```
#define NEED_XMALLOC
#include "C_util.h"
#define get_byte(val, byte)                     \
  (((val) >> ((byte)*CHAR_BIT)) & 0xffu)
#define SWAP(x,y)                               \
  __extension__ ({__typeof(x) __temp = x;                           \
      x = y;                                                        \
      y = __temp;})
//should return true if the 2 arguments are in the correct order
//and false otherwised
typedef int(*cmp_fun)(void*,void*);
typedef void(*sort_fn)(void**, size_t, cmp_fun);
typedef void(*int_sort_fn)(uint64_t*, size_t);
void print_arr(ulong *arr, int len, FILE *out);
int cmp_gt(void *x, void *y){
  return ((uintptr_t)x) > ((uintptr_t) y);
}
int cmp_lt(void *x, void *y){
  return ((uintptr_t)x) < ((uintptr_t) y);
}
//threshold for switching to insertion sort in merge/quick sort
//this is the size used by glibc
#define INSERTION_SORT_THRESHOLD 4
void insertion_sort_generic(void **input, size_t len, cmp_fun cmp){
  uint i,j;
  for(i=1;i<len;i++){
    void *temp = input[i];
    j = i;
    while(j>0 && !cmp(input[j-1],temp)){
      input[j] = input[j-1];
      j--;
    }
    input[j] = temp;
  }
}
void insertion_sort_u64(uint64_t *input, size_t len){
  uint i,j;
  for(i=1;i<len;i++){
    uint64_t temp = input[i];
    j = i;
    while(j > 0 && (temp < input[j-1])){
      input[j] = input[j-1];
      j--;
    }
    input[j] = temp;
  }
}
static inline long median(long x, long y, long z){
  long lg = MAX(x,y);
  long sm = MIN(x,y);
  long ret = z;
  if(z > lg){
    ret = lg;
  } else if(z < sm){
    ret = sm;
  }
  return ret;
}
/*
  Similar to the algorithm used by glibc, but glibc uses an explict stack
  while we use recursion, since I'm lazy.
*/
static inline int qsort_partition_generic(void **arr, long left, long right,
                                          cmp_fun cmp){
  long pivot_idx = median(left, right, (left+right)/2);
  void* pivot = arr[pivot_idx];
  long i = left - 1, j = right + 1;
  do {
    do {
```



```
      i++;
    } while(cmp(arr[i], pivot));
    do {
      j--;
    } while(cmp(pivot, arr[j]));
    if(i<j){
      SWAP(arr[i], arr[j]);
    } else {
      return j;
    }
  } while(i<=j);
  return j;
}
void qsort_generic(void **arr, size_t len, cmp_fun cmp){
  if(len <= INSERTION_SORT_THRESHOLD){
    insertion_sort_generic(arr, len, cmp);
  } else {
    long pivot_idx = qsort_partition_generic(arr, 0, len-1, cmp);
    qsort_generic(arr, pivot_idx+1, cmp);
    qsort_generic(arr+(pivot_idx+1), len - (pivot_idx+1), cmp);
  }
  return;
}
static inline int qsort_partition_u64(uint64_t *arr, long left, long right){
  long pivot_idx = median(left, right, (left + (right-left)/2));
  uint64_t pivot = arr[pivot_idx];
  long i = left - 1, j = right + 1;
  do {
    do {
      i++;
    } while(arr[i] < pivot);
    do {
      j--;
    } while(pivot < arr[j]);
    if(i<j){
      SWAP(arr[i], arr[j]);
    } else {
      return j;
    }
  } while(i<=j);
  return j;
}
void qsort_u64(uint64_t *arr, size_t len){
  if(len <= INSERTION_SORT_THRESHOLD){
    //putting a check for non-zero len here might speed things up
    insertion_sort_u64(arr, len);
  } else {
    long pivot_idx = qsort_partition_u64(arr, 0, len-1);
    qsort_u64(arr, pivot_idx+1);
    qsort_u64(arr+(pivot_idx+1), len - (pivot_idx+1));
  }
  return;
}
static void merge(void **A, void **B, void **tmp,
                  size_t n1, size_t n2, cmp_fun cmp){
  while(n1 > 0 && n2 > 0){
    if(cmp(*A,*B)){
      *tmp++ = *A++;
      n1--;
    } else {
      *tmp++ = *B++;
      n2--;
