Εργασία 5: Πίνακες - Δείκτες - Αρχεία

Χρήστος Μαργιώλης - Εργαστηριακό τμήμα 9

Ιανουάριος 2020

Περιεχόμενα

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

Το κάθε πρόγραμμα, είναι δομημένο ως εξής: Υπάρχουν πέντε φάκελοι, καθώς και ένα Makefile στο top directory. Στον φάκελο src βρίσκονται οι πηγαίοι κώδικες, στον include τα header files, στον obj τα object files και στον bin το εκτελέσιμο αρχείο. Στον φάκελο txt υπάρχουν τα text files που διαβάζονται οι γράφονται από το κάθε πρόγραμμα. Το Makefile είναι υπεύθυνο για την μεταγλώττιση όλων των αρχείων μαζί, και την τοποθέτησή τους στους κατάλληλους φακέλους, την εκτέλεση των προγραμμάτων, καθώς και τον καθαρισμό των φακέλων (διαγράφει τα object files και το εκτελέσιμο με την εντολή make clean).

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);
}
```

```c
void print_combs(int *arr, int N, int x1, int x2, int y1, int y2)
{
    int *currComb = (int *)malloc(N * sizeof(int));
    int *freqArr = (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, freqArr, 0, N-1, 0, &printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2, N);
        print_other(N, unFrstCond, unScndCondOnly, printed, arr, freqArr);
    }

    free(currComb);
    free(freqArr);
}


void combinations(int *arr, int *currComb, int *freqArr, int start, int end, int index, int *printed, int *unFrstCond, int *unScndCondOnly, int x1, int x2, int y1, int y2, int N)
{
    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)
                {
                    frequency(freqArr, currComb, arr, N);
                    (*printed)++;
                    printf("\n");
                }
            }
        }
        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, freqArr, i+1, end, index+1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2, N);
    }

}


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;
}


void frequency(int *freqArr, int *currComb, int *arr, int N)
{
    int pos, i;

    for (i = 0; i < COMBSN; i++)
    {
        pos = find_pos(arr, N, *(currComb + i));
        (*(freqArr + pos))++;
```

```c
    }
}


long int combinations_count(int N)
{
    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,int *arr, int *freqArr)
{
    int i;

    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\n", printed);

    for (i = 0; i < N; i++)
        printf("%d appeared %d times\n", *(arr + i), *(freqArr + i));

}
```

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, int, int);

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

void frequency(int *, int *, int *, int);
long int combinations_count(int);
long double factorial(int);
void print_other(int, int, int, int, int *, int *);

#endif
```

2.4 arrhandler.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;
```

```c
    *b = temp;
}


int find_pos(int *arr, int numIter, int val)
{
    int pos, i;

    for (i = 0; i < numIter; i++)
        if (val == *(arr + i))
            pos = i;

    return pos;
}
```

2.5 arrhandler.h

```c
#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 *);

int find_pos(int *, int, int);

#endif
```

2.6 Διάγραμμα ροής

2.7 Περιγραφή υλοποιήσης

3 kcombinations - συνδυασμοί με K

3.1 main.c

```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

```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);
        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 *freqArr = (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, freqArr, 0, N-1, 0, N, K, &printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
        print_other(N, K, unFrstCond, unScndCondOnly, printed, arr, freqArr);
    }

    free(currComb);
    free(freqArr);
}


void combinations(int *arr, int *currComb, int *freqArr, int start, int end, int index, int N, 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)
                {
                    frequency(freqArr, currComb, arr, N);
                    (*printed)++;
                    printf("\n");
                }
            }
        }
        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, freqArr, i+1, end, index
    +1, N, 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(int *freqArr, int *currComb, int *arr, int N)
{
    int pos, i;

    for (i = 0; i < N; i++)
    {
        pos = find_pos(arr, N, *(currComb + i));
        (*(freqArr + pos))++;
    }
}


long int combinations_count(int N, int K)
{
    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, int *arr, int *freqArr)
{
    int i;

    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);
```

```c
    printf("Printed combinations: %d\n\n", printed);

    for (i = 0; i < N; i++)
        printf("%d appeared %d times\n", *(arr + i), *(freqArr + i));
}
```

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, int, int);

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

int frequency(int *, int *, int *, int);
long int combinations_count(int, int);
long double factorial(int);
void print_other(int, int, int, int, int, int *, int *);

