```
/**
 * \file add.c
 *
 * Add floating point numbers read from standard input and output the
 * sum on standard output and the amount of time required to sum the
 * numbers to standard error.
 *
 * \author eaburns
 * \date 30-07-2010
 */

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sys/time.h>

const char *const seq_str = "seq";
const char *const sort_str = "sort";
const char *const min2_scan_str = "min2_scan";
const char *const heap_str = "heap";

void pullup(double[], unsigned int);
void pushdown(double[], unsigned int, unsigned int);
void print_heap(double[], unsigned int);
unsigned int get_smallest(double[], unsigned int, unsigned int, unsigned int);
unsigned int getp(unsigned int);
unsigned int getl(unsigned int);
unsigned int getr(unsigned int);
double pop(double[], unsigned int *);
void push(double[], double, unsigned int *);

/*
 * You need to implement this function.  It should insert the numbers
 * from 'ary' into a heap.  Until the heap has only a single element
 * remaining, pull off the two smallest numbers add them and push
 * their sum.
 *
 * Put the final result in the location pointed to by 'result'.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int do_heap_sum(double ary[], unsigned int n,
                                double *result)
{
        int i = 0;
        double num1, num2, sum = 0;

        unsigned int mid = (n/2) - 1;

        /* create heap */
        for(i = mid; i >=0; i--){
                pushdown(ary, i, n);
        }

        /*fprintf(stderr, "heap created!\n");
        print_heap(ary, n);*/


        while(n > 1){
                num1 = pop(ary, &n);
                num2 = pop(ary, &n);
                sum = num1 + num2;
                push(ary, sum, &n);
        }

        *result = ary[0];
```



```
    return 0;
}

double pop(double ary[], unsigned int * n){
        double num = 0;
        num = ary[0];
        ary[0] = ary[*n-1];
        ary[*n-1] = -1;
        *n = *n - 1;
        pushdown(ary, 0, *n);
        return num;
}

void push(double ary[], double val, unsigned int * n){
        ary[*n] = val;
        pullup(ary, *n);
        *n = *n + 1;
}

void pullup(double ary[], unsigned int index){
        unsigned int pIndex = 0;
        double temp = 0;

        /* stop if at the root */
        if(index == 0){
                return;
        }

        /* swap if value is less than parent */
        pIndex = getp(index);
        if ( ary[index] < ary[pIndex] ){
                temp = ary[index];
                ary[index] = ary[pIndex];
                ary[pIndex] = temp;
                pullup(ary, pIndex);
        }
}

void pushdown(double ary[], unsigned int index, unsigned int n){
        unsigned int lIndex, rIndex, min = 0;
        double temp = 0;

        lIndex = getl(index);
        rIndex = getr(index);

        /* stop if my children go off the array
         * If they don't, find the smallest index */
        if((lIndex >= n && rIndex >= n)){
                return;
        }
        else if(lIndex >= n){
                if(index < rIndex)
                        min = index;
                else
                        min = rIndex;
        }
        else if(rIndex >= n){
                if(index < lIndex)
                        min = index;
                else
                        min = lIndex;
        }
        else{
                min = get_smallest(ary, index, lIndex, rIndex);
        }
        /* swap if a child is smaller than index */
        if ( min != index ){
                temp = ary[index];
                ary[index] = ary[min];
```

```
                ary[min] = temp;
                pushdown(ary, min, n);
        }
}

unsigned int get_smallest(double ary[], unsigned int a,
                unsigned int b, unsigned int c){
        if(ary[a] < ary[b] && ary[a] < ary[c]){
                return a;
        }
        else if(ary[b] < ary[a] && ary[b] < ary[c]){
                return b;
        }
        else if(ary[c] < ary[b] && ary[c] < ary[a]){
                return c;
        }

        /* theoretically unreachable */
        return 0;
}


unsigned int getp(unsigned int index){
        return (index - 1) / 2;
}

unsigned int getl(unsigned int index){
        return (index * 2) + 1;
}

unsigned int getr(unsigned int index){
        return (index * 2) + 2;
}

void print_heap(double ary[], unsigned int n){
        unsigned int i, l, r = 0;
        fprintf(stderr, "***************PRINTING HEAP!************\n");
        for(i = 0; i < n; i++){
                l = getl(i);
                r = getr(i);
                fprintf(stderr, "parent: %f\n", ary[i]);
                if(l < n)
                        fprintf(stderr, "left child: %f\n", ary[getl(i)]);
                if(r < n)
                        fprintf(stderr, "right child: %f\n", ary[getr(i)]);
        }
}


/*************************************************************
 * Examples
 *************************************************************/

/*
 * An example algorithm that sequentially sums the values in the order
 * they are given.
 */
static unsigned int do_seq_sum(double ary[], unsigned int n,
                               double *result)
{
    unsigned int i;
    double sum = 0;

    for (i = 0; i < n; i += 1)
        sum += ary[i];

    *result = sum;

    return 0;
```



```
}

/* Comparison function for qsort(). */
static int compare(const void *a, const void *b)
{
    return *(double *) a - *(double *) b;
}


/*
 * An example algorithm that sorts the numbers in ascending order and
 * adds them.