    }
  }
  if(n1 > 0){
    memcpy(tmp, A, n1*sizeof(void*));
  }
  if(n2 > 0){
    memcpy(tmp, B, n2*sizeof(void*));
```

```
  }
}
static void mergesort_internal(void **arr, void **tmp,
                               size_t len, cmp_fun cmp){
  if(len <= INSERTION_SORT_THRESHOLD){
    insertion_sort_generic(arr, len, cmp);
    return;
  }
  size_t mid = len/2;
//  size_t n2 = len - n1;
//  void **A = arr;
//  void **B = arr + n1;
  mergesort_internal(arr, tmp, mid, cmp);
  mergesort_internal(arr + mid, tmp, len - mid, cmp);
  merge(arr, arr + mid, tmp, mid, len - mid, cmp);
  memcpy(arr, tmp, len*sizeof(void*));
}
void mergesort_generic(void **arr, size_t len, cmp_fun cmp){
  //alloca would be faster, but limits len to 8M/sizeof(void*)
  //and I'm too lazy to test the length
  void **tmp = zmalloc(len*sizeof(void*));
  mergesort_internal(arr, tmp, len, cmp);
  free(tmp);
}

static void merge_u64(uint64_t *A, uint64_t *B, uint64_t *tmp,
                      size_t n1, size_t n2){
  while(n1 > 0 && n2 > 0){
    if(*A<*B){
      *tmp++ = *A++;
      n1--;
    } else {
      *tmp++ = *B++;
      n2--;
    }
  }
  if(n1 > 0){
    memcpy(tmp, A, n1*sizeof(void*));
  }
  if(n2 > 0){
    memcpy(tmp, B, n2*sizeof(void*));
  }
}
static void mergesort_internal_u64(uint64_t *arr, uint64_t *tmp, size_t len){
  if(len <= INSERTION_SORT_THRESHOLD){
    insertion_sort_u64(arr, len);
    return;
  }
  size_t mid = len/2;
//  size_t n2 = len - n1;
//  void **A = arr;
//  void **B = arr + n1;
  mergesort_internal_u64(arr, tmp, mid);
  mergesort_internal_u64(arr + mid, tmp, len - mid);
  merge_u64(arr, arr + mid, tmp, mid, len - mid);
  memcpy(arr, tmp, len*sizeof(void*));
}
void mergesort_u64(uint64_t *input, size_t len){
  uint64_t *tmp = zmalloc(len*sizeof(uint64_t));
  mergesort_internal_u64(input, tmp, len);
  free(tmp);
}
/*
  We need to reimplement a heap to get a fast integer heapsort
*/
#define heap_left_child(i) ((2*i)+1)
#define heap_right_child(i) ((2*i)+2)
#define heap_parent(i) ((i-1)/2)
static void sift_down(uint64_t *heap, size_t root, size_t len){
```



```
  int64_t l,r,swap;
  while((l=heap_left_child(root)) < len){
    r = heap_right_child(root);
    swap = root;
    if(heap[l] > heap[swap]){
      swap = l;
    }
    if(r < len && (heap[r] > heap[swap])){
      swap = r;
    }
    if(swap != root){
      SWAP(heap[root], heap[swap]);
      root = swap;
    } else {
      break;
    }
  }
}
static void heapify(uint64_t *heap, size_t len){
  int64_t start = heap_parent(len-1);
  while(start >= 0){
    sift_down(heap, start, len);
    start--;
  }
}
void heapsort_u64(uint64_t *arr, size_t len){
  heapify(arr, len);
  int64_t end = len-1;
  do {
    SWAP(arr[end], arr[0]);
    sift_down(arr, 0, end);
  } while(--end > 0);
}
/*
  This used to use arrays for hist and sum, and loops, but it sometimes failed.
  I manually unrolled all the loops and used seperate variables and it worked.
  I could probably figure out what was wrong and  put back in the loops and arra
ys,
  but this works and I'm lazy.
*/
/*
  Note for posterity:
  I made a mistake when I first wrote this, I set size of the histograms to
  be 255 (i.e 0xff) bytes each, however a byte can take on 256 (i.e 0x100) possi
ble
  values. This caused a subtle bug, the function would work so long as the byte
0xff didn't
  appear in the input, but if it did then memory out of bounds would be written
(generally
  writing to hist_n[0xff] would write to hist_n+1[0] instead). It took me a whil
e
  to realize what was going on.
