }
}

/*
 * Heap sort:
 *  Turns the array into a "max heap" (see definition of the __heapify
 *  function). The second phase then iterates @n times, taking the next-largest
 *  item from the heap and placing it at its final location in the sorted
 *  array.
 *
 *  Heap sort can be considered a more efficient version of the selection sort,
 *  where the cost of building a heap is repaid by more efficient selection of
 *  the next item to place in its final location.
 * Space: O(1)
 * Time:  O(n*log(n))
 */
static void heap_sort(int a[], int n)
{
```

```c
    int i;

    /* Phase 1: build a max heap. */
    for ( i = n/2; i >= 0; i-- )
        __heapify(a, i, n);

    /* Phase 2: efficient 'selection sort' using the heap structure. */
    for ( i = n-1; i > 0; i-- )
    {
        exchange(a, 0, i);
        __heapify(a, 0, i);
    }
}



/*
 * *************************************
 * TESTING HARNESS
 * *************************************
 */

static struct {
    void (*fn)(int [], int);
    char *name;
} sort_methods[] = {
    { selection_sort, "Selection" },
    { insertion_sort, "Insertion" },
    { bubble_sort,    "Bubble" },
    { shell_sort,     "Shell" },
    { quick_sort,     "Quick" },
    { merge_sort,     "Merge" },
    { heap_sort,      "Heap" },
    { NULL, NULL }
};

int main(int argc, char **argv)
{
    int size, i, *orig, *sort;
    struct timeval tv1, tv2;
    long sec, msec;

    if ( argc != 3 )
    {
    usage:
        fprintf(stderr, "%s reverse|ordered|small|random <size>\n",
                argv[0]);
        fprintf(stderr, " Sort inputs:\n");
        fprintf(stderr, "  reverse: reverse order\n");
        fprintf(stderr, "  ordered: already sorted order\n");
        fprintf(stderr, "  small:   values in range 0-9 (many duplicates)\n");
        fprintf(stderr, "  random:  random order, full key range\n");
        return 0;
    }

    size = atoi(argv[2]);
    if ( (size < 0) || (size > 10000000) )
    {
        fprintf(stderr, "Size %d is out of range\n", size);
        return 0;
    }
    printf("Using %d elements\n", size);
```



```c
    orig = malloc(size * sizeof(int));
    sort = malloc(size * sizeof(int));
    if ( (orig == NULL) || (sort == NULL) )
    {
        fprintf(stderr, "Out of memory\n");
        return 0;
    }

    for ( i = 0; i < size; i++ )
        orig[i] = rand();

    if ( !strcmp(argv[1], "reverse") )
    {
        shell_sort(orig, size);
        for ( i = 0; i < (size-i-1); i++ )
            exchange(orig, i, size-i-1);
    }
    else if ( !strcmp(argv[1], "ordered") )
    {
        shell_sort(orig, size);
    }
    else if ( !strcmp(argv[1], "small") )
    {
        for ( i = 0; i < size; i++ )
            orig[i] %= 10;
    }
    else if ( strcmp(argv[1], "random") )
    {
        goto usage;
    }

    for ( i = 0; sort_methods[i].fn != NULL; i++ )
    {
        printf("%s sort... ", sort_methods[i].name); fflush(stdout);
        memcpy(sort, orig, size * sizeof(int));
        gettimeofday(&tv1, NULL);
        (*sort_methods[i].fn)(sort, size);
        gettimeofday(&tv2, NULL);
        sec = tv2.tv_sec - tv1.tv_sec;
        msec = (tv2.tv_usec - tv1.tv_usec) / 1000;
        if ( msec < 0 )
        {
            sec--;
            msec += 1000;
        }
        printf("%lds %ldms\n", sec, msec);
    }

    return 0;
}
```