```
================================================================================
            /local/submit/submit/comp10002/ass1/jlafontaine/src/ass1.c
================================================================================

/* Program to do "calculations" on numeric CSV data files.

   Skeleton program written by Alistair Moffat, ammoffat@unimelb.edu.au,
   September 2020, with the intention that it be modified by students
   to add functionality, as required by the assignment specification.

   Student Authorship Declaration:

   (1) I certify that except for the code provided in the initial skeleton
   file, the  program contained in this submission is completely my own
   individual work, except where explicitly noted by further comments that
   provide details otherwise.  I understand that work that has been developed
   by another student, or by me in collaboration with other students, or by
   non-students as a result of request, solicitation, or payment, may not be
   submitted for assessment in this subject.  I understand that submitting for
   assessment work developed by or in collaboration with other students or
   non-students constitutes Academic Misconduct, and may be penalized by mark
   deductions, or by other penalties determined via the University of
   Melbourne Academic Honesty Policy, as described at
   https://academicintegrity.unimelb.edu.au.

   (2) I also certify that I have not provided a copy of this work in either
   softcopy or hardcopy or any other form to any other student, and nor will I
   do so until after the marks are released. I understand that providing my
   work to other students, regardless of my intention or any undertakings made
   to me by that other student, is also Academic Misconduct.

   (3) I further understand that providing a copy of the assignment
   specification to any form of code authoring or assignment tutoring service,
   or drawing the attention of others to such services and code that may have
   been made available via such a service, may be regarded as Student General
   Misconduct (interfering with the teaching activities of the University
   and/or inciting others to commit Academic Misconduct).  I understand that
   an allegation of Student General Misconduct may arise regardless of whether
   or not I personally make use of such solutions or sought benefit from such
   actions.

   Signed by: JAMES LA FONTAINE   1079860
   Dated:     26/09/2020

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <ctype.h>
#include <math.h>
#include <assert.h>

/* these #defines provided as part of the initial skeleton */

#define MAXCOLS 20  /* maximum number of columns to be handled */
#define MAXROWS 999 /* maximum number of rows to be handled */
#define LABLEN  20  /* maximum length of each column header */
#define LINELEN 100 /* maximum length of command lines */

#define ERROR   (-1)    /* error return value from some functions */

#define O_NOC   '-' /* the "do nothing" command */
#define O_IND   'i' /* the "index" command */
#define O_ANA   'a' /* the "analyze" command */
#define O_DPY   'd' /* the "display" command */
#define O_PLT   'p' /* the "plot" command */
#define O_SRT   's' /* the "sort" command */

#define CH_COMMA ','    /* comma character */
#define CH_CR    '\r'   /* pesky CR character in DOS-format files */
#define CH_NL    '\n'   /* newline character */
```

```
/* if you wish to add further #defines, put them below this comment */

#define DEBUG 0
#if DEBUG
#define DUMP_DBL(x) printf("line %d: %s = %.6f\n", __LINE__, #x, x)
#define DUMP_INT(x) printf("line %d: %s = %d\n", __LINE__, #x, x)
#define DUMP_STR(x) printf("line %d: %s = %s\n", __LINE__, #x, x)
#else
#define DUMP_DBL(x)
#define DUMP_INT(x)
#define DUMP_STR(x)
#endif

#define L_THAN (-1)             /* returned by cmp when n1 is less than n2 */
#define EQUAL 0                  /* returned by cmp when n1 is equal to n2 */
#define G_THAN 1              /* returned by cmp when n1 is greater than n2 */

#define FIND_MIN -1                             /* key for find_min_or_max() */
#define FIND_MAX 1                              /* key for find_min_or_max() */

#define IS_SORTED 1        /* returned by is_sorted when a column is sorted */
#define NOT_SORTED 0    /* returned by is_sorted when a column isn't sorted */

#define EQUAL_RANGE pow(10, -6)      /* numbers are equal within this range */

#define D_SPACES 8    /* the multiple of spaces before a heading in display */

#define ONE_INSTANCE 1      /* one instance of a consecutive row in display */

#define ONE_SWAP 1       /* dumped when debugging to track when swaps occur */

#define INTERVALS 10                    /* number of intervals in histogram */
#define ENDPOINTS 11           /* number of interval endpoints in histogram */
#define ZERO_FREQUENCY 0  /* initalise the frequency array elements to zero */