*/
void radix_sort_u8_u64(uint64_t *in, size_t sz){
  uint64_t *hist = alloca(8 * 0x100 * sizeof(uint64_t));
  uint64_t *hist0 = hist;
  uint64_t *hist1 = hist0 + 0x100;
  uint64_t *hist2 = hist1 + 0x100;
  uint64_t *hist3 = hist2 + 0x100;
  uint64_t *hist4 = hist3 + 0x100;
  uint64_t *hist5 = hist4 + 0x100;
  uint64_t *hist6 = hist5 + 0x100;
  uint64_t *hist7 = hist6 + 0x100;
  uint64_t total = 0, sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
  uint64_t sum4 = 0, sum5 = 0, sum6 = 0 , sum7 = 0;
  size_t i;
  memset(hist, '\0', 8*0x100*sizeof(uint64_t));
  for(i=0;i<sz;i++){
    hist0[get_byte(in[i], 0)]++;
```

```
    hist1[get_byte(in[i], 1)]++;
    hist2[get_byte(in[i], 2)]++;
    hist3[get_byte(in[i], 3)]++;
    hist4[get_byte(in[i], 4)]++;
    hist5[get_byte(in[i], 5)]++;
    hist6[get_byte(in[i], 6)]++;
    hist7[get_byte(in[i], 7)]++;
  }
#define set_index(byte)                          \
  total = CAT(hist,byte)[i] + CAT(sum,byte);     \
  CAT(hist,byte)[i] = CAT(sum, byte);            \
  CAT(sum, byte) = total;

  for(i=0;i<0x100;i++){
    set_index(0);
    set_index(1);
    set_index(2);
    set_index(3);
    set_index(4);
    set_index(5);
    set_index(6);
    set_index(7);
  }

  uint64_t *temp = zmalloc(sz*sizeof(uint64_t));
  uint64_t *a = in, *b = temp;
  size_t pos = 0;
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 0);
    b[hist0[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 1);
    b[hist1[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 2);
    b[hist2[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 3);
    b[hist3[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 4);
    b[hist4[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 5);
    b[hist5[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 6);
    b[hist6[pos]++] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_byte(a[i], 7);
    b[hist7[pos]++] = a[i];
  }
  SWAP(a, b);
  free(temp);
//  return in;
```



```
}
void radix_sort_compact_u64(uint64_t *in, size_t sz){
  uint64_t hist[8][0x100];
  uint64_t total = 0, sum[8];
  size_t i, j;
  memset(hist, '\0', 8*0x100*sizeof(uint64_t));
  memset(sum, '\0', 8*sizeof(uint64_t));
  for(i=0;i<sz;i++){
    hist[0][get_byte(in[i], 0)]++;
    hist[1][get_byte(in[i], 1)]++;
    hist[2][get_byte(in[i], 2)]++;
    hist[3][get_byte(in[i], 3)]++;
    hist[4][get_byte(in[i], 4)]++;
    hist[5][get_byte(in[i], 5)]++;
    hist[6][get_byte(in[i], 6)]++;
    hist[7][get_byte(in[i], 7)]++;
  }
  for(i=0;i<0x100;i++){
    for(j=0;j<8;j++){
      total = hist[j][i] + sum[j];
      hist[j][i] = sum[j];
      sum[j] = total;
    }
  }

  uint64_t *temp = zmalloc(sz*sizeof(uint64_t));
  uint64_t *a = in, *b = temp;
  size_t pos = 0;
  for(j=0;j<8;j++){
    for(i=0;i<sz;i++){
      pos = get_byte(a[i], j);
      b[hist[j][pos]++] = a[i];
    }
    SWAP(a, b);
  }
  free(temp);
//  return in;
}
/*
  Radix sort to sort arbitary data based on integer keys.
