

Documentation of The Exercise 5-17

Question:

Add a field-searching capability, so sorting may be done on fields within lines, each field sorted according to an independent set of options. (The index for this book was sorted with -df for the index category and -n for the page numbers.)

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <limits.h>
#include <math.h>

#define MAX_NUMBER_OF_LINES 5000

#define MAX_LINE_LENGTH 1000
#define ALLOC_MEM_SIZE 10000

#define MAXIMUM_NUMBER_OF_FIELDS 100
#define MAXIMUM_NUMBER_OF_FIELD_OPTIONS 4

#define INT_MAXIMUM_NUMBER_OF_DIGITS (int)(floor(log10(fabs(INT_MAX))) + 1)

static char alloc_buf[ALLOC_MEM_SIZE];
static char *alloc_p = alloc_buf;

char *alloc(int size);
void afree(char *ptr);

int get_line(char line[], int max_line_len);

int arguments_list_parsing(int argc, char *argv[]);

int nth_blank_pos_in_string(const char *s, int n);
char *sub_string(const char *s, int start, int end);
int read_lines(char *line_ptr[], const int max_nr_of_lines);
void write_lines(char *line_ptr[], const int nr_of_lines);

int numcmp(const char *s1, const char *s2);
int my_strcasecmp(const char *s1, const char *s2);
int fields_compare(const char *s1, const char *s2);
void swap(void *v[], int i, int j);
void my_quick_sort(void *v[], int start, int end, int (*comp)(void *, void *));

int sorting_order = 1; /* 1 ascending, -1 descending */
int fold = 0; /* 0 case sensitive, 1 case insensitive */
```

```

int directory = 0; /*0 normal, 1 directory*/
int (*comp)(const char *, const char *) = my_strcasecmp;

enum field_option
{
    INDEX_OF_FIELD_COMP,
    ORDER,
    FOLD,
    DIRECTORY
};

int nr_of_fields_to_compare = 0;
int (*fields_indexes_to_comp[MAXIMUM_NUMBER_OF_FIELDS])(const char *, const char *);
int fields_options[MAXIMUM_NUMBER_OF_FIELDS][MAXIMUM_NUMBER_OF_FIELD_OPTIONS];

int main(int argc, char *argv[])
{
    if (!arguments_list_parsing(argc, argv))
    {
        puts("Error: invalid arguments.");
        return EXIT_FAILURE;
    }

    int nr_of_lines;
    char *line_ptr[MAX_NUMBER_OF_LINES];

    if ((nr_of_lines = read_lines(line_ptr, MAX_NUMBER_OF_LINES)) != -1)
    {
        my_quick_sort((void **)line_ptr, 0, nr_of_lines - 1, (int (*)(void *, void *))comp);
        if(EXIT_SUCCESS) printf("The result after applying q-sort: \n");
        write_lines(line_ptr, nr_of_lines);
    }
    else
    {
        puts("Error: input too large.");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}

int arguments_list_parsing(int argc, char *argv[]) {
    int i;
    for (i = 1; i < argc; ++i) {
        int arguments_length = strlen(argv[i]);
        if (arguments_length > 1 && argv[i][0] == '-')
        {
            int j;

```

```

for (j = 1; j < arguments_length; ++j)
{
    if (isdigit(argv[i][j]) && !fields_options[i - 1][INDEX_OF_FIELD_COMP])
    {
        char field_index_str[INT_MAXIMUM_NUMBER_OF_DIGITS];

        int k = 0;
        while (isdigit(argv[i][j]) && j < arguments_length && k <
INT_MAXIMUM_NUMBER_OF_DIGITS)
        {
            field_index_str[k++] = argv[i][j++];
        }
        field_index_str[k] = '\0';

        fields_options[i - 1][INDEX_OF_FIELD_COMP] = atoi(field_index_str);

