DERSİN ADI: Algoritma Analizi

DERSİN EĞİTMENİ: Dr. Öğr. Üyesi Mehmet Amaç GÜVENSAN

ÖĞRENCİ ADI: Ertuğrul ŞENTÜRK

ÖĞRENCİ NO: 18011028

ÖĞRENCİ MAIL: mdesenturk@gmail.com

DÖNEM: 3

GRUP NO: 2

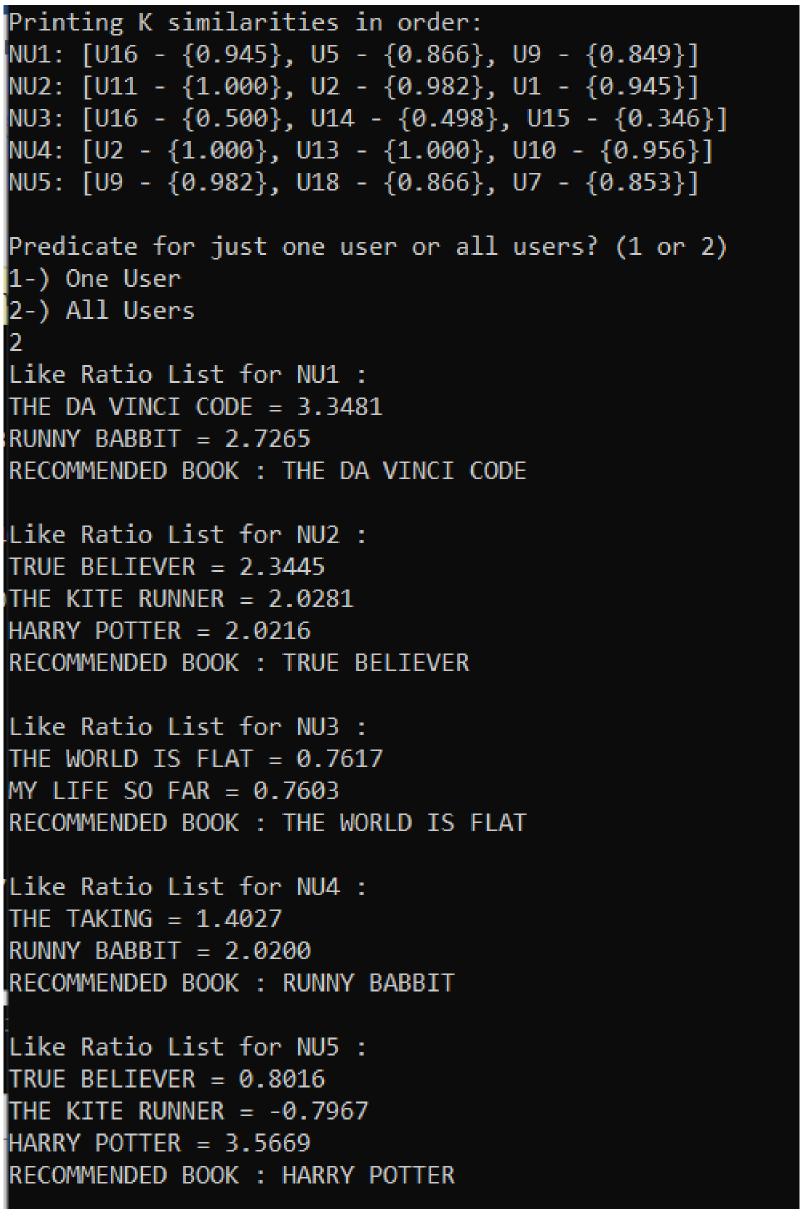
ÖDEV NO: Proje

ÖDEV KONUSU: Kitap Öneri Sistemi

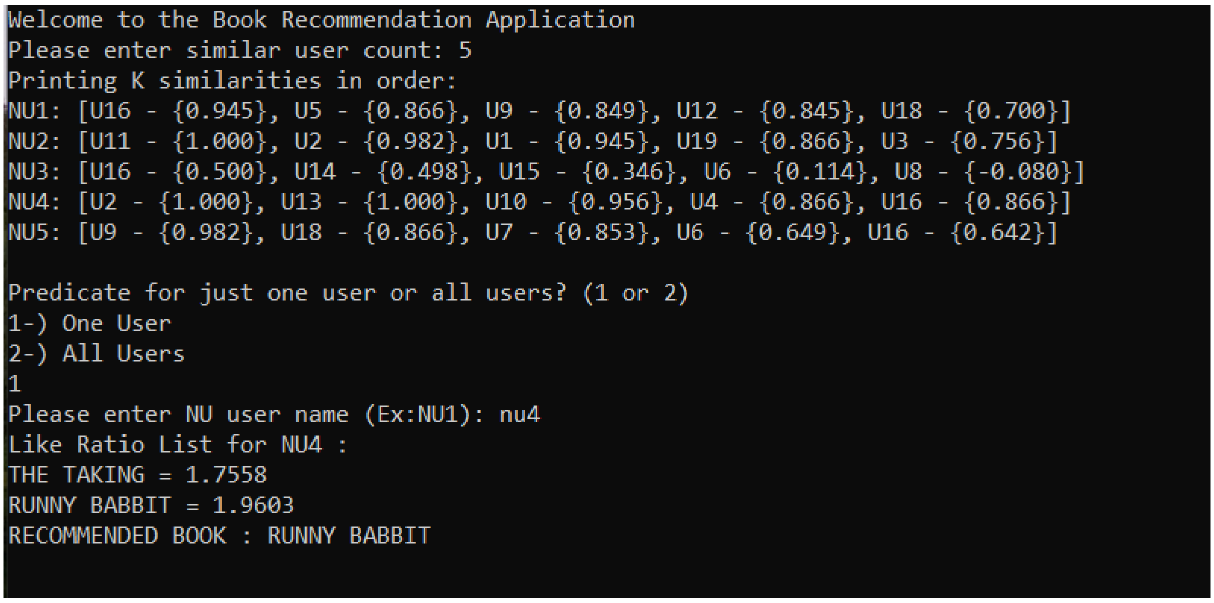
Algoritma:

1. Verilen dosyadaki veriler uygun şekilde okunup dosyaya kaydedildi.
2. Kullanıcıdan hesaplamada kullanılacak benzer kullanıcı sayısı alındı.
3. Verilen inputlar ile her nu üyesinin u üyesi ile pearson kaysayısı hesaplandı.
4. İstenilen benzer kullanıcı sayısı kadar maximum eleman dizinin başına taşındı ve indexleri k_max adından bir diziye kaydedildi.
5. Bu kullanıcıların benzerlik oranı hakkında bilgi yazdırıldı.
6. Sonrasında kullanıcıdan tek kullanıcıya ait mi yoksa tüm kullanıcıya ait mi öneri yapılacağının bilgisi alındı.
7. Alınan bilgiye göre seçilen nu kullanıcıların okumadığı kitaplar hakkında predicate fonksiyonu çağırılarak tahmin yapıldı.
8. Tahmin yapılan kitapların tahmin değerleri arasından maksimumu kullanıcı için önerilen kitap olacağından o kitabın bilgisi yazdırıldı.