#define BASE_SCALE 1            /* base integer scaling factor of histogram */

#define HISTOGRAM_WIDTH 60                            /* max histogram width */

#define DOUBLE_CONVERSION 1.0  /* multiply int by this to convert to double */


/* and then, here are some types for you to work with */
typedef char head_t[LABLEN+1];

typedef double csv_t[MAXROWS][MAXCOLS];

typedef double freq_t[INTERVALS][MAXCOLS];

/*************************************************************/

/* function prototypes */

void get_csv_data(csv_t D, head_t H[],  int *dr,  int *dc, int argc,
    char *argv[]);
void error_and_exit(char *msg);
void print_prompt(void);
int  get_command(int dc, int *command, int ccols[], int *nccols);
void handle_command(int command, int ccols[], int nccols,
            csv_t D, head_t H[], int dr, int dc);
void do_index(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols);

/* add further function prototypes below here */

int cmp(double n1, double n2);

void do_analyze(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols);
double find_min_or_max(csv_t D, int dr, int c, int key);
double find_avg(csv_t D, int dr, int c);
double find_med(csv_t D, int dr, int c);
int is_sorted(csv_t D, int dr, int c);

void do_display(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols);
int find_instances(csv_t D, int curr_row, int dr, int ccols[], int nccols);
```

```
void do_sort(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols);
void insertionsort(csv_t D, int dr, int dc, int ccols[], int nccols);
void tiebreak(csv_t D, int curr_row, int dc, int ccols[], int nccols);
void row_swap(csv_t D, int curr_row, int dc);

void do_plot(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols);
void plot_histogram(csv_t D, int dr, int ccols[], int nccols);
void find_endpoints(csv_t D, double E[], int dr, int ccols[], int nccols);
void create_frequency_matrix(csv_t D, freq_t F, double E[], int *scal_fact,
                             int dr, int ccols[], int nccols);

/*************************************************************/

/* main program controls all the action
*/
int
main(int argc, char *argv[]) {

    head_t H[MAXCOLS];  /* labels from the first row in csv file */
    csv_t D;        /* the csv data stored in a 2d matrix */
    int dr=0, dc=0;     /* number of rows and columns in csv file */
    int ccols[MAXCOLS];
    int nccols;
    int command;

    /* this next is a bit of magic code that you can ignore for
       now, it reads csv data from a file named on the
       commandline and saves it to D, H, dr, and dc
       */
    get_csv_data(D, H, &dr, &dc, argc, argv);

    /* ok, all the input data has been read, safe now to start
       processing commands against it */

    print_prompt();
    while (get_command(dc, &command, ccols, &nccols) != EOF) {
        handle_command(command, ccols, nccols,
            D, H, dr, dc);
        print_prompt();
    }

    /* all done, so pack up bat and ball and head home */
    printf("\nTa daa!!!\n");
    return 0;
}

/*************************************************************/

/* prints the prompt indicating ready for input
*/
void
print_prompt(void) {
    printf("> ");
}

/*************************************************************/

/* read a line of input into the array passed as argument
   returns false if there is no input available
   all whitespace characters are removed
   all arguments are checked for validity
   if no argumnets, the numbers 0..dc-1 are put into the array
*/
int
get_command(int dc, int *command, int columns[], int *nccols) {
    int i=0, c, col=0;
    char line[LINELEN];
    /* comand is in first character position */
    if ((*command=getchar()) == EOF) {
        return EOF;
    }
    /* and now collect the rest of the line, integer by integer,
       sometimes in C you just have to do things the hard way */
    while (((c=getchar())!=EOF) && (c!='\n')) {
```

```
            if (isdigit(c)) {
                /* digit contributes to a number */
                line[i++] = c;
            } else if (i!=0)  {
                /* reached end of a number */
                line[i] = '\0';
                columns[col++] = atoi(line);
                /* reset, to collect next number */
                i = 0;
            } else {
                /* just discard it */
            }
        }
        if (i>0) {
            /* reached end of the final number in input line */
            line[i] = '\0';
            columns[col++] = atoi(line);
        }

        if (col==0) {
            /* no column numbers were provided, so generate them */
            for (i=0; i<dc; i++) {
                columns[i] = i;
            }
            *nccols = dc;
            return !EOF;
        }

        /* otherwise, check the one sthat were typed against dc,
           the number of cols in the CSV data that was read */
