Εργασία 5: Πίνακες - Δείκτες - Αρχεία

Χρήστος Μαργιώλης - Εργαστηριακό τμήμα 9

Ιανουάριος 2020

Περιεχόμενα

1 Δομή προγραμμάτων και οδηγίες εκτέλεσης 2
1.1 Εκτέλεση από Linux . . . 2
1.2 Δομή φακέλων . . . 2

2 combinations - συνδυασμοί 2
2.1 main.c . . . 2
2.2 combinations.c . . . 2
2.3 combinations.h . . . 6
2.4 arrhandler.c . . . 6
2.5 arrhandler.h . . . 8
2.6 Διάγραμμα ροής . . . 8
2.7 Περιγραφή υλοποιήσης . . . 8

3 kcombinations - συνδυασμοί με K 8
3.1 main.c . . . 8
3.2 kcombinations.c . . . 9
3.3 kcombinations.h . . . 12
3.4 arrhandler.c . . . 13
3.5 arrhandler.h . . . 14
3.6 Διάγραμμα ροής . . . 15
3.7 Περιγραφή υλοποιήσης . . . 15

4 fcombinations - συνδυασμοί από αρχείο 15
4.1 main.c . . . 15
4.2 fcombinations.c . . . 15
4.3 fcombinations.h . . . 18
4.4 arrhandler.c . . . 19
4.5 arrhandler.h . . . 21
4.6 Διάγραμμα ροής . . . 21
4.7 Περιγραφή υλοποιήσης . . . 21

5 minesweeper - ναρκαλιευτής 21
5.1 main.c . . . 21
5.2 minesweeper.c . . . 21
5.3 minesweeper.h . . . 26
5.4 gameplay.c . . . 27
5.5 gameplay.h . . . 31
5.6 settings.c . . . 31
5.7 settings.h . . . 33
5.8 outputs.c . . . 33
5.9 outputs.h . . . 35
5.10 Διάγραμμα ροής . . . 35
5.11 Περιγραφή υλοποιήσης . . . 35

6 Διευκρινήσεις 35

7 Εργαλεία 35

1 Δομή προγραμμάτων και οδηγίες εκτέλεσης

1.1 Εκτέλεση από Linux

```
$ cd path-to-program
$ make
$ make run
$ make run ARGS=txt/data.txt #fcombinations ONLY
$ make clean #optional
```

1.2 Δομή φακέλων

2 combinations - συνδυασμοί

2.1 main.c

```c
#include "combinations.h"

int main(int argc, char **argv)
{
    int *arr, N, x1, x2, y1, y2;

    N = get_n();

    arr = fill_array(N);
    quicksort(arr, 0, N-1);
    x_pair(&x1, &x2);
    y_pair(&y1, &y2);
    print_combs(arr, N, x1, x2, y1, y2);

    free(arr);

    return 0;
}
```

2.2 combinations.c

```c
#include "combinations.h"

int get_n()
{
    int N;

    do
    {
        system("clear||cls");
        printf("N (6 < N <= 49): ");
        scanf("%d", &N);
    } while (N <= 6 || N > 49);

    system("clear||cls");

    return N;
}


void x_pair(int *x1, int *x2)
{
    do
    {
        printf("x1: ");
        scanf("%d", x1);
        printf("x2: ");
        scanf("%d", x2);
    } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
}


void y_pair(int *y1, int *y2)
{
    do
    {
        printf("y1: ");
        scanf("%d", y1);
        printf("y2: ");
        scanf("%d", y2);
    } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
}


void print_combs(int *arr, int N, int x1, int x2, int y1,
    int y2)
{
    int *currComb = (int *)malloc(N * sizeof(int));
    int unFrstCond = 0, unScndCondOnly = 0, printed = 0;

    if (currComb == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        combinations(arr, currComb, 0, N-1, 0, &printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
        print_other(N, unFrstCond, unScndCondOnly, printed);
    }

    free(currComb);
}


void combinations(int *arr, int *currComb, int start, int end, int index, int *printed, int *unFrstCond, int *unScndCondOnly, int x1, int x2, int y1, int y2)
{
    int i, j;

    if (index == COMBSN)
    {
        for (j = 0; j < COMBSN; j++)
        {
            if (even_calc(currComb, x1, x2) && sum_comb_calc(currComb, y1, y2))
            {
                printf("%d ", *(currComb + j));
                if (j == COMBSN - 1) { (*printed)++; printf("\n"); }
            } // add freq
        }
        if (!even_calc(currComb, x1, x2) && sum_comb_calc(currComb, y1, y2)) (*unFrstCond)++;
        if (!sum_comb_calc(currComb, y1, y2)) (*unScndCondOnly)++;
        return;
    }

    for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
    {
        *(currComb + index) = *(arr + i);
        combinations(arr, currComb, i+1, end, index+1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
    }
}


bool even_calc(int *arr, int x1, int x2)
{
    int numEven = 0, i;

    for (i = 0; i < COMBSN; i++)
```

```
        if (*(arr + i) % 2 == 0) numEven++;

    return (numEven >= x1 && numEven <= x2) ? true : false;
}


bool sum_comb_calc(int *arr, int y1, int y2)
{
    int sumNums = 0, i;

    for (i = 0; i < COMBSN; i++)
        sumNums += *(arr + i);

    return (sumNums >= y1 && sumNums <= y2) ? true : false;
}


int frequency()
{

}


long int combinations_count(int N) // wtf ???????
{
    return (factorial(N) / (factorial(COMBSN) * factorial(N
    - COMBSN)));
}


long double factorial(int num)
{
    int i;
    long double fac;
    if (num == 0) return -1;
    else for (i = 1, fac = 1; i <= num; i++) fac *= i;
    return fac;
}


void print_other(int N, int unFrstCond, int unScndCondOnly,
    int printed)
{
    printf("\nTotal number of combinations %d to %d: %ld\n",
     N, COMBSN, combinations_count(N));
    printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
    printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
    printf("Printed combinations: %d\n", printed);
```

```
}
```