Ekran görüntüleri:



```
Printing K similarities in order:
NU1: [U16 - {0.945}, U5 - {0.866}, U9 - {0.849}]
NU2: [U11 - {1.000}, U2 - {0.982}, U1 - {0.945}]
NU3: [U16 - {0.500}, U14 - {0.498}, U15 - {0.346}]
NU4: [U2 - {1.000}, U13 - {1.000}, U10 - {0.956}]
NU5: [U9 - {0.982}, U18 - {0.866}, U7 - {0.853}]

Predicate for just one user or all users? (1 or 2)
1-) One User
2-) All Users
2
Like Ratio List for NU1 :
THE DA VINCI CODE = 3.3481
RUNNY BABBIT = 2.7265
RECOMMENDED BOOK : THE DA VINCI CODE

Like Ratio List for NU2 :
TRUE BELIEVER = 2.3445
THE KITE RUNNER = 2.0281
HARRY POTTER = 2.0216
RECOMMENDED BOOK : TRUE BELIEVER

Like Ratio List for NU3 :
THE WORLD IS FLAT = 0.7617
MY LIFE SO FAR = 0.7603
RECOMMENDED BOOK : THE WORLD IS FLAT

Like Ratio List for NU4 :
THE TAKING = 1.4027
RUNNY BABBIT = 2.0200
RECOMMENDED BOOK : RUNNY BABBIT

Like Ratio List for NU5 :
TRUE BELIEVER = 0.8016
THE KITE RUNNER = -0.7967
HARRY POTTER = 3.5669
RECOMMENDED BOOK : HARRY POTTER
```



```
Welcome to the Book Recommendation Application
Please enter similar user count: 5
Printing K similarities in order:
NU1: [U16 - {0.945}, U5 - {0.866}, U9 - {0.849}, U12 - {0.845}, U18 - {0.700}]
NU2: [U11 - {1.000}, U2 - {0.982}, U1 - {0.945}, U19 - {0.866}, U3 - {0.756}]
NU3: [U16 - {0.500}, U14 - {0.498}, U15 - {0.346}, U6 - {0.114}, U8 - {-0.080}]
NU4: [U2 - {1.000}, U13 - {1.000}, U10 - {0.956}, U4 - {0.866}, U16 - {0.866}]
NU5: [U9 - {0.982}, U18 - {0.866}, U7 - {0.853}, U6 - {0.649}, U16 - {0.642}]

Predicate for just one user or all users? (1 or 2)
1-) One User
2-) All Users
1
Please enter NU user name (Ex:NU1): nu4
Like Ratio List for NU4 :
THE TAKING = 1.7558
RUNNY BABBIT = 1.9603
RECOMMENDED BOOK : RUNNY BABBIT
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <float.h>

// Max available book count
#define MAX_BOOK_COUNT 50
// Max available book name
#define MAX_BOOK_NAME 200
// Max available user count
#define MAX_USER_COUNT 100
// Buffer size for a row
#define ROW_BUFFER 10
// Database file name
const static char data[] = "RecomendationDataSet.csv";
// Struct definition for keeping user info
// name = user name
// book_list = book ratings giving by the user
struct{
        char name[ROW_BUFFER];
        int* book_list;
}typedef user;

// Database reading functions
int read_book_names(FILE* database,char *** book_names);
int read_users(FILE* database, user users[], int book_count);
bool read_database(const char* file_name,FILE** database);
bool skip_line(FILE* database);
// Pearson calculation
float calculate_pearson(user u1, user u2, int book_count);
float* calculate_all_similarities(user nu_user,user* u_users,int u_count,int book_count);
// K max element calculation
int* find_kmax(int k, float* values, int size);
// Predication calculation for a book
float calculate_pred(user nu, int book_number, user u_users[],const int* kmax, int k,int book_count);
void predicate_user(user nu_user,int nu_order,user u_users[],int k,int** kmax_list,char** book_names,int book_count);
// Test function
int ** test1(int k, user nu_users[],int nu_count,user u_users[],int u_count,int book_count);
void test2and3(user nu_users[], int nu_count, user u_users[], int** kmax_list, int k, char** book_names,int book_count);
// Dellocation functions
void free_book_names(char** book_names);
void free_users(user users[],int u_count);
void free_kmax_list(int** kmax_list,int nu_count);

int main(){
        // variable definitions
        int k;
        int book_count;
        int u_count;
        int nu_count;
        char** book_names;
        int** kmax_list;
        user u_users[MAX_USER_COUNT];
        user nu_users[MAX_USER_COUNT];

        // reading database
        FILE* database;
        if (!read_database(data,&database))
                return 0;
```

```c
        // initializing variables
        book_count = read_book_names(database,&book_names);
        u_count = read_users(database,u_users,book_count);
        skip_line(database);
        nu_count = read_users(database,nu_users,book_count);
        fclose(database);
        printf("Welcome to the Book Recommendation Application\n");
        printf("Please enter similar user count: ");
        scanf("%d",&k);
        if (k>u_count || k<1){
                printf("k value is invalid");
                return 0;
        }
        // Starting test1 and generating kmax_list
        kmax_list = test1(k,nu_users,nu_count,u_users,u_count,book_count);
        // Starting test2 and test 3
        test2and3(nu_users, nu_count, u_users, kmax_list, k, book_names, book_count);
        // memory is freed to recover from memory leak
        free_book_names(book_names);
        free_users(u_users, u_count);
        free_users(nu_users, nu_count);
        free_kmax_list(kmax_list,nu_count);
        return 0;
}
/* Book name reading function
        param:
                database = file pointer to read database from file
                book_names == array which is keeps book names;

        var:
                i = iterator;