#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;
}


int find_pos(int *arr, int numIter, int val)
{
    int pos, i;

    for (i = 0; i < numIter; i++)
        if (val == *(arr + i))
            pos = i;

    return pos;
}
```

3.5 arrhandler.h

```c
#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

```c
#include "fcombinations.h"

int main(int argc, char **argv)
{
    int *arr, N, x1, x2, y1, y2;
    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
    {
        N = get_n(dataFile);
        arr = fill_array(N, dataFile);
        quicksort(arr, 0, N-1);
        x_pair(&x1, &x2, dataFile);
        y_pair(&y1, &y2, dataFile);
        print_combs(arr, N, x1, x2, y1, y2);
    }

    fclose(dataFile);
    free(arr);

    return 0;
}
```

4.2 fcombinations.c

```c
#include "fcombinations.h"

int get_n(FILE *dataFile)
{
    int N;

    do
    {
        fscanf(dataFile, "%d\n", &N);

        printf("%d\n", N);
    } while (N <= 6 || N > 49);

    return N;
}


void x_pair(int *x1, int *x2, FILE *dataFile)
{
    do
    {
        fscanf(dataFile, "%d\n", x1);
        fscanf(dataFile, "%d\n", x2);
    } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
}


void y_pair(int *y1, int *y2, FILE *dataFile)
{
    do
    {
        fscanf(dataFile, "%d\n", y1);
        fscanf(dataFile, "%d\n", 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 *freqArr = (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, freqArr, 0, N-1, 0, &printed, &unFrstCond, &unScndCondOnly, x1, x2, y1, y2, N);
        print_other(N, unFrstCond, unScndCondOnly, printed, arr, freqArr);
    }

    free(currComb);
    free(freqArr);
}


void combinations(int *arr, int *currComb, int *freqArr, int start, int end, int index, int *printed, int *unFrstCond, int *unScndCondOnly, int x1, int x2, int y1, int y2, int N)
{
    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)
                {
                    frequency(freqArr, currComb, arr, N);
                    (*printed)++;
                    printf("\n");
                }
            }
        }
        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, freqArr, i+1, end, index+1, printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2, N);
    }

}


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;
}


void frequency(int *freqArr, int *currComb, int *arr, int N)
{
    int pos, i;

    for (i = 0; i < COMBSN; i++)
    {
        pos = find_pos(arr, N, *(currComb + i));
        (*(freqArr + pos))++;
    }
}


long int combinations_count(int N)
{
    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,int *arr, int *freqArr)
```

```c
{
    int i;

    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\n", printed);

    for (i = 0; i < N; i++)
        printf("%d appeared %d times\n", *(arr + i), *(freqArr + i));

}
```

4.3 fcombinations.h

```c
#ifndef COMBINATIONS_H
#define COMBINATIONS_H

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

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

#define COMBSN 6

int get_n(FILE *);

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

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

void frequency(int *, int *, int *, int);
long int combinations_count(int);
long double factorial(int);
void print_other(int, int, int, int, int *, int *);

#endif
```

4.4 arrhandler.c

```c
#include "arrhandler.h"

int *fill_array(int N, FILE *dataFile)
{
    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
        {
            fscanf(dataFile, "%d\n", &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++; }
            }
        } 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;
}


int find_pos(int *arr, int numIter, int val)
{
    int pos, i;

    for (i = 0; i < numIter; i++)
        if (val == *(arr + i))
            pos = i;

    return pos;
}
```

4.5 arrhandler.h

```
#ifndef ARRHANDLER_H
#define ARRHANDLER_H

#include "fcombinations.h"

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

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

```c

int find_pos(int *, int, int);

#endif
```

4.6 Διάγραμμα ροής

4.7 Περιγραφή υλοποιήσης

5 minesweeper - ναρκαλιευτής

5.1 main.c

```c
#include "minesweeper.h"

int main(int argc, char **argv)
{

    main_win();
    options_menu();
    int yMax, xMax;
    WINDOW *menuWin = menu_win(&yMax, &xMax);

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

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

    free(dispboard);
    free(mineboard);

    endwin();

    return 0;
}
```

5.2 minesweeper.c

```c
#include "minesweeper.h"

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);

    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] = BLANK;
}


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);
    }

    return mineboard;
```

```c
}


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() % ROWS;
        hRand = rand() % COLS;
        mineboard[wRand][hRand] = MINE;
    }
}


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

    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)
{
    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 "navigation.h"
#include "outputs.h"
#include "wins.h"