*/
void radix_sort_keys(void **in, size_t sz, uint64_t(*get_key)(void*)){
  uint64_t hist[8][0x100];
  uint64_t total = 0, sum[8];
  size_t i, j;
  memset(hist, '\0', 8*0x100*sizeof(uint64_t));
  memset(sum, '\0', 8*sizeof(uint64_t));
  for(i=0;i<sz;i++){
    uint64_t key = get_key(in[i]);
    hist[0][get_byte(key, 0)]++;
    hist[1][get_byte(key, 1)]++;
    hist[2][get_byte(key, 2)]++;
    hist[3][get_byte(key, 3)]++;
    hist[4][get_byte(key, 4)]++;
    hist[5][get_byte(key, 5)]++;
    hist[6][get_byte(key, 6)]++;
    hist[7][get_byte(key, 7)]++;
  }
  for(i=0;i<0x100;i++){
    for(j=0;j<8;j++){
      total = hist[j][i] + sum[j];
      hist[j][i] = sum[j];
      sum[j] = total;
    }
  }

  void **temp = zmalloc(sz*sizeof(void*));
  void **a = in, **b = temp;
  size_t pos = 0;
```

```
  for(j=0;j<8;j++){
    for(i=0;i<sz;i++){
      uint64_t key = get_key(a[i]);
      pos = get_byte(key, j);
      b[hist[j][pos]++] = a[i];
    }
    SWAP(a, b);
  }
  free(temp);
//  return in;
}
/*
  same as above but using histograms of words instead of bytes. This will be
  faster for a large enough size (I'm not totally sure what the threshold is), b
ut
  it uses a lot more memory
*/
#define get_word(val, word)                     \
  (((val) >> ((2*word)*CHAR_BIT)) & 0xffffu)
void radix_sort_u16_u64(uint64_t *in, size_t sz){
  uint64_t *hist = alloca(4 * 0x10000 * sizeof(uint64_t));
  uint64_t *hist0 = hist;
  uint64_t *hist1 = hist0 + 0x10000;
  uint64_t *hist2 = hist1 + 0x10000;
  uint64_t *hist3 = hist2 + 0x10000;
  uint64_t total = 0, sum0 = 0, sum1 = 0, sum2 = 0, sum3 = 0;
  size_t i;
  memset(hist, '\0', 4*0x10000*sizeof(uint64_t));
  for(i=0;i<sz;i++){
    hist0[get_word(in[i], 0)]++;
    hist1[get_word(in[i], 1)]++;
    hist2[get_word(in[i], 2)]++;
    hist3[get_word(in[i], 3)]++;
  }
  for(i=0;i<0x10000;i++){
    total = hist0[i] + sum0;
    hist0[i] = sum0;
    sum0 = total;

    total = hist1[i] + sum1;
    hist1[i] = sum1;
    sum1 = total;

    total = hist2[i] + sum2;
    hist2[i] = sum2;
    sum2 = total;

    total = hist3[i] + sum3;
    hist3[i] = sum3;
    sum3 = total;
  }
  uint64_t *temp = zmalloc(sz*sizeof(uint64_t));
  uint64_t *a = in, *b = temp;
  size_t pos = 0;
  for(i=0;i<sz;i++){
    pos = get_word(a[i], 0);
    b[(hist0[pos]++)] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_word(a[i], 1);
    b[(hist1[pos]++)] = a[i];
  }
  SWAP(a, b);
  for(i=0;i<sz;i++){
    pos = get_word(a[i], 2);
    b[(hist2[pos]++)] = a[i];
  }
  SWAP(a, b);
```



```
  for(i=0;i<sz;i++){
    pos = get_word(a[i], 3);
    b[(hist3[pos]++)] = a[i];
  }
  free(temp);
//  return in;
}
void radix_sort_u64(uint64_t *in, size_t sz){
  radix_sort_u16_u64(in,sz);
}
```

```c
/**
 * \file sort_skele.c
 *
 * Sorts integers from a file using various algorithms.
 *
 * \author eaburns
 * \date 28-07-2010
 */

#include "C_util.h"

/* The various sorting algorithm names */
static const char *counting_sort_str = "counting";
static const char *radix_sort_str = "radix";
static const char *quick_sort_str = "quick";
static const char *insertion_sort_str = "insertion";
static const char *system_quick_sort_str = "system_quick";

/* Number of bits in an unsigned long. */
static const unsigned long ulong_bits = 8 * sizeof(unsigned long);

extern void qsort_u64(uint64_t *input, size_t len);
extern void radix_sort_u64(uint64_t *input, size_t len);
extern void insertion_sort_u64(uint64_t *input, size_t len);

/************************************************************
 * Functions that you must implement
 ************************************************************/

/*
 * Radix sort
 * Return 0 on success and 1 on failure.
 */
static unsigned int do_radix_sort(unsigned long ary[],
                                  unsigned long n, unsigned long nbits){
  radix_sort_u64(ary, n);
  return 0;
}


/*
 * Counting sort
 * Return 0 on success and 1 on failure.