2.3 combinations.h

```c
#ifndef COMBINATIONS_H
#define COMBINATIONS_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "arrhandler.h"
#include "ccolors.h"

#define COMBSN 6

void x_pair(int *, int *);
void y_pair(int *, int *);

void print_combs(int *, int, int, int, int, int);
void combinations(int *, int *, int, int, int, int *, int *,
    int *, int, int, int, int);

bool even_calc(int *, int, int);
bool sum_comb_calc(int *, int, int);

int frequency();
long int combinations_count(int);
long double factorial(int);
void print_other(int, int, int, int); // add freq

#endif
```

2.4 arrhandler.c

```c
#include "arrhandler.h"

int *fill_array(int N)
{
    int num, i = 0;
    int *arr = (int *)malloc(N * sizeof(int));

    if (arr == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        do
        {
            printf("arr[%d]: ", i);
            scanf("%d", &num);

            if (num >= 1 && num <= 49)
            {
                if (i == 0) { *(arr + i) = num; i++; }
                else
                {
                    if (!exists_in_array(arr, N, num)) { *(
    arr + i) = num; i++; }
                    else printf("Give a different number.\n"
    );
                }
            }
            else printf("Give a number in [1, 49].\n");
        } while (i < N);
    }

    return arr;
}


bool exists_in_array(int *arr, int N, int num)
{
    int *arrEnd = arr + (N - 1);
    while (arr <= arrEnd && *arr != num) arr++;
    return (arr <= arrEnd) ? true : false;
}


void quicksort(int *arr, int low, int high)
{
    if (low < high)
    {
        int partIndex = partition(arr, low, high);
        quicksort(arr, low, partIndex - 1);
        quicksort(arr, partIndex + 1, high);
    }
}


int partition(int *arr, int low, int high)
{
    int pivot = *(arr + high);
    int i = (low - 1), j;

    for (j = low; j <= high - 1; j++)
```

```
        if (*(arr + j) < pivot)
            swap(arr + ++i, arr + j);

    swap(arr + (i + 1), arr + high);
    return (i + 1);
}


void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

2.5 arrhandler.h

```
#ifndef ARRHANDLER_H
#define ARRHANDLER_H

#include "combinations.h"

int *fill_array(int);
bool exists_in_array(int *, int, int);

void quicksort(int *, int, int);
int partition(int *, int, int);
void swap(int *, int *);

#endif
```

2.6 Διάγραμμα ροής

2.7 Περιγραφή υλοποιήσης

3 kcombinations - συνδυασμοί με K

3.1 main.c

```
#include "kcombinations.h"

int main(int argc, char **argv)
{
    int *arr, N, K, x1, x2, y1, y2;

    N = get_n();
    K = get_k(N);

    arr = fill_array(N);
    quicksort(arr, 0, N-1);
    x_pair(&x1, &x2);
    y_pair(&y1, &y2);
    print_combs(arr, N, K, x1, x2, y1, y2);

    free(arr);

    return 0;
}
```

3.2 kcombinations.c

```
#include "kcombinations.h"

int get_n()
{
    int N;

    do
    {
        system("clear||cls");
        printf("N (6 < N <= 49): ");
        scanf("%d", &N);
    } while (N <= 6 || N > 49);

    return N;
}


int get_k(int N)
{
    int K;

    do
    {
        printf("K (K < N <= 49): ");
        scanf("%d", &K);
    } while (K >= N || K > 49);

    system("clear||cls");

    return K;
}


void x_pair(int *x1, int *x2)
{
    do
    {
        printf("x1: ");
        scanf("%d", x1);
```