        --j;
        ++nr_of_fields_to_compare;
    }
    else
    {
        switch (argv[i][j])
        {
            case 'n':
                comp = numcmp;
                break;

            case 'f':
                fold = 1;
                break;

            case 'd':
                directory = 1;
                break;

            case 'r':
                sorting_order = -1;
                break;

            default:
                return 0;
                break;
        }
    }
}

if (nr_of_fields_to_compare || argc > 2)
{

```

```

        if (!fields_options[i - 1][INDEX_OF_FIELD_COMP])
        {
            return 0;
        }

        fields_indexes_to_comp[i - 1] = comp;
        fields_options[i - 1][ORDER] = sorting_order;
        fields_options[i - 1][FOLD] = fold;
        fields_options[i - 1][DIRECTORY] = directory;

        comp = my_strcasecmp;
        sorting_order = 1;
        fold = 0;
        directory = 0;
    }
}
else
{
    return 0;
}
}

if (nr_of_fields_to_compare && nr_of_fields_to_compare == argc - 1)
{
    comp = fields_compare;
}
else if (argc > 2)
{
    return 0;
}

return 1;
}

```

```

int nth_blank_pos_in_string(const char *s, int n)
{
    int pos = 0;
    while (n && *s != '\0')
    {
        if (*s == ' ' || *s == '\t')
        {
            do
            {
                ++pos;
                ++s;
            } while (*s == ' ' || *s == '\t');
        }
        --n;
    }
}

```

```

    }
    else
    {
        ++pos;
        ++s;
    }
}

return pos;
}

char *sub_string(const char *s, int start, int end)
{
    if (start > end)
    {
        return NULL;
    }

    const int len = end - start;
    char *dest = alloc(len + 1);
    int i;
    for (i = start; i < end && s[i] != '\0'; ++i)
    {
        *dest = s[i];
        ++dest;
    }
    *dest = '\0';

    return dest - len;
}

int get_line(char line[], int max_line_len)
{
    int c;
    int i;

    for (i = 0; i < max_line_len - 1 && (c = getc(stdin)) != EOF && c != '\n'; ++i)
    {
        line[i] = c;
    }

    if (c == '\n')
    {
        line[i] = c;
        ++i;
    }

    line[i] = '\0';
}

```

```

    return i;
}

int read_lines(char *line_ptr[], const int max_nr_of_lines)
{
    int line_length;
    int nr_of_lines = 0;

    char *current_line = alloc(MAX_LINE_LENGTH);
    char *current_line_copy = NULL;

    while ((line_length = get_line(current_line, MAX_LINE_LENGTH)))
    {
        if (nr_of_lines >= max_nr_of_lines || (current_line_copy = alloc(line_length)) == NULL)
        {
            return -1;
        }
        else
        {
            current_line[line_length - 1] = '\0';
            strcpy(current_line_copy, current_line);
            line_ptr[nr_of_lines++] = current_line_copy;
        }
    }

    afree(current_line);

    return nr_of_lines;
}

void write_lines(char *line_ptr[], const int nr_of_lines)
{
    int i;
    for (i = 0; i < nr_of_lines; ++i)
    {
        puts(line_ptr[i]);
        afree(line_ptr[i]);
    }
}

int numcmp(const char *s1, const char *s2)
{
    double nr1 = atof(s1);
    double nr2 = atof(s2);

    if (nr1 < nr2)

```

```

{
    return sorting_order * -1;
}
else if (nr1 > nr2)
{
    return sorting_order * 1;
}

return 0;
}

```

```

int my_strcasecmp(const char *s1, const char *s2)
{
    while (*s1 != '\0' && *s2 != '\0')
    {
        if (directory)
        {
            while (*s1 != '\0' && !isalnum(*s1) && !isspace(*s1))
            {
                ++s1;
            }
            while (*s2 != '\0' && !isalnum(*s2) && !isspace(*s2))
            {
                ++s2;
            }
        }

        int result = fold ? tolower(*s1) - tolower(*s2) : *s1 - *s2;
        if (result == 0)
        {
            ++s1;
            ++s2;
        }
        else
        {
            return sorting_order * result;
        }
    }

    return 0;
}

```

```

int fields_compare(const char *s1, const char *s2)
{
    int i = 0;
    while (i < nr_of_fields_to_compare)
    {
        int start_s1 = nth_blank_pos_in_string(s1, fields_options[i][INDEX_OF_FIELD_COMP] - 1);

```