#define BLANK ’ ’
#define MINE ’*’
#define CLEAR "                                "
#define SPACE ’ ’
#define ENTER ’\n’
#define OPEN_LOWER ’o’
#define OPEN_UPPER ’O’
#define FLAG ’F’
#define FLAG_LOWER ’f’
#define FLAG_UPPER ’F’
#define DEFUSE_LOWER ’g’
#define DEFUSE_UPPER ’G’
#define DEFUSED ’D’
#define QUIT ’q’

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

```
#include "gameplay.h"

void play_minesweeper(WINDOW *gameWin, char **dispboard,
    char **mineboard, int COLS, int ROWS, int NMINES)
{
    int mboardXLoc = 0, mboardYLoc = 0;
    bool gameOver = false, cantFlag = false;
    int numDefused = 0;
    int yMax, xMax, yMiddle, xMiddle;
    char move;
    getmaxyx(stdscr, yMax, xMax);
    yMiddle = yMax / 2;
    xMiddle = xMax / 2;

    print_board(gameWin, dispboard, COLS, ROWS);

```

```
do
{
    navigate(gameWin, dispboard, &move, &mboardXLoc, &
mboardYLoc);

    if (move == ENTER || move == SPACE || move ==
OPEN_LOWER || move == OPEN_UPPER) // handle cell opening
    {
        transfer(dispboard, mineboard, mboardYLoc,
mboardXLoc);
        reveal(gameWin, dispboard, mboardYLoc,
mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
        cantFlag = true;
        if (dispboard[mboardYLoc][mboardXLoc] == MINE)
gameOver = true;
    }
    else if (move == FLAG_LOWER || move == FLAG_UPPER)
// handle falgs
    {
        if (dispboard[mboardYLoc][mboardXLoc] == FLAG)
dispboard[mboardYLoc][mboardXLoc] = BLANK; // undo flag
        else if (dispboard[mboardYLoc][mboardXLoc] !=
FLAG && dispboard[mboardYLoc][mboardXLoc] != BLANK)
continue; // dont flag an already opened mine
        else dispboard[mboardYLoc][mboardXLoc] = FLAG;
// flag if not flagged already
        reveal(gameWin, dispboard, mboardYLoc,
mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
    }
    else if (move == DEFUSE_LOWER || move ==
DEFUSE_UPPER) // check for defuse
    {
        if (dispboard[mboardYLoc][mboardXLoc] == FLAG &&
 mineboard[mboardYLoc][mboardXLoc] == MINE) // is_mine
replace
        {
            numDefused++;
            refresh();
            dispboard[mboardYLoc][mboardXLoc] =
mineboard[mboardYLoc][mboardXLoc] = DEFUSED;
            reveal(gameWin, dispboard, mboardYLoc,
mboardXLoc, mboardYLoc + 1, 3*mboardXLoc + 2);
        }
        else if (dispboard[mboardYLoc][mboardXLoc] ==
FLAG && mineboard[mboardYLoc][mboardXLoc] != MINE)
gameOver = true; // handle false defusal
    }

    mvprintw(1, xMiddle-8, "Defused mines: %d/%d",
numDefused, NMINES);
```

```

    } while (((mboardYLoc >= 0 && mboardYLoc < COLS) && (mboardXLoc >= 0 && mboardXLoc < ROWS)) &&
                numDefused < NMINES && !gameOver && move != QUIT);

    if (gameOver == true)
    {
        game_over(gameWin, mineboard, yMiddle, xMiddle);
        getchar();
        print_board(gameWin, mineboard, COLS, ROWS);
        filewrite(mineboard, COLS, ROWS, mboardXLoc, mboardYLoc, "lost");
    }

    if (numDefused == NMINES)
    {
        game_won(gameWin, yMiddle, xMiddle);
        getchar();
        filewrite(mineboard, COLS, ROWS, mboardXLoc, mboardYLoc, "won");
    }
}


void transfer(char **dispboard, char **mineboard, int mboardYLoc, int mboardXLoc)
{
    dispboard[mboardYLoc][mboardXLoc] = mineboard[mboardYLoc][mboardXLoc];
}


void reveal(WINDOW *gameWin, char **dispboard, int mboardYLoc, int mboardXLoc, int yLoc, int xLoc)
{
    mvwaddch(gameWin, yLoc, xLoc, dispboard[mboardYLoc][mboardXLoc]);
    wrefresh(gameWin);
}


void flag_handler()
{

}


bool is_flagged()
{
```

```c

}


bool is_defused(char **dispboard, char **mineboard, int mboardYLoc, int mboardXLoc)
{
    return ((dispboard[mboardYLoc][mboardXLoc] == DEFUSED)) ? true : false;
}
```