 */
static unsigned int do_counting_sort(unsigned long ary[],
                                     unsigned long n, unsigned long nbits){
  if (nbits > 20){
    exit(EXIT_FAILURE);
  }
  int i, idx;
  int hist_size = sizeof(uint64_t) * (1<<nbits);
  uint64_t *hist = alloca(hist_size);
  memset(hist, '\0', hist_size);
  for(i=0;i<n;i++){
    hist[ary[i]]++;
  }
  //This is faster than doing a seperate loop to compute the indices
  //but makes the sort not stable, which is find for a counting sort
  //but not for a radix sort.
  for(i=0, idx=0;i<(1<<nbits) && idx<n;i++){
    while(hist[i]--){
      ary[idx++] = i;
    }
  }
  return 0;
}


/*
```



```c
 * Quicksort
 * Return 0 on success and 1 on failure.
 */
static unsigned int do_quicksort(unsigned long ary[],
                                 unsigned long n, unsigned long nbits){
  qsort_u64(ary, n);
  return 0;
}


/************************************************************
 *
 * You probably don't need to modify anything beyond here.
 *
 ************************************************************/


/************************************************************
 * Example: Using the standard library's qsort() routine.
 ************************************************************/

/* The comparison function for qsort(). */
static int compare(const void *_a, const void *_b)
{
  unsigned long *a = (unsigned long *) _a;
  unsigned long *b = (unsigned long *) _b;

  return *a - *b;
}


/*
 * Uses the standard library quicksort function.
 * Return 0 on success and 1 on failure.
 */
static unsigned int do_system_quicksort(unsigned long ary[],
                                        unsigned long n,
                                        unsigned long nbits)
{
  /* This is the system's quick sort function. */
  qsort(ary, n, sizeof(*ary), compare);

  return 0;
}


/************************************************************
 * Example: insertion sort
 ************************************************************/

/*
 * Insertion sort.
 * Return 0 on success and 1 on failure.
 */
static unsigned int do_insertion_sort(unsigned long ary[], unsigned long n,
                                      unsigned long nbits)
{
  unsigned long j;

  for (j = 1; j < n; j += 1) {
    unsigned long key = ary[j];
    unsigned long i = j;
    while (i > 0 && ary[i-1] > key) {
      ary[i] = ary[i-1];
      i -= 1;
    }
    ary[i] = key;
  }
```

```
  return 0;
}

/*
 * Read the header from the input file and returts the number of
 * values in the input using the 'nvalues' argument and the number of
 * bits for each number using the 'nbits' argument.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int read_file_header(FILE * infile, unsigned long *nvalues,
                                     unsigned long *nbits)
{
  int ret;
  unsigned long _nvalues, _nbits;

  ret = fscanf(infile, " %lu %lu", &_nvalues, &_nbits);
  if (ret == EOF) {
    fprintf(stderr, "Unexpected end of file\n");
    return 1;
  }
  if (ret != 2) {
    fprintf(stderr, "Malformed file header\n");
    return 1;
  }

  if (_nbits > ulong_bits) {
    fprintf(stderr, "%lu bits input values are too big\n", _nbits);
    fprintf(stderr, "Word size seems to be %lu bits\n", ulong_bits);
    return 1;
  }

  if (nvalues)
    *nvalues = _nvalues;
  if (nbits)
    *nbits = _nbits;

  return 0;
}

/*
 * Reads the next number from the input file into the 'num' argument.
 * If the end of the file is reached then 'num' is left as is and
 * 'eof' is set to 1, otherwise 'eof' is set to zero.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int read_next_number(FILE * infile, unsigned long *num,
                                     unsigned int *eof)
{
  int ret;
  unsigned long _num;

  ret = fscanf(infile, " %lu", &_num);
  if (ret == EOF) {
    *eof = 1;
    return 0;
  }
  if (ret != 1) {
    perror("fscanf failed");
    return 1;
  }

  *num = _num;
  *eof = 0;

  return 0;
}
```



```
/*
 * Reads 'n' numbers from the given input file into the provided
 * array.
 *
 * Returns 0 on success and 1 on failure.  The state of 'ary' on
 * failure is unspecified.