        for (i=0; i<col; i++) {
            if (columns[i]<0 || columns[i]>=dc) {
                printf("%d is not between 0 and %d\n",
                    columns[i], dc);
                /* and change to "do nothing" command */
                *command = O_NOC;
            }
        }
        /* all good */
        *nccols = col;
        return !EOF;
    }

    /*****************************************************************/

    /* this next is a bit of magic code that you can ignore for now
       and that will be covered later in the semester; it reads the
       input csv data from a file named on the commandline and saves
       it into an array of character strings (first line), and into a
       matrix of doubles (all other lines), using the types defined
       at the top of the program.  If you really do want to understand
       what is happening, you need to look at:
       -- The end of Chapter 7 for use of argc and argv
       -- Chapter 11 for file operations fopen(), and etc
    */
    void
    get_csv_data(csv_t D, head_t H[],  int *dr,  int *dc, int argc,
            char *argv[]) {
        FILE *fp;
        int rows=0, cols=0, c, len;
        double num;

        if (argc<2) {
            /* no filename specified */
            error_and_exit("no CSV file named on commandline");
        }
        if (argc>2) {
            /* confusion on command line */
            error_and_exit("too many arguments supplied");
        }
        if ((fp=fopen(argv[1], "r")) == NULL) {
            error_and_exit("cannot open CSV file");
        }

```

```
        /* ok, file exists and can be read, next up, first input
           line will be all the headings, need to read them as
           characters and build up the corresponding strings */
        len = 0;
        while ((c=fgetc(fp))!=EOF && (c!=CH_CR) && (c!=CH_NL)) {
            /* process one input character at a time */
            if (c==CH_COMMA) {
                /* previous heading is ended, close it off */
                H[cols][len] = '\0';
                /* and start a new heading */
                cols += 1;
                len = 0;
            } else {
                /* store the character */
                if (len==LABLEN) {
                    error_and_exit("a csv heading is too long");
                }
                H[cols][len] = c;
                len++;
            }
        }
        /* and don't forget to close off the last string */
        H[cols][len] = '\0';
        *dc = cols+1;

        /* now to read all of the numbers in, assumption is that the input
           data is properly formatted and error-free, and that every row
           of data has a numeric value provided for every column */
        rows = cols = 0;
        while (fscanf(fp, "%lf", &num) == 1) {
            /* read a number, put it into the matrix */
            if (cols==*dc) {
                /* but first need to start a new row */
                cols = 0;
                rows += 1;
            }
            /* now ok to do the actual assignment... */
            D[rows][cols] = num;
            cols++;
            /* and consume the comma (or newline) that comes straight
               after the number that was just read */
            fgetc(fp);
        }
        /* should be at last column of a row */
        if (cols != *dc) {
            error_and_exit("missing values in input");
        }
        /* and that's it, just a bit of tidying up required now  */
        *dr = rows+1;
        fclose(fp);
        printf("  csv data loaded from %s", argv[1]);
        printf(" (%d rows by %d cols)\n", *dr, *dc);
        return;
    }

    /*****************************************************************/

    void
    error_and_exit(char *msg) {
        printf("Error: %s\n", msg);
        exit(EXIT_FAILURE);
    }

    /*****************************************************************/

    /* the 'i' index command
    */
    void
    do_index(csv_t D, head_t H[], int dr, int dc,
            int ccols[], int nccols) {
        int i, c;
        printf("\n");
        for (i=0; i<nccols; i++) {
            c = ccols[i];
```

```
370         printf("  column %2d: %s\n", c, H[c]);

            DUMP_INT(ccols[i]);
            DUMP_INT(nccols);
        }
375 }


    /*****************************************************************
    *****************************************************************

380 
    Below here is where you do most of your work, and it shouldn't be
    necessary for you to make any major changes above this point (except
    for function prototypes, and perhaps some new #defines).

385 Below this point you need to write new functions that provide the
    required functionality, and modify function handle_command() as you
    write (and test!) each one.

    Tackle the stages one by one and you'll get there.

390 
    Have Fun!!!