```c
        printf("x2: ");
        scanf("%d", x2);
    } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
}


void y_pair(int *y1, int *y2)
{
    do
    {
        printf("y1: ");
        scanf("%d", y1);
        printf("y2: ");
        scanf("%d", y2);
    } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
}


void print_combs(int *arr, int N, int K, int x1, int x2, int
     y1, int y2)
{
    int *currComb = (int *)malloc(N * sizeof(int));
    int unFrstCond = 0, unScndCondOnly = 0, printed = 0;

    if (currComb == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        combinations(arr, currComb, 0, N-1, 0, K, &printed,
    &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
        print_other(N, K, unFrstCond, unScndCondOnly,
    printed);
    }

    free(currComb);
}


void combinations(int *arr, int *currComb, int start, int
    end, int index, int K, int *printed, int *unFrstCond, int
     *unScndCondOnly, int x1, int x2, int y1, int y2)
{
    int i, j;

    if (index == K)
    {
        for (j = 0; j < K; j++)
        {
            if (even_calc(currComb, K, x1, x2) &&
    sum_comb_calc(currComb, K, y1, y2))
            {
                printf("%d ", *(currComb + j));
                if (j == K - 1) { (*printed)++; printf("\n")
    ; }
            } // add freq
        }
        if (!even_calc(currComb, K, x1, x2) && sum_comb_calc
    (currComb, K, y1, y2)) (*unFrstCond)++;
        if (!sum_comb_calc(currComb, K, y1, y2)) (*
    unScndCondOnly)++;
        return;
    }

    for (i = start; i <= end && end-i+1 >= K-index; i++)
    {
        *(currComb + index) = *(arr + i);
        combinations(arr, currComb, i+1, end, index+1, K,
    printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
    }
}


bool even_calc(int *arr, int K, int x1, int x2)
{
    int numEven = 0, i;

    for (i = 0; i < K; i++)
        if (*(arr + i) % 2 == 0) numEven++;

    return (numEven >= x1 && numEven <= x2) ? true : false;
}


bool sum_comb_calc(int *arr, int K, int y1, int y2)
{
    int sumNums = 0, i;

    for (i = 0; i < K; i++)
        sumNums += *(arr + i);

    return (sumNums >= y1 && sumNums <= y2) ? true : false;
}


int frequency()
```

```c
{

}


long int combinations_count(int N, int K) // wtf ???????
{
    return (factorial(N) / (factorial(K) * factorial(N - K))
    );
}


long double factorial(int num)
{
    int i;
    long double fac;
    if (num == 0) return -1;
    else for (i = 1, fac = 1; i <= num; i++) fac *= i;
    return fac;
}


void print_other(int N, int K, int unFrstCond, int
    unScndCondOnly, int printed)
{
    printf("\nTotal number of combinations %d to %d: %ld\n",
     N, K, combinations_count(N, K));
    printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
    printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
    printf("Printed combinations: %d\n", printed);
}
```

3.3 kcombinations.h

```c
#ifndef COMBINATIONS_H
#define COMBINATIONS_H

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#include "arrhandler.h"
#include "ccolors.h"

void x_pair(int *, int *);
void y_pair(int *, int *);

void print_combs(int *, int, int, int, int, int, int);
void combinations(int *, int *, int , int , int , int , int *,
    int *, int *, int , int , int , int);

bool even_calc(int *, int , int , int);
bool sum_comb_calc(int *, int , int , int);

int frequency();
long int combinations_count(int , int);
long double factorial(int);
void print_other(int , int , int , int , int); // add freq

#endif
```

3.4 arrhandler.c

```c
#include "arrhandler.h"

int *fill_array(int N)
{
    int num, i = 0;
    int *arr = (int *)malloc(N * sizeof(int));

    if (arr == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        do
        {
            printf("arr[%d]: ", i);
            scanf("%d", &num);

            if (num >= 1 && num <= 49)
            {
                if (i == 0) { *(arr + i) = num; i++; }
                else
                {
                    if (!exists_in_array(arr, N, num)) { *(
    arr + i) = num; i++; }
                    else printf("Give a different number.\n"
    );
                }
            }
            else printf("Give a number in [1, 49].\n");
        } while (i < N);
    }

    return arr;
}


bool exists_in_array(int *arr, int N, int num)
{
    int *arrEnd = arr + (N - 1);
    while (arr <= arrEnd && *arr != num) arr++;
    return (arr <= arrEnd) ? true : false;
}


void quicksort(int *arr, int low, int high)
{
    if (low < high)
    {
        int partIndex = partition(arr, low, high);
        quicksort(arr, low, partIndex - 1);
        quicksort(arr, partIndex + 1, high);
    }
}


int partition(int *arr, int low, int high)
{
    int pivot = *(arr + high);
    int i = (low - 1), j;

    for (j = low; j <= high - 1; j++)
        if (*(arr + j) < pivot)
            swap(arr + ++i, arr + j);

    swap(arr + (i + 1), arr + high);
    return (i + 1);
}


void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

3.5 arrhandler.h

```
#ifndef ARRHANDLER_H
#define ARRHANDLER_H

```

```
#include "kcombinations.h"

int *fill_array(int);
bool exists_in_array(int *, int, int);

void quicksort(int *, int, int);
int partition(int *, int, int);
void swap(int *, int *);

#endif
```