                book_count = count of books;
                buff = buffer to keep a row content;
                c = temporary char for reading character

        return:
                book_count = count of books;
*/

int read_book_names(FILE* database,char *** book_names){
        int i;
        int book_count;
        char buff[MAX_BOOK_NAME];
        char c;
        // allocate memory for book list
        *book_names = (char**) malloc(sizeof(char*) * MAX_BOOK_COUNT);
        for(i = 0; i < MAX_BOOK_COUNT; i++){
                // allocate memory for each book
                (*book_names)[i] = (char*) malloc(sizeof(char) * MAX_BOOK_NAME);
        }
        // Remove first , for first empty
        c = fgetc(database);
        i = 0;
        book_count = 0;
        while(c != '\n' && c != EOF){
                c = fgetc(database);
                if(c!=','){
                                // Add characters to buffer
                                buff[i] = c;
                                i++;
                }
        else{
                // Add endl at the end of the buffer for stoping buffer
                buff[i] = '\0';
                // Copy buffer to array
                strncpy((*book_names)[book_count],buff,i+1);
                // Increase book count
                book_count++;
                // Start buff from begining
                i = 0;
        }
    }
    // Add last cell.
    buff[i-1] = '\0';
    strncpy((*book_names)[book_count], buff, i+1);
    // Return book count
    return book_count+1;
}

/* User reading function
    param:
        database = file pointer to read database from file
        users == array which is users;
        book_count = count of books;

    var:
        i,j,k = iterator;
        buff = buffer to keep a row content;
        c = temporary char for reading character
        temp = keeps integer value of a row

    return:
        user count;
*/
int read_users(FILE* database, user users[], int book_count){
    int i, j, k;
    char buff[ROW_BUFFER];
    char c;
    int temp;
    // Read first char
    c = fgetc(database);
    i = 0;
    // If first char is , that means first cell is empty
    while(c!=',' && c!=EOF){
        // Set values for new user
        users[i].book_list = (int*) malloc(sizeof(int)*book_count);
        j = 0;
        // Read until end of the line
        while (c!='\n' && c != EOF){
                k=0;
                // Read a cell
                while(c != ',' && c != '\n' && c != EOF){
                        buff[k] = c;
                        c = fgetc(database);
                        k++;
                }
                // For end of the cells
                if(c != '\n'){
                        // Copt last cell
                        buff[k] = '\0';
                        // For first cell read username
                        if(j == 0){
                                strcpy(users[i].name, buff);
                        }
                        // For other cells convert value to integer and add it into book list
                        else{
                                temp = atoi(buff);
                                users[i].book_list[j-1] = temp;
                        }
                        c = fgetc(database);
                        // Clear buffer for end of the line
                        if(c == '\n')
                                buff[0] = '\0';
                }
                j++;
        }
        // Add last cell
        temp = atoi(buff);
        if(temp != 0){
                j--;
        }
        users[i].book_list[j-1] = temp;
        while(c == '\n')
                c = fgetc(database);
        i++;
    }
    // Return user count
    return i;
}
/* File reading function
    param:
        file_name = file location for reading
        database = file pointer to read database from file;
    return:
        success of reading
*/
bool read_database(const char* file_name,FILE** database){
    *database = fopen(file_name,"r");
    if(!*database){
        printf("File not found\n");
        return false;
    }
    return true;
}
/* File reading function
    param:
        database = file pointer to read database from file;


    var:
        c = temporary char for reading character
    return:
        success of reading
*/
bool skip_line(FILE* database){
    char c = '.';
    while(c != '\n')
        c = fgetc(database);
}
```

```
/* Pearson coefficent calculation function
        param:
                u1 = first user
                u2 = second user
                book_count = book count;

        var:
                i = iterator
                numerator = top of the division
                denominator1, denominator2 = botton of the division
                average1, average2 = average values
                common_book = book count readed by both user

        return:
                pearson coefficent
*/
float calculate_pearson(user u1, user u2, int book_count){
        int i;
        float numerator = 0;
        float denominator1 = 0;