 */
static unsigned int do_sorted_sum(double ary[], unsigned int n,
                                  double *result)
{
    qsort(ary, n, sizeof(ary[0]), compare);

    return do_seq_sum(ary, n, result);
}

/* Swaps two numbers. */
static void swap(unsigned int *a, unsigned int *b)
{
    unsigned int tmp = *a;
    *a = *b;
    *b = tmp;
}

/* Scans the array for the indices of the two minimum values. */
static void find_min_two(double ary[], unsigned int n, unsigned int *min1,
                         unsigned int *min2)
{
    unsigned int i;
    unsigned int m1, m2;

    assert(n > 1);

    m1 = 0;
    m2 = 1;
    if (ary[m2] < ary[m1])
        swap(&m1, &m2);

    for (i = 2; i < n; i += 1) {
        if (ary[i] < ary[m2]) {
                m2 = i;
                if (ary[m2] < ary[m1])
                        swap(&m1, &m2);
        }
    }

    if (min1)
        *min1 = m1;
    if (min2)
        *min2 = m2;
}

/*
 * An example algorithm that continually scans the numbers looking for
 * the minimum two and then sums them.
 */
static unsigned int do_min_two_sum(double ary[], unsigned int n,
                                   double *result)
{
    unsigned int min1, min2;

    while (n > 1) {
                double vl1, vl2;
```

```
                find_min_two(ary, n, &min1, &min2);
                if (min1 < min2)
                        swap(&min1, &min2);
                vl1 = ary[min1];
                vl2 = ary[min2];
                ary[min1] = ary[n - 1];
                ary[min2] = ary[n - 2];
                ary[n - 2] = vl1 + vl2;
                n -= 1;
    }

    *result = ary[0];

    return 0;
}

/************************************************************
 * Do not edit the functions below here.
 ************************************************************/

/*
 * Reads 'n' doubles into 'ary'.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int read_doubles(FILE * infile, double ary[],
                                 unsigned int n)
{
    unsigned int i;

    for (i = 0; i < n; i += 1) {
                int ret = fscanf(infile, " %lf", &ary[i]);
                if (ret == EOF) {
                        fprintf(stderr, "Unexpected end of file\n");
                        return 1;
                }
                if (ret != 1) {
                        fprintf(stderr, "Failed to scan a float\n");
                        return 1;
                }
    }

    return 0;
}


/*
 * Reads the header from the input file where the header is the number of values
 that will be following.
 *
 * Return 0 on success and 1 on failuer.
 */
static unsigned int read_number_of_values(FILE * infile, unsigned int *n)
{
    int ret;
    unsigned int _n;

    ret = fscanf(stdin, "%u\n", &_n);
    if (ret == EOF) {
                fprintf(stderr, "Unexpected end of file\n");
                return 1;
    }
    if (ret != 1) {
                fprintf(stderr, "Failed to read number of values to add\n");
                return 1;
    }

    if (n)
        *n = _n;
```



```
    return 0;
}

/* Print the usage string and exit. */
static void usage(void)
{
    fprintf(stderr, "Usage:\n");
    fprintf(stderr,
            "add <algorithm>\n"
            "where <algorithm> is one of: seq, sort, min2_scan or heap\n");
    exit(EXIT_FAILURE);
}


/* 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);
}


/*
 * Dispatches the given algorithm to compute the sum of the values in
 * the array.  If the algorithm name is unknown then the usage string
 * is printed and the program exits.
 *
 * Returns 0 on success and 1 on failure.
 */
static unsigned int do_sum(FILE * outfile, const char *const alg,
                           double ary[], unsigned int n)
{
    double result = -1;
    double start_time, end_time;
    unsigned int err = 0;

    start_time = get_current_seconds();

    if (strcmp(alg, seq_str) == 0)
        err = do_seq_sum(ary, n, &result);
    else if (strcmp(alg, sort_str) == 0)
        err = do_sorted_sum(ary, n, &result);
    else if (strcmp(alg, min2_scan_str) == 0)
        err = do_min_two_sum(ary, n, &result);
    else if (strcmp(alg, heap_str) == 0)
        err = do_heap_sum(ary, n, &result);
    else
        usage();

    if (!err) {
        end_time = get_current_seconds();
        fprintf(stderr, "%f\n", end_time - start_time);
        fprintf(outfile, "%f\n", result);
    }

    return err;
}
```

```
int main(int argc, char *const argv[])
{
    int ret = EXIT_SUCCESS;
    unsigned int err;
    unsigned int n;
    double *ary;

    if (argc < 2)
        usage();

    err = read_number_of_values(stdin, &n);
    if (err)
        return EXIT_SUCCESS;

    ary = malloc(n * sizeof(*ary));
    if (!ary) {
        perror("malloc failed");
        goto out;
    }

    err = read_doubles(stdin, ary, n);
    if (err)
        goto out;

    err = do_sum(stdout, argv[1], ary, n);
    if (err)
        goto out;

  out:
    if (ary)
        free(ary);

    return ret;
}
```