    *****************************************************************
    *****************************************************************/

395 
    /* this function examines each incoming command and decides what
       to do with it, kind of traffic control, deciding what gets
       called for each command, and which of the arguments it gets
    */
400 void
    handle_command(int command, int ccols[], int nccols,
                csv_t D, head_t H[], int dr, int dc) {
        if (command==O_NOC) {
            /* the null command, just do nothing */
405     } else if (command==O_IND) {
            do_index(D, H, dr, dc, ccols, nccols);
        } else if (command==O_ANA) {
            do_analyze(D, H, dr, dc, ccols, nccols);
        } else if (command==O_DPY) {
410         do_display(D, H, dr, dc, ccols, nccols);
        } else if (command==O_SRT) {
            do_sort(D, H, dr, dc, ccols, nccols);
        } else if (command==O_PLT) {
            do_plot(D, H, dr, dc, ccols, nccols);
415     /* and now a last option for things that aren't known */
        } else {
            printf("command '%c' is not recognized"
                " or not implemented yet\n", command);
        }
420     return;
    }

    /*****************************************************************/

425 /* compares two doubles in the CSV data array and returns a value indicating
       if the first value is less than, equal to, or greater than the second value
       (inspired by cmp function in binarysearch.c written by Alistair Moffat)
    */
    int
430 cmp(double n1, double n2) {

        if (n1 <= n2 - EQUAL_RANGE) {
            return L_THAN;
        } else if (n1 < n2 + EQUAL_RANGE) {
435         return EQUAL;
        } else {
            return G_THAN;
        }
    }
440 
    /*****************************************************************/

    /* the 'a' analyze command
```

```
 */
void
do_analyze(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols) {
    int i, c;

    /* print a newline to move below the prompt line */
    printf("\n");

    /* print the stats for each of the specified columns */
    for (i=0; i<nccols; i++) {
        c = ccols[i];
        if (is_sorted(D, dr, c)) {
            printf("%17s (sorted)\n", H[c]);
        } else {
            printf("%17s\n", H[c]);
        }
        printf("  max = %7.1f\n", find_min_or_max(D, dr, c, FIND_MAX));
        printf("  min = %7.1f\n", find_min_or_max(D, dr, c, FIND_MIN));
        printf("  avg = %7.1f\n", find_avg(D, dr, c));
        if (is_sorted(D, dr, c)) {
            printf("  med = %7.1f\n", find_med(D, dr, c));
        }
        /* print a newline after each set of column stats except the last one
        */
        if (i<nccols-1) {
            printf("\n");
        }
    }
}

/*****************************************************************/

/* finds the lowest or highest number in the specified column of D depending
   on the key that is input
*/
double
find_min_or_max(csv_t D, int dr, int c, int key) {
    int i;
    double min_or_max=D[0][c];

    for (i=1; i<dr; i++) {
        /* find the min if the FIND_MIN key is input */
        if (key == FIND_MIN) {
            if (D[i][c] < min_or_max) {
                min_or_max = D[i][c];
            }
        /* otherwise we must be finding the max */
        } else {
            if (D[i][c] > min_or_max) {
                min_or_max = D[i][c];
            }
        }
    }
    return min_or_max;
}

/*****************************************************************/

/* finds the average of the specified column of D
*/
double
find_avg(csv_t D, int dr, int c) {
    int i, totnums=0;
    double sum=0, avg;

    for (i=0; i<dr; i++) {
        sum += D[i][c];
        totnums += 1;
    }
    avg = sum/totnums;
    return avg;
}

/*****************************************************************/
```

```
    /* finds the median of the specified column of D
520 */
    double
    find_med(csv_t D, int dr, int c) {
        double med;

525     /* check if number of rows is odd or even before calculating median */
        if (dr%2==0) {
            med = (D[(dr/2)-1][c] + D[(dr/2)][c])/2;

            DUMP_INT((dr/2)-1);
530         DUMP_INT(dr/2);
        } else {
            med = D[(dr/2)][c];

            DUMP_INT(dr/2);
535     }
        return med;
    }

    /*****************************************************************/
540
    /* checks if the specified column of D is sorted in ascending order
    */
    int
    is_sorted(csv_t D, int dr, int c) {
545     int i;

        /* check if any element is bigger than its subsequent element */
        for (i=0; i<dr-1; i++) {
            if (cmp(D[i][c],D[i+1][c]) == G_THAN) {
550             return NOT_SORTED;
            }
        }
        return IS_SORTED;
    }
555
    /*****************************************************************/

    /* the 'd' display command
    */
560 void
    do_display(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols) {