 */
static unsigned int read_into_array(FILE * infile, unsigned long n,
                                    unsigned long ary[])
{
  unsigned long i;

  for (i = 0; i < n; i += 1) {
    unsigned int err, eof;
    err = read_next_number(infile, &ary[i], &eof);
    if (err)
      return 1;
    if (eof) {
      fprintf(stderr, "Unexpected EOF when reading %lu values", n);
      return 1;
    }
  }

  return 0;
}

/* Writes the given number to the output file. */
static void output_number(FILE * outfile, unsigned long num)
{
  fprintf(outfile, "%lu\n", num);
}

/* Output the given array to the output file. */
static void output_from_array(FILE * outfile, unsigned long ary[],
                              unsigned long n)
{
  unsigned long i;

  for (i = 0; i < n; i += 1)
    output_number(outfile, ary[i]);
}

/* Print the usage string */
static void usage(void)
{
  fprintf(stderr, "usage: sort <algorithm> <infile> <outfile>\n");
  fprintf(stderr,
              "Where <algorithm> is one of: counting, radix, quick,\n"
              "                 insertion or system_quick\n"
              "and <infile> and/or <outfile> may be '-' to indicate that\n"
          "the standard input and/or output stream should be used\n");
}


/*
 * Reads the file header and the values.  The return value is an array
 * of values that must be freed by the caller.
 */
static unsigned long *get_values(FILE * infile, unsigned long *n,
                                 unsigned long *nbits)
{
  unsigned int err;
  unsigned long _n, _nbits;
  unsigned long *ary;

  err = read_file_header(infile, &_n, &_nbits);
  if (err)
    return NULL;
```

```
  ary = malloc(sizeof(*ary) * _n);
  if (!ary) {
    perror("Failed to allocate array");
    return NULL;
  }

  err = read_into_array(infile, _n, ary);
  if (err) {
    free(ary);
    return NULL;
  }

  if (n)
    *n = _n;
  if (nbits)
    *nbits = _nbits;

  return ary;
}

/* Gets the time of day in seconds. */
static double get_current_seconds(void)
{
  double sec, usec;
  struct timeval tv;

  if (gettimeofday(&tv, NULL) < 0) {
    perror("gettimeofday failed");
    exit(EXIT_FAILURE);
  }

  sec = tv.tv_sec;
  usec = tv.tv_usec;

  return sec + (usec / 1000000);
}

/*
 * Reads the values, begins the timer, calls the sorting algorithm,
 * stops the timer and outputs the values.  The time taken is printed
 * to standard error.
 */
static unsigned int do_sort(const char *const algorithm, FILE * infile,
                            FILE * outfile)
{
  int err = 0;
  double start, end;
  unsigned long n, nbits;
  unsigned long *ary;

  ary = get_values(infile, &n, &nbits);
  if (!ary)
    return 1;

  start = get_current_seconds();

  if (strcmp(algorithm, counting_sort_str) == 0) {
    err = do_counting_sort(ary, n, nbits);

  } else if (strcmp(algorithm, radix_sort_str) == 0) {
    err = do_radix_sort(ary, n, nbits);

  } else if (strcmp(algorithm, quick_sort_str) == 0) {
    err = do_quicksort(ary, n, nbits);

  } else if (strcmp(algorithm, insertion_sort_str) == 0) {
    err = do_insertion_sort(ary, n, nbits);

  } else if (strcmp(algorithm, system_quick_sort_str) == 0) {
```



```
    err = do_system_quicksort(ary, n, nbits);

  } else {
    fprintf(stderr, "Impossible\n");
    exit(EXIT_FAILURE);
  }

  end = get_current_seconds();

  output_from_array(outfile, ary, n);
  fprintf(stderr, "%f\n", end - start);

  free(ary);

  return err;
}


int main(int argc, char *const argv[])
{
  int ret = EXIT_SUCCESS;
  unsigned int err;
  FILE *infile = stdin, *outfile = stdout;

  if (argc < 4 || (strcmp(argv[1], counting_sort_str) != 0
                   && strcmp(argv[1], radix_sort_str) != 0
                   && strcmp(argv[1], quick_sort_str) != 0
                   && strcmp(argv[1], insertion_sort_str) != 0
                   && strcmp(argv[1], system_quick_sort_str) != 0)) {
    usage();
    return EXIT_FAILURE;
  }


  if (strcmp(argv[2], "-") != 0) {
    infile = fopen(argv[2], "r");
    if (!infile) {
      perror("Failed to open input file for reading");
      ret = EXIT_FAILURE;
      goto out;
    }
  }

  if (strcmp(argv[3], "-") != 0) {
    outfile = fopen(argv[3], "w");
    if (!outfile) {
      perror("Failed to open output file for writing");
      ret = EXIT_FAILURE;
      goto out;
    }
  }

  err = do_sort(argv[1], infile, outfile);
  if (err)
    ret = EXIT_FAILURE;

 out:

  if (outfile && outfile != stdout)
    fclose(outfile);
  if (infile && infile != stdin)
    fclose(infile);

  return ret;
}
```