3.6 Διάγραμμα ροής

3.7 Περιγραφή υλοποιήσης

4 fcombinations - συνδυασμοί από αρχείο

4.1 main.c

```
#include "fcombinations.h"

int main(int argc, char **argv)
{
    int N, K;
    int *arr;
    int x1, x2, y1, y2;

    read_file(argv);

    return 0;
}
```

4.2 fcombinations.c

```
#include "fcombinations.h"

void read_file(char **argv)
{
    FILE *dataFile = fopen(argv[1], "r");

    if (dataFile == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        printf("Cool\n");
        // fscanf();
    }

    fclose(dataFile);
}


void x_pair(int *x1, int *x2)
{
    do
    {
        printf("x1: ");
        scanf("%d", x1);
        printf("x2: ");
        scanf("%d", x2);
    } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
}


void y_pair(int *y1, int *y2)
{
    do
    {
        printf("y1: ");
        scanf("%d", y1);
        printf("y2: ");
        scanf("%d", y2);
    } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
}


void print_combs(int *arr, int N, int x1, int x2, int y1,
    int y2)
{
    int *currComb = (int *)malloc(N * sizeof(int));
    int unFrstCond = 0, unScndCondOnly = 0, printed = 0;

    if (currComb == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        combinations(arr, currComb, 0, N-1, 0, &printed, &
    unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
        print_other(N, unFrstCond, unScndCondOnly, printed);
    }
```

```c

    free(currComb);
}


void combinations(int *arr, int *currComb, int start, int end, int index, int *printed, int *unFrstCond, int *unScndCondOnly, int x1, int x2, int y1, int y2)
{
    int i, j;

    if (index == COMBSN)
    {
        for (j = 0; j < COMBSN; j++)
        {
            if (even_calc(currComb, x1, x2) && sum_comb_calc(currComb, y1, y2))
            {
                printf("%d ", *(currComb + j));
                if (j == COMBSN - 1) { (*printed)++; printf("\n"); }
            } // add freq
        }
        if (!even_calc(currComb, x1, x2) && sum_comb_calc(currComb, y1, y2)) (*unFrstCond)++;
        if (!sum_comb_calc(currComb, y1, y2)) (*unScndCondOnly)++;
        return;
    }

    for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
    {
        *(currComb + index) = *(arr + i);
        combinations(arr, currComb, i+1, end, index+1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
    }
}


bool even_calc(int *arr, int x1, int x2)
{
    int numEven = 0, i;

    for (i = 0; i < COMBSN; i++)
        if (*(arr + i) % 2 == 0) numEven++;

    return (numEven >= x1 && numEven <= x2) ? true : false;
}

```

```c

bool sum_comb_calc(int *arr, int y1, int y2)
{
    int sumNums = 0, i;

    for (i = 0; i < COMBSN; i++)
        sumNums += *(arr + i);

    return (sumNums >= y1 && sumNums <= y2) ? true : false;
}


int frequency()
{

}


long int combinations_count(int N) // wtf ???????
{
    return (factorial(N) / (factorial(COMBSN) * factorial(N
    - COMBSN)));
}


long double factorial(int num)
{
    int i;
    long double fac;
    if (num == 0) return -1;
    else for (i = 1, fac = 1; i <= num; i++) fac *= i;
    return fac;
}


void print_other(int N, int unFrstCond, int unScndCondOnly,
    int printed)
{
    printf("\nTotal number of combinations %d to %d: %ld\n",
     N, COMBSN, combinations_count(N));
    printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
    printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
    printf("Printed combinations: %d\n", printed);
}
```

4.3 fcombinations.h

```c
#ifndef COMBINATIONS_H
```

```c
#define COMBINATIONS_H

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

#include "arrhandler.h"
#include "ccolors.h"

#define COMBSN 6

void read_file();

void x_pair(int *, int *);
void y_pair(int *, int *);

void print_combs(int *, int, int, int, int, int);
void combinations(int *, int *, int, int, int, int *, int *,
    int *, int, int, int, int);

bool even_calc(int *, int, int);
bool sum_comb_calc(int *, int, int);

int frequency();
long int combinations_count(int);
long double factorial(int);
void print_other(int, int, int, int); // add freq

#endif
```