```

int end_s1 = nth_blank_pos_in_string(s1, fields_options[i][INDEX_OF_FIELD_COMP]);
char *field_s1 = sub_string(s1, start_s1, end_s1);

int start_s2 = nth_blank_pos_in_string(s2, fields_options[i][INDEX_OF_FIELD_COMP] - 1);
int end_s2 = nth_blank_pos_in_string(s2, fields_options[i][INDEX_OF_FIELD_COMP]);
char *field_s2 = sub_string(s2, start_s2, end_s2);

comp = fields_indexes_to_comp[i];
sorting_order = fields_options[i][ORDER];
fold = fields_options[i][FOLD];
directory = fields_options[i][DIRECTORY];

int comp_result = comp(field_s1, field_s2);

afree(field_s1);
afree(field_s2);

if (comp_result == 0)
{
    ++i;
}
else
{
    return comp_result;
}
}

return 0;
}

void swap(void *v[], int i, int j)
{
    void *temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

void my_quick_sort(void *v[], int start, int end, int (*comp)(void *, void *))
{
    if ((long)start >= (long)end)
    {
        return;
    }

    swap(v, start, (start + end) / 2);

    int last = start;

```



```

int i;
for (i = start + 1; i <= end; ++i)
{
    if ((*comp)(v[i], v[start]) < 0)
    {
        swap(v, ++last, i);
    }
}

swap(v, start, last);
my_quick_sort(v, start, last - 1, comp);
my_quick_sort(v, last + 1, end, comp);
}

char *alloc(int size)
{
    if (alloc_buf + ALLOC_MEM_SIZE - alloc_p >= size)
    {
        alloc_p += size;
        return alloc_p - size;
    }