5.5 gameplay.h

```c
#ifndef GAMEPLAY_H
#define GAMEPLAY_H

#include "minesweeper.h"

void play_minesweeper(struct _win_st*, char **, char **, int, int, int);
void transfer(char **, char **, int, int);
void reveal(struct _win_st*, char **, int, int, int, int);
void flag_handler();
bool is_flagged();
bool is_defused(char **, char **, int, int);

#endif
```

5.6 navigation.c

```c
#include "navigation.h"

void navigate(WINDOW *gameWin, char **mineboard, char *move, int *mboardXLoc, int *mboardYLoc)
{
    int yMax, xMax;
    static int yLoc = 1, xLoc = 2;
    getmaxyx(gameWin, yMax, xMax);
    wmove(gameWin, yLoc-1, xLoc);

    update_curs(gameWin, yLoc, xLoc);
    *mboardYLoc = yLoc-1;
    *mboardXLoc = (xLoc-2)/3;
    mvprintw(1, 1, "Current position: (%d, %d) ", *mboardXLoc+1, *mboardYLoc+1);
    refresh();
    getmv(gameWin, move, &yLoc, &xLoc, yMax, xMax);
}


```

```
void getmv(WINDOW *gameWin, char *move, int *yLoc, int *xLoc
    , int yMax, int xMax)
{
    *move = wgetch(gameWin);
    switch (*move) // vim keys support!!
    {
        case ’k’: case ’K’:
        case ’w’: case ’W’:
            mvup(yLoc, xLoc);
            break;
        case ’j’: case ’J’:
        case ’s’: case ’S’:
            mvdown(yLoc, xLoc, yMax, xMax);
            break;
        case ’h’: case ’H’:
        case ’a’: case ’A’:
            mvleft(yLoc, xLoc);
            break;
        case ’l’: case ’L’:
        case ’d’: case ’D’:
            mvright(yLoc, xLoc, yMax, xMax);
            break;
        default: break;
    }
}


void mvup(int *yLoc, int *xLoc)
{
    (*yLoc)--;
    if (*yLoc < 1) *yLoc = 1;
}


void mvdown(int *yLoc, int *xLoc, int yMax, int xMax)
{
    (*yLoc)++;
    if (*yLoc > yMax-2) *yLoc = yMax-2;
}


void mvleft(int *yLoc, int *xLoc)
{
    *xLoc -= 3;
    if (*xLoc < 2) *xLoc = 2;
}


void mvright(int *yLoc, int *xLoc, int yMax, int xMax)
{
```

```c
    *xLoc += 3;
    if (*xLoc > xMax-3) *xLoc = xMax-3;
}


void update_curs(WINDOW *gameWin, int yLoc, int xLoc)
{
    wmove(gameWin, yLoc, xLoc);
}
```

5.7 navigation.h

```c
#ifndef NAVIGATION_H
#define NAVIGATION_H

#include "minesweeper.h"

void navigate(struct _win_st*, char **, char *, int *, int *
    );
void getmv(struct _win_st*, char *, int *, int *, int, int);
void mvup(int *, int *);
void mvdown(int *, int *, int, int);
void mvleft(int *, int *);
void mvright(int *, int *, int, int);
void update_curs(struct _win_st*, int, int);

#endif
```

5.8 settings.c

```c
#include "settings.h"

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

    do
    {
        mvwprintw(menuWin, 1, 1, "Columns (Min = 5, Max = %d
    ): ", (xMax-2)/3 - 2);
        wrefresh(menuWin);
        scanw("%d", &COLS);
        mvwprintw(menuWin, 1, COLS_CHAR_LENGTH, "%d", COLS);
        wrefresh(menuWin);
    } while (COLS < 5 || COLS > (xMax-2)/3 - 2);

    return COLS;
}