4.4 arrhandler.c

```c
#include "arrhandler.h"

int *fill_array(int N)
{
    int num, i = 0;
    int *arr = (int *)malloc(N * sizeof(int));

    if (arr == NULL)
    {
        set_color(BOLD_RED);
        printf("Error! Not enough memory, exiting...\n");
        exit(EXIT_FAILURE);
        set_color(STANDARD);
    }
    else
    {
        do
        {
            printf("arr[%d]: ", i);
            scanf("%d", &num);

            if (num >= 1 && num <= 49)
            {
                if (i == 0) { *(arr + i) = num; i++; }
                else
                {
                    if (!exists_in_array(arr, N, num)) { *(arr + i) = num; i++; }
                    else printf("Give a different number.\n");
                }
            }
            else printf("Give a number in [1, 49].\n");
        } while (i < N);
    }

    return arr;
}


bool exists_in_array(int *arr, int N, int num)
{
    int *arrEnd = arr + (N - 1);
    while (arr <= arrEnd && *arr != num) arr++;
    return (arr <= arrEnd) ? true : false;
}


void quicksort(int *arr, int low, int high)
{
    if (low < high)
    {
        int partIndex = partition(arr, low, high);
        quicksort(arr, low, partIndex - 1);
        quicksort(arr, partIndex + 1, high);
    }
}


int partition(int *arr, int low, int high)
{
    int pivot = *(arr + high);
    int i = (low - 1), j;

    for (j = low; j <= high - 1; j++)
        if (*(arr + j) < pivot)
            swap(arr + ++i, arr + j);

    swap(arr + (i + 1), arr + high);
    return (i + 1);
}


void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

4.5 arrhandler.h

```
#ifndef ARRHANDLER_H
#define ARRHANDLER_H

#include "fcombinations.h"

int *fill_array(int);
bool exists_in_array(int *, int, int);

void quicksort(int *, int, int);
int partition(int *, int, int);
void swap(int *, int *);

#endif
```

4.6 Διάγραμμα ροής

4.7 Περιγραφή υλοποιήσης

5 minesweeper - ναρκαλιευτής

5.1 main.c

```
#include "minesweeper.h"

int main(int argc, char **argv)
{
    main_win();
    start();
    endwin();

    return 0;
}
```

5.2 minesweeper.c

```c
#include "minesweeper.h"

void main_win()
{
    initscr();
    noecho();
    cbreak();

    WINDOW *mainWin = newwin(0, 0, 0, 0);
    box(mainWin, 0, 0);
    refresh();
    wrefresh(mainWin);
    keypad(mainWin, true);
}


void start()
{
    int yMax, xMax;
    int numSettings = 3;
    getmaxyx(stdscr, yMax, xMax);

    WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
     5);
    box(menuWin, 0, 0);
    refresh();
    wrefresh(menuWin);
    keypad(menuWin, true);

    set_mode(menuWin);

    int COLS = set_cols(menuWin, xMax);
    int ROWS = set_rows(menuWin, yMax);
    int NMINES = set_nmines(menuWin, COLS * ROWS);

    game_win(COLS, ROWS, NMINES);
    getchar();
}


void game_win(int COLS, int ROWS, int NMINES)
{
    int yMax, xMax;
    getmaxyx(stdscr, yMax, xMax);

    WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
     // fix 43
    box(gameWin, 0, 0);
    refresh();
    wrefresh(gameWin);
    keypad(gameWin, true);

    char **dispboard = init_dispboard(gameWin, COLS, ROWS);
    char **mineboard = init_mineboard(gameWin, COLS, ROWS,
   NMINES);

    selection(gameWin, dispboard, mineboard, COLS, ROWS,
   NMINES);

    free(dispboard);
    free(mineboard);
}


char **init_dispboard(WINDOW *gameWin, int COLS, int ROWS)
{
    int i;
    char **dispboard = (char **)malloc(COLS * sizeof(char *)
   );
    for (i = 0; i < COLS; i++)
        dispboard[i] = (char *)malloc(ROWS);

    if (dispboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...
   ");
        exit(EXIT_FAILURE);
    }
    else
    {
        fill_dispboard(dispboard, COLS, ROWS);
        print_board(gameWin, dispboard, COLS, ROWS);
        getchar();
    }

    return dispboard;
}


void fill_dispboard(char **dispboard, int COLS, int ROWS)
{
    int i, j;

    for (i = 0; i < COLS; i++)
        for (j = 0; j < ROWS; j++)
            dispboard[i][j] = HIDDEN;
}


char **init_mineboard(WINDOW *gameWin, int COLS, int ROWS,
    int NMINES)
{
    int i;
    char **mineboard = (char **)malloc(COLS * sizeof(char *)
    );
    for (i = 0; i < COLS; i++)
        mineboard[i] = (char *)malloc(ROWS);

    if (mineboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...
    ");
        exit(EXIT_FAILURE);
    }
    else
    {
        place_mines(mineboard, COLS, ROWS, NMINES);
        add_adj(mineboard, COLS, ROWS);
        fill_spaces(mineboard, COLS, ROWS, NMINES);