        int i, j, c, spaces, instances;

        /* print a newline to move below the prompt line */
565     printf("\n");

        /* print the headings with the appropriate spacing */
        for (i=nccols-1; i>=0; i--) {
            c = ccols[i];
570         spaces = (i+1)*D_SPACES;
            printf("%*s\n", spaces, H[c]);
        }

        /* print out each row as specified after finding out the number of
575        instances */
        for (i=0; i<dr ; i+=instances) {
            instances = find_instances(D, i, dr, ccols, nccols);
            for (j=0; j<nccols ;j++) {
                c = ccols[j];
580             printf(" %7.1f", D[i][c]);
            }
            if (instances==ONE_INSTANCE) {
                printf("  ( 1 instance)");
            } else {
585             printf("  (%2d instances)", instances);
            }
                printf("\n");
        }
    }
590
    /*****************************************************************/
```

```
/* finds the number of consecutive equal rows based on the specified columns
*/
int
find_instances(csv_t D, int curr_row, int dr, int ccols[], int nccols) {
    int i, j, c, instances=ONE_INSTANCE;

    for (j=curr_row; j<dr ; j++) {
        for (i=0; i<nccols ;i++) {
            c = ccols[i];
            /* if any elements aren't equal then we have found the amount of
               consecutive instances of the current row and can return */
            if (cmp(D[j][c],D[j+1][c]) != EQUAL) {
                return instances;
            }
        }
        instances += 1;
    }
    return instances;
}

/*****************************************************************/

/* the 's' sort command
*/
void
do_sort(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols) {
    int i, c;

    insertionsort(D, dr, dc, ccols, nccols);

    /* print the confirmation message that indicates the type of sorting and
       the completion of sorting */
    printf("\n   sorted by: ");
    for (i=0; i<nccols; i++) {
        c = ccols[i];
        if (i<nccols-1) {
            printf("%s, ", H[c]);
        } else {
            printf("%s\n", H[c]);
        }
    }
}

/*****************************************************************/

/* insertion sort by specified columns into ascending order
   (adapted from sort_int_array in insertionsort.c written by Alistair Moffat)
*/
void
insertionsort(csv_t D, int dr, int dc, int ccols[], int nccols) {
    int i, j, pc=ccols[0];    /* pc is the primary key column for sorting */

    for (i=1; i<dr; i++) {
        for (j=i-1; j>=0 && cmp(D[j+1][pc],D[j][pc])<=EQUAL; j--) {
            /* break ties if the row values in the primary column are tied */
            if (cmp(D[j+1][pc],D[j][pc]) == EQUAL) {

                DUMP_DBL(D[j+1][pc]);
                DUMP_DBL(D[j][pc]);

                tiebreak(D, j, dc, ccols, nccols);
            /* otherwise the next row value must be less than the current row
               value so perform a row swap */
            } else {
                row_swap(D, j, dc);
            }
        }
    }
}

/*****************************************************************/

/* breaks ties between rows being compared during insertion sort by using
```

```
   the secondary specified columns
*/
void
tiebreak(csv_t D, int curr_row, int dc, int ccols[], int nccols) {
    int i, c, next_row=curr_row+1;

    /* compare the value between tied rows in the secondary columns until
       the tie is broken or all columns have been checked */
    for (i=1; i<nccols; i++) {
        c = ccols[i];

        DUMP_DBL(D[next_row][c]);
        DUMP_DBL(D[curr_row][c]);

        /* the current row wins the tie and so the rows will be swapped */
        if (cmp(D[next_row][c],D[curr_row][c])==L_THAN) {
            row_swap(D, curr_row, dc);
            return;
        /* the subsequent row wins the tie and no row swap must be performed */
        } else if (cmp(D[next_row][c],D[curr_row][c])==G_THAN) {
            return;
        }
    }
}

/*****************************************************************/

/* swaps rows of the CSV array element-by-element for the purposes of sorting
   (adapated from int_swap function in insertionsort.c by Alistair Moffat)
*/
void
row_swap(csv_t D, int curr_row, int dc) {
    int i, next_row=curr_row+1;
    double tmp;

    for (i=0; i<dc; i++) {
        tmp = D[curr_row][i];
        D[curr_row][i] = D[next_row][i];
        D[next_row][i] = tmp;
    }
    DUMP_INT(ONE_SWAP);
}

/*****************************************************************/

/* the 'p' plot command
*/
void
do_plot(csv_t D, head_t H[], int dr, int dc, int ccols[], int nccols) {
    int i, j, c;
    double elemkey = D[0][ccols[0]];    /* used to check if all elements in
                                           the specified columns are equal */

    /* print a newline to move below the prompt line */
    printf("\n");

    /* check if all elements in the specified columns are equal */
    for (i=0; i<nccols; i++) {
        c = ccols[i];