```

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

    do
    {
        mvwprintw(menuWin, 2, 1, "Rows (Min = 5, Max = %d): ", yMax-14);
        wrefresh(menuWin);
        scanw("%d", &ROWS);
        mvwprintw(menuWin, 2, ROWS_CHAR_LENGTH, "%d", ROWS);
        wrefresh(menuWin);
    } while (ROWS < 5 || ROWS > yMax - 14);

    return ROWS;
}


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, NMINES_CHAR_LENGTH, "%d", NMINES);
        wrefresh(menuWin);
    } while (NMINES < 1 || NMINES > DIMENSIONS-15);

    return NMINES;
}
```

5.9 settings.h

```c
#ifndef SETTINGS_H
#define SETTINGS_H

#include "minesweeper.h"

#define COLS_CHAR_LENGTH strlen("Columns (Min = 5, Max = XXX): ") + 1
#define ROWS_CHAR_LENGTH strlen("Rows (Min = 5, Max = YYY): ") + 1
#define NMINES_CHAR_LENGTH strlen("Mines (Max = MMM): ") + 1

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

#endif
```

5.10 outputs.c

```c
#include "outputs.h"

void print_board(WINDOW *gameWin, char **board, int COLS,
    int ROWS)
{
    int i, j, x, y = 1;

    print_grid(gameWin, ROWS, COLS);

    for (i = 0; i < ROWS; i++)
    {
        x = 2;
        for (j = 0; j < COLS; j++)
        {
            wattron(gameWin, A_BOLD);
            mvwaddch(gameWin, y, x, board[i][j]);
            x += 3;
        }
        y++;
    }

    wrefresh(gameWin);
    wattron(gameWin, A_BOLD);
}


void print_grid(WINDOW *gameWin, int ROWS, int COLS)
{
    int i, j;

    for (i = 1; i <= ROWS; i++)
    {
        wmove(gameWin, i, 1);
        for (j = 0; j < COLS; j++)
            wprintw(gameWin, "[ ]");
    }

    wrefresh(gameWin);
}


void filewrite(char **mineboard, int COLS, int ROWS, int
    hitRow, int hitCol, const char *status)
{
    int i, j;
    FILE *mnsOut = fopen("txt/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", hitRow+1, hitCol+1)
            : fprintf(mnsOut, "Last mine defused at position (%d, %d)\n\n", hitRow+1, hitCol+1);
        fprintf(mnsOut, "Board overview\n\n");

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

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

    fclose(mnsOut);
}


void game_won(WINDOW *gameWin, int yMiddle, int xMiddle)
{
    wclear(gameWin);
    wrefresh(gameWin);
    wattron(stdscr, A_BOLD);
    mvwprintw(stdscr, yMiddle-2, xMiddle-11, "You defused all the mines!");
    mvwprintw(stdscr, yMiddle-1, xMiddle-3, "You won :)");
    mvwprintw(stdscr, yMiddle, xMiddle-11, "Press any key to continue");
    refresh();
    wattroff(stdscr, A_BOLD);
}


```

```c
void game_over(WINDOW *gameWin, char **mineboard, int yMiddle, int xMiddle)
{
    wclear(gameWin);
    wrefresh(gameWin);
    wattron(stdscr, A_BOLD);
    mvwprintw(stdscr, yMiddle-2, xMiddle-24, "You hit a mine! (or tried to defuse the wrong cell)");
    mvwprintw(stdscr, yMiddle-1, xMiddle-4, "Game over :(");
    mvwprintw(stdscr, yMiddle, xMiddle-11, "Press any key to continue");
    refresh();
    wattroff(stdscr, A_BOLD);
}
```

5.11 outputs.h

```c
#ifndef OUTPUTS_H
#define OUTPUTS_H

#include "minesweeper.h"

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

#endif
```

5.12 wins.c

```c
#include "wins.h"

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

    WINDOW *mainWin = newwin(0, 0, 0, 0);
    wattron(mainWin, A_BOLD);
    box(mainWin, 0, 0);
    refresh();
    wrefresh(mainWin);
    wattroff(mainWin, A_BOLD);
}


WINDOW *menu_win(int *yMax, int *xMax)
```

```c
{
    int numSettings = 3;
    getmaxyx(stdscr, *yMax, *xMax);
    WINDOW *menuWin = newwin(numSettings+2, *xMax-8, *yMax-8, 4);
    wattron(menuWin, A_BOLD);
    box(menuWin, 0, 0);
    wrefresh(menuWin);
    wattroff(menuWin, A_BOLD);
    return menuWin;
}