        float denominator2 = 0;
        float average1 = 0;
        float average2 = 0;
        int common_book = 0;
        //Calculate sum value for books both readed
        for(i=0;i<book_count;i++){
                // Only calculate if both user readed same book
                if(u1.book_list[i]>0 && u2.book_list[i]>0){
                                average1+=u1.book_list[i];
                                average2+=u2.book_list[i];
                                common_book++;
                }
        }
        // Calculate average by dividing common book count
        average1 /= (float)common_book;
        average2 /= (float)common_book;
        for(i=0;i<book_count;i++){
                // Only calculate if both user readed same book
                if(u1.book_list[i]>0 && u2.book_list[i]>0){
                                numerator += ((float)u1.book_list[i] - average1) * ((float)u2.book_list[i] - average2);
                                denominator1 += pow(((float)u1.book_list[i] - average1), 2);
                                denominator2 += pow(((float)u2.book_list[i] - average2), 2);
                }
        }
        // calculate result
        return numerator / (sqrt(denominator1) * sqrt(denominator2));
}
/* Pearson coefficent array generator for a nu user
        param:
                nu_user = nu user, u_users = u user list , u_count = u user count , book_count = book count;
        var:
                i = iterator , pearson_values = list of pearson values for a nu user
        return:
                list of pearson values for a nu user*/
float* calculate_all_similarities(user nu_user,user* u_users,int u_count,int book_count){
        int i;
        float* pearson_values = (float*) malloc(sizeof(float)*u_count);
        for(i=0;i<u_count;i++){
                pearson_values[i] = calculate_pearson(nu_user,u_users[i],book_count);
        }
        return pearson_values;
}
```

```
/* Calculates first k maxiumum
        param:
                k = similar user count
                values = pearson values for a nu user
                size = u user count
        var:
                i,j = iterator
                t,temp = temp values
                kmax = array to keeps k max users index;
        return:
                array to keeps k max users index
*/
int* find_kmax(int k, float* values, int size){
        int i,j;
        int t;
        float temp;
        int* kmax = (int*) malloc(sizeof(int)*size);
        for(i=0;i<size;i++){
                kmax[i] = i;
        }
        for(i=0;i<k;i++){
                for(j=i+1;j<size;j++){
                                // swap index array and value array partial bubble sort with k step
                                if(values[i]<values[j]){
                                                temp = values[i];
                                                values[i] = values[j];
                                                values[j] = temp;
                                                t = kmax[i];
                                                kmax[i] = kmax[j];
                                                kmax[j] = t;
                                }
                }
        }
        return kmax;
}

/* Calculation of predication value for a book from first k user
        param:
                nu = nu user to generate recommendation
                book_number = book order for predication
                kmax = k max u user's index
                k = similar user count
                book_count = count of all books
        var:
                i = iterator
                numerator = top part of division
                denominator = botton part of division
                similarity = keeps pearson value between u user and nu user
                user u = temp value for keeping each u user
                u_average = average value for u_users;
                nu_average = average value for nu_user;
                read_count = temporary value for readed book count for any user;
        return:
                predication value for selected book and nu user
*/
float calculate_pred(user nu, int book_number, user u_users[],const int* kmax, int k,int book_count){
        int i,j;
        float numerator = 0;
        float denominator = 0;
        float similarity;
        user u;
        float u_average;
```

```
	float nu_average = 0;
	int read_count = 0;
	for(i = 0;i<book_count;i++){
		if(nu.book_list[i]!=0){
				read_count++;
				nu_average += (float)nu.book_list[i];
		}
	}
	nu_average /= (float) read_count;
	for(i = 0; i < k; i++){
		u = u_users[kmax[i]];
		u_average = 0;
		read_count = 0;
		for(j = 0; j<book_count; j++){
				if(u.book_list[j]!=0){
						u_average += (float)u.book_list[j];
						read_count++;
				}
		}
		u_average /= (float) read_count;
		similarity = calculate_pearson(nu,u,book_count);
		numerator += similarity*((float)u.book_list[book_number]-u_average);
		denominator += similarity;
	}
	return nu_average + (numerator/denominator);