        for (j=0; j<dr; j++) {
            if (cmp(elemkey,D[j][c])!=EQUAL) {
                plot_histogram(D, dr, ccols, nccols);
                break;
            }
        }
         /* if we have already plotted the histogram then we can stop checking
            if the elements are all equal */
        if (j<dr) {
            break;
        }
    }
    /* if we have checked all necessary elements now then they must all be
       equal and we don't need to plot a histogram */
        if (i==nccols) {
```

```
            printf("all selected elements are %.1f\n", elemkey);
        }
}

/*************************************************************/

/* plots a frequency histogram of all data in the specified columns as a
   "sideways" bar chart
*/
void
plot_histogram(csv_t D, int dr, int ccols[], int nccols) {
    int i, j, k, scal_fact=BASE_SCALE;
    double E[ENDPOINTS];            /* array that stores the 11 endpoints */
    freq_t F;                       /* stores the frequencies of each element
                                       in an appropriate interval row and
                                       specified column inside a 2d matrix */

    find_endpoints(D, E, dr, ccols, nccols);
    create_frequency_matrix(D, F, E, &scal_fact, dr, ccols, nccols);

    /* plot the histogram using the frequency array and endpoint array */
    printf("%11.1f +\n", E[0]);
    for (i=0; i<INTERVALS; i++) {
        for (j=0; j<nccols; j++) {
            printf("%11d |", ccols[j]);
            /* simply print a ] for every unit of frequency in the relevant
               section of the frequency array */
            for (k=0; k<F[i][j]; k++){
                printf("]");
            }
            printf("\n");
        }
        printf("%11.1f +\n", E[i+1]);
    }
    printf("  scale = %d\n", scal_fact);
}

/*************************************************************/

/* finds and stores the 11 endpoints of the 10 equal-width intervals for the
   graph by calculating the range and individual interval widths
*/
void
find_endpoints(csv_t D, double E[], int dr, int ccols[], int nccols) {
    int i, j, c;
    double plot_min=D[0][ccols[0]], plot_max=D[0][ccols[0]];
    double range, interval_width;

    /* find min and max value across all specified columns */
    for (i=0; i<nccols; i++) {
        c = ccols[i];
        for (j=0; j<dr; j++) {
            if (cmp(plot_min,D[j][c])==G_THAN) {
                plot_min = D[j][c];
            } else if (cmp(plot_max,D[j][c])==L_THAN) {
                plot_max = D[j][c];
            }
        }
    }
    /* find the range of the histogram and split it into 10 intervals of
       equal width */
    range = (plot_max+EQUAL_RANGE) - (plot_min-EQUAL_RANGE);
    interval_width = range/INTERVALS;

    /* create an array of the 11 endpoints of the 10 intervals */
    E[0] = plot_min - EQUAL_RANGE;
    for (i=1; i<ENDPOINTS; i++) {
        E[i] = E[i-1] + interval_width;

        DUMP_DBL(E[i]);
    }
}

/*************************************************************/
```

```
/* creates a 2d matrix of the frequency of each relevant element within each
   of the 10 intervals while also adjusting the scaling factor if necessary
*/
void
create_frequency_matrix(csv_t D, freq_t F, double E[], int *scal_fact, int dr,
                        int ccols[], int nccols) {
    int i, j, k, c;

    /* create an array of frequencies with intervals as rows and specified
       columns as columns */
    for (i=0; i<INTERVALS; i++) {
        for (j=0; j<nccols; j++) {
            F[i][j] = ZERO_FREQUENCY;
            c = ccols[j];
            /* check each relevant element one at a time and see which
               interval it is a part of and add 1 to the frequency of this
               interval within the relevant column in the frequency array */
            for (k=0; k<dr; k++) {
                if (cmp(E[i],D[k][c])==L_THAN && cmp(D[k][c],E[i+1])<=EQUAL) {

                    DUMP_DBL(F[i][j]);

                    F[i][j] += 1;

                    DUMP_DBL(F[i][j]);
                    DUMP_DBL(D[k][c]);
                    DUMP_DBL(E[i+1]);
                }
                /* if any of the frequencies will exceed 60 on the graph then
                   increase the scaling factor by 1 */
                if (F[i][j] > HISTOGRAM_WIDTH * *scal_fact) {
                    *scal_fact += BASE_SCALE;
                }
            }
        }
    }

    /* if the scaling factor has been increased, convert the frequencies by
       dividing each element by the scaling factor and round up to nearest
       integer */
    if (*scal_fact>BASE_SCALE) {
        for (i=0; i<INTERVALS; i++) {
            for (j=0; j<nccols; j++) {
                F[i][j] = ceil(F[i][j]/(*scal_fact * DOUBLE_CONVERSION));
            }
        }
    }
}

/*****************************************************************/
/*                      algorithms are fun                       */
/*****************************************************************/
```