    return NULL;
}

void afree(char *ptr)
{
    if (ptr >= alloc_buf && ptr < alloc_buf + ALLOC_MEM_SIZE)
    {
        alloc_p = ptr;
    }
}

```

Here is the basic functionalities of the program that I have implemented:

The my_quick_sort() function that I have implemented is the quicksort algorithm used for sorting an array of elements. It takes the following parameters:

void *v[]: This parameter is a pointer to an array of void pointers. It represents the array to be sorted. Since it is an array of void pointers, it can be used to sort arrays of different data types.

int start: This parameter represents the starting index of the subarray to be sorted. It indicates the first element of the subarray.

int end: This parameter represents the ending index of the subarray to be sorted. It indicates the last element of the subarray.

`int (*comp)(void *, void *)`: This parameter is a function pointer to the comparison function that will be used for sorting. The comparison function should take two void pointers as arguments and return an integer indicating the relative order of the elements.

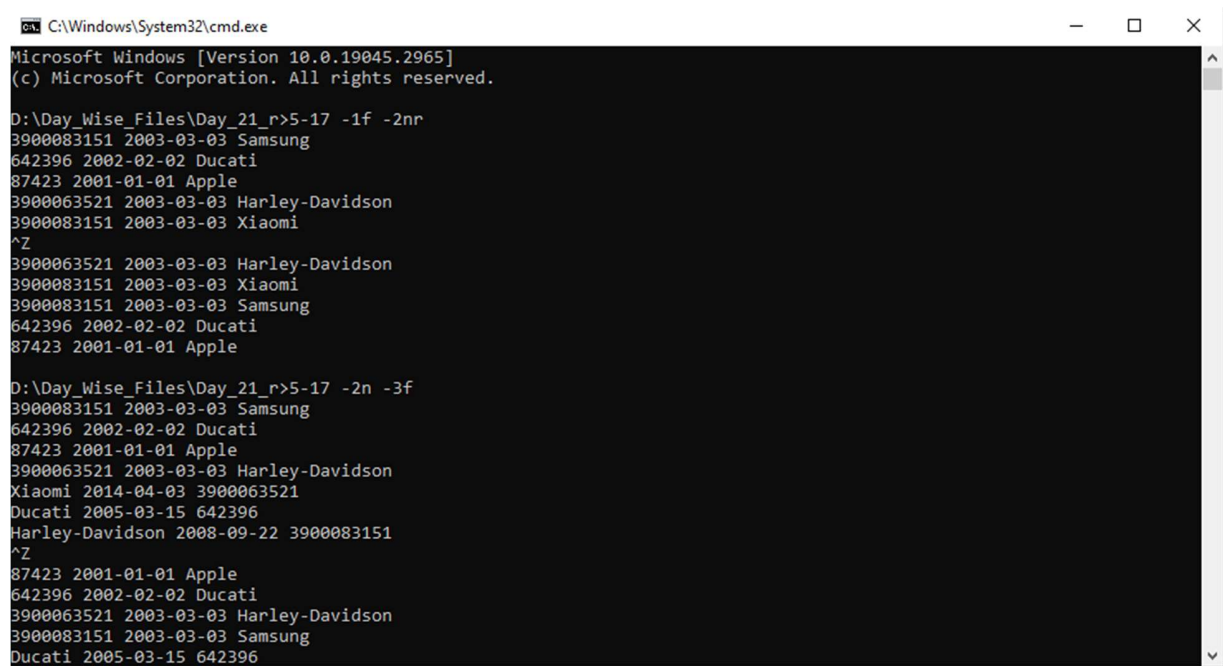
The `my_quick_sort()` function uses the quicksort algorithm to divide the array into subarrays, recursively sorts the subarrays, and then combines them to obtain the sorted array. It works by selecting a pivot element, partitioning the array into two subarrays based on the pivot, and recursively applying the same process to the subarrays.

The comparison function (`comp`) is used to determine the order of elements during the sorting process. By providing different comparison functions, you can control the sorting behavior based on your specific requirements, such as sorting in ascending or descending order, or sorting based on different criteria.

Overall, the `my_quick_sort()` function is a flexible sorting algorithm that can be used to sort arrays of different data types and allows customization of the sorting behavior through the comparison function.

Inputs and Outputs:

1.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

D:\Day_Wise_Files\Day_21_r>5-17 -1f -2nr
3900083151 2003-03-03 Samsung
642396 2002-02-02 Ducati
87423 2001-01-01 Apple
3900063521 2003-03-03 Harley-Davidson
3900083151 2003-03-03 Xiaomi
^Z
3900063521 2003-03-03 Harley-Davidson
3900083151 2003-03-03 Xiaomi
3900083151 2003-03-03 Samsung
642396 2002-02-02 Ducati
87423 2001-01-01 Apple

D:\Day_Wise_Files\Day_21_r>5-17 -2n -3f
3900083151 2003-03-03 Samsung
642396 2002-02-02 Ducati
87423 2001-01-01 Apple
3900063521 2003-03-03 Harley-Davidson
Xiaomi 2014-04-03 3900063521
Ducati 2005-03-15 642396
Harley-Davidson 2008-09-22 3900083151
^Z
87423 2001-01-01 Apple
642396 2002-02-02 Ducati
3900063521 2003-03-03 Harley-Davidson
3900083151 2003-03-03 Samsung
Ducati 2005-03-15 642396
```

2.

```
C:\Windows\System32\cmd.exe
D:\Day_Wise_Files\Day_21_r>5-17 -1f -2nr -d
Error: invalid arguments.

D:\Day_Wise_Files\Day_21_r>5-17 -1f -2nrd
3900083151 2003-03-03 Samsung
642396 2002-02-02 Ducati
87423 2001-01-01 Apple
3900063521 2003-03-03 Harley-Davidson
3900083151 2003-03-03 Xiaomi
1234567890 2005-05-05 Google
^Z
1234567890 2005-05-05 Google
3900063521 2003-03-03 Harley-Davidson
3900083151 2003-03-03 Samsung
3900083151 2003-03-03 Xiaomi
642396 2002-02-02 Ducati
87423 2001-01-01 Apple
D:\Day_Wise_Files\Day_21_r>
```