        // tests
        //print_board(gameWin, mineboard, COLS, ROWS);
        //filewrite(mineboard, COLS, ROWS, 1, 2);
    }

    return mineboard;
}


void place_mines(char **mineboard, int COLS, int ROWS, int
    NMINES)
{
    int i, wRand, hRand;

    srand(time(NULL));

    for (i = 0; i < NMINES; i++)
    {
        wRand = rand() % COLS;
        hRand = rand() % ROWS;
        mineboard[wRand][hRand] = MINE;
    }
}


void add_adj(char **mineboard, int COLS, int ROWS)
{
    int i, j;

    for (i = 0; i < COLS; i++)
```

```c
        for (j = 0; j < ROWS; j++)
            if (!is_mine(mineboard, i, j))
                mineboard[i][j] = adj_mines(mineboard, i, j, COLS, ROWS) + '0';
}


bool is_mine(char **mineboard, int row, int col)
{
    return (mineboard[row][col] == MINE) ? true : false;
}

bool outof_bounds(int row, int col, int COLS, int ROWS)
{
    return (row < 0 || row > COLS-1 || col < 0 || col > ROWS-1) ? true : false;
}



int8_t adj_mines(char **mineboard, int row, int col, int COLS, int ROWS)
{
    int8_t numAdj = 0;

    if (!outof_bounds(row, col - 1, COLS, ROWS)      && mineboard[row][col-1]    == MINE) numAdj++; // North
    if (!outof_bounds(row, col + 1, COLS, ROWS)      && mineboard[row][col+1]    == MINE) numAdj++; // South
    if (!outof_bounds(row + 1, col, COLS, ROWS)      && mineboard[row+1][col]    == MINE) numAdj++; // East
    if (!outof_bounds(row - 1, col, COLS, ROWS)      && mineboard[row-1][col]    == MINE) numAdj++; // West
    if (!outof_bounds(row + 1, col - 1, COLS, ROWS)  && mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
    if (!outof_bounds(row - 1, col - 1, COLS, ROWS)  && mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
    if (!outof_bounds(row + 1, col + 1, COLS, ROWS)  && mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
    if (!outof_bounds(row - 1, col + 1, COLS, ROWS)  && mineboard[row-1][col+1]  == MINE) numAdj++; // South-West

    return numAdj;
}


void fill_spaces(char **mineboard, int COLS, int ROWS, int NMINES)
{
    int i, j;

    for (i = 0; i < COLS; i++)
        for (j = 0; j < ROWS; j++)
            if (mineboard[i][j] != MINE && mineboard[i][j] == '0')
                mineboard[i][j] = '-';
}
```

5.3 minesweeper.h

```
#ifndef MINESWEEPER_H
#define MINESWEEPER_H

#if defined linux || defined __unix__
#include <ncurses.h>
#elif defined _WIN32 || defined _WIN64
#include <pdcurses.h>
#include <stdint.h>
#endif

#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "settings.h"
#include "gameplay.h"
#include "outputs.h"

#define HIDDEN '#'
#define MINE '*'
#define CLEAR "
                        "

void main_win();
void start();

void game_win(int, int, int);
char **init_dispboard(struct _win_st*, int, int);
void fill_dispboard(char **, int, int);
char **init_mineboard(struct _win_st*, int, int, int);

void place_mines(char **, int, int, int);
void add_adj(char **, int, int);
bool is_mine(char **, int, int);
bool outof_bounds(int, int, int, int);
int8_t adj_mines(char **, int, int, int, int);
void fill_spaces(char **, int, int, int);

#endif
```

5.4 gameplay.c

```c
#include "minesweeper.h"

void main_win()
{
    initscr();
    noecho();
    cbreak();

    WINDOW *mainWin = newwin(0, 0, 0, 0);
    box(mainWin, 0, 0);
    refresh();
    wrefresh(mainWin);
    keypad(mainWin, true);
}


void start()
{
    int yMax, xMax;
    int numSettings = 3;
    getmaxyx(stdscr, yMax, xMax);

    WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
     5);
    box(menuWin, 0, 0);
    refresh();
    wrefresh(menuWin);
    keypad(menuWin, true);

    set_mode(menuWin);

    int COLS = set_cols(menuWin, xMax);
    int ROWS = set_rows(menuWin, yMax);
    int NMINES = set_nmines(menuWin, COLS * ROWS);

    game_win(COLS, ROWS, NMINES);
    getchar();
}


void game_win(int COLS, int ROWS, int NMINES)
{
    int yMax, xMax;
    getmaxyx(stdscr, yMax, xMax);

    WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
     // fix 43
    box(gameWin, 0, 0);
    refresh();
    wrefresh(gameWin);
    keypad(gameWin, true);

    char **dispboard = init_dispboard(gameWin, COLS, ROWS);
    char **mineboard = init_mineboard(gameWin, COLS, ROWS,
   NMINES);

    selection(gameWin, dispboard, mineboard, COLS, ROWS,
   NMINES);

    free(dispboard);
    free(mineboard);
}


char **init_dispboard(WINDOW *gameWin, int COLS, int ROWS)
{
    int i;
    char **dispboard = (char **)malloc(COLS * sizeof(char *)
   );
    for (i = 0; i < COLS; i++)
        dispboard[i] = (char *)malloc(ROWS);

    if (dispboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...
   ");
        exit(EXIT_FAILURE);
    }
    else
    {
        fill_dispboard(dispboard, COLS, ROWS);
        print_board(gameWin, dispboard, COLS, ROWS);
        getchar();
    }

    return dispboard;
}


void fill_dispboard(char **dispboard, int COLS, int ROWS)
{
    int i, j;

    for (i = 0; i < COLS; i++)
        for (j = 0; j < ROWS; j++)
            dispboard[i][j] = HIDDEN;
}

```

```

char **init_mineboard(WINDOW *gameWin, int COLS, int ROWS, 
    int NMINES)
{
    int i;
    char **mineboard = (char **)malloc(COLS * sizeof(char *)
    );
    for (i = 0; i < COLS; i++)
        mineboard[i] = (char *)malloc(ROWS);

    if (mineboard == NULL)
    {
        mvprintw(1, 1, "Error, not enough memory, exiting...
    ");
        exit(EXIT_FAILURE);
    }
    else
    {
        place_mines(mineboard, COLS, ROWS, NMINES);
        add_adj(mineboard, COLS, ROWS);
        fill_spaces(mineboard, COLS, ROWS, NMINES);

        // tests
        //print_board(gameWin, mineboard, COLS, ROWS);
        //filewrite(mineboard, COLS, ROWS, 1, 2);
    }

    return mineboard;
}


void place_mines(char **mineboard, int COLS, int ROWS, int 
    NMINES)
{
    int i, wRand, hRand;

    srand(time(NULL));

    for (i = 0; i < NMINES; i++)
    {
        wRand = rand() % COLS;
        hRand = rand() % ROWS;
        mineboard[wRand][hRand] = MINE;
    }
}


void add_adj(char **mineboard, int COLS, int ROWS)
{
    int i, j;
```

```c

    for (i = 0; i < COLS; i++)
        for (j = 0; j < ROWS; j++)
            if (!is_mine(mineboard, i, j))
                mineboard[i][j] = adj_mines(mineboard, i, j, COLS, ROWS) + ’0’;
}


bool is_mine(char **mineboard, int row, int col)
{
    return (mineboard[row][col] == MINE) ? true : false;
}

bool outof_bounds(int row, int col, int COLS, int ROWS)
{
    return (row < 0 || row > COLS-1 || col < 0 || col > ROWS-1) ? true : false;
}



int8_t adj_mines(char **mineboard, int row, int col, int COLS, int ROWS)
{
    int8_t numAdj = 0;

    if (!outof_bounds(row, col - 1, COLS, ROWS)      && mineboard[row][col-1]     == MINE) numAdj++; // North
    if (!outof_bounds(row, col + 1, COLS, ROWS)      && mineboard[row][col+1]     == MINE) numAdj++; // South
    if (!outof_bounds(row + 1, col, COLS, ROWS)      && mineboard[row+1][col]     == MINE) numAdj++; // East
    if (!outof_bounds(row - 1, col, COLS, ROWS)      && mineboard[row-1][col]     == MINE) numAdj++; // West
    if (!outof_bounds(row + 1, col - 1, COLS, ROWS)  && mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
    if (!outof_bounds(row - 1, col - 1, COLS, ROWS)  && mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
    if (!outof_bounds(row + 1, col + 1, COLS, ROWS)  && mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
    if (!outof_bounds(row - 1, col + 1, COLS, ROWS)  && mineboard[row-1][col+1]  == MINE) numAdj++; // South-West

    return numAdj;
}


void fill_spaces(char **mineboard, int COLS, int ROWS, int NMINES)
```

```c
{
    int i, j;

    for (i = 0; i < COLS; i++)
        for (j = 0; j < ROWS; j++)
            if (mineboard[i][j] != MINE && mineboard[i][j] =
= '0')
                mineboard[i][j] = '-';
}
```

5.5 gameplay.h

```c
#ifndef GAMEPLAY_H
#define GAMEPLAY_H

#include "minesweeper.h"

#define DEFUSEKEY 'd'

void selection(struct _win_st*, char **, char **, int, int,
    int);
bool transfer(char **, char **, int, int, int, int *);
void reveal(struct _win_st*, char **, int, int);
bool defused(char **, char **, int, int, int, int *);