}/* Makes predication for one nu user
	param:
		nu_user = nu user to generate recommendation, nu_order = order no to user
		u_users = u user array
		k = similar user count, kmax_list = k max index array
		book_names = all book names
		book_count = count of all books
	var:
		i = iterator
		pred = predication value for each unreaded book
		denominator = botton part of division
		max_index = keeps index of max predication
		max_value = keeps value of max predication
*/
void predicate_user(user nu_user,int nu_order,user u_users[],int k,int** kmax_list,char** book_names,int book_count){
		int i;
		float pred;
		int max_index = 0;
		float max_value=FLT_MIN;
		printf("Like Ratio List for %s : \n",nu_user.name);
		for(i=0;i<book_count;i++){
				// If NU user not readed that book yet
				if(nu_user.book_list[i] == 0){
						// Calculate each non readed book predication
						pred = calculate_pred(nu_user,i,u_users,kmax_list[nu_order],k,book_count);
						// Print the values and book names
						printf("%s = %0.4f\n",book_names[i],pred);
						// Keep track of maximum
						if(max_value<pred){
								max_value = pred;
								max_index = i;
						}
				}
		}
		// Print maximum as a recommended book
		printf("RECOMMENDED BOOK : %s\n\n",book_names[max_index]);
}
```

```c
/* Calculation of predication value for a book from first k user
	param:
		k = similar user count
		nu_users = nu user array to generate recommendation
		nu_count = nu user count
		u_users = u user array for predication
		u_count = u user count
		book_count = count of all books

	var:
		i,j = iterator
		kmax_list = k max index array value for selected book and nu user
		similarities = array of similartiy values for a nu user;
		similaritiy_list = array of similarties values for each nu user;

	return:
		kmax_list = k max index array value for selected book and nu user
*/


int ** test1(int k, user nu_users[],int nu_count,user u_users[],int u_count,int book_count){
	int i,j;
	// Memory allocation for calculating arrays
	int ** kmax_list = (int**) malloc(sizeof(int*)*nu_count);
	float* similarities;
	float** similaritiy_list = (float**) malloc(sizeof(float*)*nu_count);
	// For each nu user
	for(i = 0;i<nu_count;i++){
		// Calculate similarity values for each u user
		similarities = calculate_all_similarities(nu_users[i],u_users,u_count,book_count);
		// Find k max;
		kmax_list[i] = find_kmax(k,similarities,u_count);
		// Generate similarity array for printing
		similaritiy_list[i] = similarities;
	}

	// Print similarity values and deallocate arrays
	printf("Printing K similarities in order:\n");
	for(i=0;i<nu_count;i++){
		printf("NU%d: [",i+1);
		for(j=0;j<k-1;j++){
				printf("U%d - {%0.3f}, ",kmax_list[i][j]+1,similaritiy_list[i][j]);
		}
		printf("U%d - {%0.3f}]\n",kmax_list[i][j]+1,similaritiy_list[i][j]);
		free(similaritiy_list[i]);
	}
	free(similaritiy_list);
	printf("\n");
	// Return k min list
	return kmax_list;
}/* Calculation of predication value for a book from first k user
	param:
		nu_users = nu user array to generate recommendation
		nu_count = nu user count
		u_users = u user array for predication
		kmax_list = k max index array value for selected book and nu user
		k = similar user count, book_names = names of all books
		book_count = count of all books
	var:
		i = iterator
		choice = print option - input from user
		user_name = user name - input from user
```

```
void test2and3(user nu_users[], int nu_count, user u_users[], int** kmax_list, int k, char** book_names,int book_count){
	int i;
	char choice;
	char user_name[ROW_BUFFER];
	// Get input for print one selected user's result or all users result
	printf("Predicate for just one user or all users? (1 or 2)\n1-) One User\n2-) All Users\n");
	scanf(" %c",&choice);
	// Print all user
	if(choice == '2'){
		for(i=0;i<nu_count;i++){
				predicate_user(nu_users[i],i,u_users,k,kmax_list,book_names,book_count);
		}
		return;
	}
	else{
		// Get user name
		printf("Please enter NU user name (Ex:NU1): ");
		scanf("%s",user_name);
		// Find user index
		i=0;
		while(i<nu_count && strcasecmp(user_name,nu_users[i].name) != 0)
				i++;
		// If user exist call predicate for that user
		if(i== nu_count)
				printf("User not found");
		else
				predicate_user(nu_users[i],i,u_users,k,kmax_list,book_names,book_count);
	}
}/* Memory deallocation for book names
	param:
		book_names = array of book names
	var:
		i = iterator
*/
void free_book_names(char** book_names){
	int i;
	for(i=0;i<MAX_BOOK_COUNT;i++){
		free(book_names[i]);
	}
	free(book_names);
}
/* Memory deallocation for user array
	param:  users = user array , u_count = array size
	var: i = iterator
*/
void free_users(user users[],int u_count){
	int i;
	for(i=0;i<u_count;i++){
		free(users[i].book_list);
	}
}
/* Memory deallocation for k mix list array
	param:  kmax_list = k mix list array , nu_count = array size
	var: i = iterator
*/
void free_kmax_list(int** kmax_list,int nu_count){
	int i;
	for(i=0;i<nu_count;i++){
		free(kmax_list[i]);
	}
	free(kmax_list);
}
```