WINDOW *game_win(int COLS, int ROWS, int NMINES)
{
    int winRows = ROWS + 2;
    int winCols = COLS*3 + 2;
    WINDOW *gameWin = newwin(winRows, winCols, 2, 4);
    wattron(gameWin, A_BOLD);
    box(gameWin, 0, 0);
    wrefresh(gameWin);
    wattroff(gameWin, A_BOLD);
    return gameWin;
}


void options_menu()
{
    int yMax = getmaxy(stdscr);
    mvprintw(yMax-3, 5, "q Quit     w/k Move up     s/j Move down       a/h Move Left       d/l Move Right      [ENTER]/o Open cell");
    mvprintw(yMax-2, 5, "f Flag cell     g Defuse (if flagged only)");
    refresh();
}
```

5.13 wins.h

```c
#ifndef WINS_H
#define WINS_H

#include "minesweeper.h"

void main_win();
WINDOW *menu_win(int *, int *);
WINDOW *game_win(int, int, int);
void options_menu();
void options_win();

```

```
#endif
```

5.14 Διάγραμμα ροής

5.15 Περιγραφή υλοποιήσης

Ο ναρκαλιευτής αυτός χρησιμοποιεί την βιβλιοθήκη ncurses και είναι δομημένος ως εξής: Από το main.c καλούνται αρχικά οι συναρτήσεις δημιουργίας των παραθύρων που θα εμφανιστούν στην οθόνη και στην συνέχεια καλούνται οι συναρτήσεις δημιουργίας των πινάκων $M \times N$, για το ναρκοπέδιο και για τον πίνακα που έχει "κρυμμένα" τα κελιά αντίστοιχα. Τέλος από την main καλείται η συνάρτηση που θα ξεκινήσει το παιχνίδι.

Οι συναρτήσεις για τις στήλες, γραμμές, και αριθμό των ναρκών βρίσκονται στο settings.h

Στο minesweeper.c εκτελούνται όλες οι συναρτήσεις δημιουργίας πινάκων, τοποθέτησεις ναρκών, μέτρημα των βομβών στα γειτονικά κελιά, καθώς και γέμισμα των κενών θέσεων τους.

Έπειτα, στο gameplay.c εκτελείται το παιχνίδι - αρχικά τυπώνεται ο πίνακας και το περίγραμμα που υπάρχει ανάμεσα σε κάθε κελί ώστε να είναι πιο εμφανίσιμο και πιο εύχρηστο το παιχνίδι. Προκειμένου τα κελιά να τοποθετηθούν στις κατάλληλες θέσεις στον πίνακα, δηλαδή να είναι ανάμεσα στα [ ], τα στοιχεία των πινάκων τοποθετούνται κάθε φορά με απόσταση 2 στον κάθετο άξονα και 3 στον οριζόντιο το ένα από το άλλο. Με αυτά τα 2 νούμερα προκύπτουν και 2 τύποι, οι οποίοι βοηθάνε στην σωστή προσπέλαση των στοιχείων των πινάκων κατά την διάρκεια του παιχνιδιού, και στον υπολογισμό των διαστάσεων του παραθύρου που εμφανίζεται το πεδίο. Οι τύποι είναι οι εξής

$$x = rows + 2 \tag{1}$$

$$y = columns \times 3 + 2 \tag{2}$$

Αφού τυπωθεί στην οθόνη ο πίνακας με κρυμμένα τα στοιχεία του, το οποίο είναι στην ουσία ένας $M \times N$ πίνακας γεμισμένος με κενά, ξεκινάει το βασικό loop του παιχνιδιού, στο οποίο ο χρήστης μετακινείται από κελί σε κελί, επιλέγει την κίνηση που θέλει να κάνει πάνω σε κάθε κελί, και είτε χάνει ή νικάει. Προκειμένου να λειτουργήσει κάτι τέτοιο, μέσα στο loop γίνονται οι εξής λειτουργίες: Αρχικά ο κέρσορας μετακίνεται κάθε φορά που ο και χρήστης μετακινείται ώστε να μπορεί να δει σε ποιο κελί βρίσκεται και ο χρήστης πρέπει έχει

Λόγω του ότι το πρόγραμμα περιέχει πολλές μεταβλητές θεώρησα καλύτερο να εστιάσω στην λειτουργία του προγράμματος και όχι τόσο στο τι συμβολίζει η κάθε μεταβλητή.

6 Διευκρινήσεις

7 Εργαλεία

- Editors: Visual Studio Code, NVim

- Compiler: gcc
- Shell: zsh
- OS: Arch Linux
- Συγγραφή: LaTeX