#endif
```

5.6 settings.c

```c
#include "settings.h"

void set_mode(WINDOW *menuWin) // loop
{
    char mode;
    mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
);
    wrefresh(menuWin);
    scanw("%c", &mode);
    mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
): ") + 1, "%c", mode);
    wrefresh(menuWin);
    mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
spyrosROUM!!!! :-DDDD
    wrefresh(menuWin);

    switch (mode)
    {
        case 'k':
        case 'K':
            mvwprintw(menuWin, 2, 1, "Keyboard mode");
            wrefresh(menuWin);
            break;
        case 't':
        case 'T':
            mvwprintw(menuWin, 2, 1, "Text mode");
            wrefresh(menuWin);
            break;
        default:
            break;
    }
}


int set_cols(WINDOW *menuWin, int xMax)
{
    int COLS;

    do
    {
        mvwprintw(menuWin, 1, 1, "Columns (Max = %d): ", xMax-12);
        wrefresh(menuWin);
        scanw("%d", &COLS);
        mvwprintw(menuWin, 1, strlen("Columns (Max = XXX): ") + 1, "%d", COLS);
        wrefresh(menuWin);
    } while (COLS < 5 || COLS > xMax - 12);

    return COLS;
}


int set_rows(WINDOW *menuWin, int yMax)
{
    int ROWS;

    do
    {
        mvwprintw(menuWin, 2, 1, "Rows (Max = %d): ", yMax-12);
        wrefresh(menuWin);
        scanw("%d", &ROWS);
        mvwprintw(menuWin, 2, strlen("Rows (Max = YYY): ") + 1, "%d", ROWS);
        wrefresh(menuWin);
    } while (ROWS < 5 || ROWS > yMax - 12);

    return ROWS;
}

```

```c

int set_nmines(WINDOW *menuWin, int DIMENSIONS)
{
    int NMINES;

    do
    {
        mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ", DIMENSIONS-10); // -10 so the player has a chance to win
        wrefresh(menuWin);
        scanw("%d", &NMINES);
        mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ") + 1, "%d", NMINES);
        wrefresh(menuWin);
    } while (NMINES < 1 || NMINES > DIMENSIONS-10);

    return NMINES;
}
```

5.7 settings.h

```c
#ifndef SETTINGS_H
#define SETTINGS_H

#include "minesweeper.h"

void set_mode(struct _win_st*);
int set_cols(struct _win_st*, int);
int set_rows(struct _win_st*, int);
int set_nmines(struct _win_st*, int);

#endif
```

5.8 outputs.c

```c
#include "outputs.h"

void game_won(WINDOW *gameWin, char **mineboard, int yMiddle, int xMiddle)
{
    wclear(gameWin);
    mvwprintw(gameWin, yMiddle-11, xMiddle-18, "You defused all the mines!");
    mvwprintw(gameWin, yMiddle-10, xMiddle-10, "You won :)");
    wrefresh(gameWin);
    wclear(gameWin);
}


```

```c
void game_over(WINDOW *gameWin, char **mineboard, int yMiddle, int xMiddle)
{
    wclear(gameWin);
    mvwprintw(gameWin, yMiddle-11, xMiddle-11, "You hit a mine!");
    mvwprintw(gameWin, yMiddle-10, xMiddle-10, "Game over :(");
    wrefresh(gameWin);
    wclear(gameWin);
}


void print_board(WINDOW *gameWin, char **mineboard, int COLS, int ROWS)
{
    int i, j;

    for (i = 0; i < ROWS; i++)
    {
        for (j = 0; j < COLS; j++)
        {
            mvwaddch(gameWin, j+1, i+1, mineboard[i][j]);
            wrefresh(gameWin);
        }
    }
}


void filewrite(char **mineboard, int COLS, int ROWS, int hitCol, int hitRow, const char *status)
{
    int i, j;
    FILE *mnsOut = fopen("mnsout.txt", "w");

    if (mnsOut == NULL)
    {
        mvprintw(1, 1, "Error opening file, exiting...");
        exit(EXIT_FAILURE);
    }
    else
    {
        strcmp(status, "won")
            ? fprintf(mnsOut, "Mine hit at position (%d, %d)\n\n", hitCol, hitRow)
            : fprintf(mnsOut, "Last mine defused at position (%d, %d)\n\n", hitCol, hitRow);
        fprintf(mnsOut, "Board overview\n\n");

        for (i = 0; i < ROWS; i++)
        {
            for (j = 0; j < COLS; j++)
                fprintf(mnsOut, "%c ", mineboard[j][i]);
            fprintf(mnsOut, "\n");
        }

        mvprintw(1, 1, "Session written to file");
        refresh();
    }

    fclose(mnsOut);
}
```

5.9 outputs.h

```
#ifndef OUTPUTS_H
#define OUTPUTS_H

#include "minesweeper.h"

void game_won(struct _win_st*, char **, int, int);
void game_over(struct _win_st*, char **, int, int);

void print_board(struct _win_st*, char **, int, int);
void filewrite(char **, int, int, int, int, const char *);

#endif
```

5.10 Διάγραμμα ροής

5.11 Περιγραφή υλοποιήσης

6 Διευκρινήσεις

7 Εργαλεία

- Editors: Visual Studio Code, Vim
- Compiler: gcc
- Shell: zsh
- OS: Arch Linux
- Συγγραφή: LaTeX