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

2 combinations - συνδυασμοί

2.1 main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "combinations.h"
#include "arrhandler.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 <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "combinations.h"
#include "arrhandler.h"
#include "ccolors.h"


int get_n()
{
    int N;

    do
```

```
14      {
15          system("clear||cls");
16          printf("N (6 < N <= 49): ");
17          scanf("%d", &N);
18      } while (N <= 6 || N > 49);
19
20      system("clear||cls");
21
22      return N;
23  }
24
25
26  void x_pair(int *x1, int *x2)
27  {
28      do
29      {
30          printf("x1: ");
31          scanf("%d", x1);
32          printf("x2: ");
33          scanf("%d", x2);
34      } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
35  }
36
37
38  void y_pair(int *y1, int *y2)
39  {
40      do
41      {
42          printf("y1: ");
43          scanf("%d", y1);
44          printf("y2: ");
45          scanf("%d", y2);
46      } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
47  }
48
49
50  void print_combs(int *arr, int N, int x1, int x2, int y1,
        int y2)
51  {
52      int *currComb = (int *)malloc(N * sizeof(int));
53      int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
54
55      if (currComb == NULL)
56      {
57          set_color(BOLD_RED);
58          printf("Error! Not enough memory, exiting...\n");
59          exit(EXIT_FAILURE);
60          set_color(STANDARD);
61      }
62      else
```

```
63      {
64          combinations(arr, currComb, 0, N-1, 0, &printed, &
        unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
65          print_other(N, unFrstCond, unScndCondOnly, printed);
66      }
67
68      free(currComb);
69  }
70
71
72  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)
73  {
74      int i, j;
75
76      if (index == COMBSN)
77      {
78          for (j = 0; j < COMBSN; j++)
79          {
80              if (even_calc(currComb, x1, x2) && sum_comb_calc
        (currComb, y1, y2))
81              {
82                  printf("%d ", *(currComb + j));
83                  if (j == COMBSN - 1) { (*printed)++; printf(
        "\n"); }
84              } // add freq
85          }
86          if (!even_calc(currComb, x1, x2) && sum_comb_calc(
        currComb, y1, y2)) (*unFrstCond)++;
87          if (!sum_comb_calc(currComb, y1, y2)) (*
        unScndCondOnly)++;
88          return;
89      }
90
91      for (i = start; i <= end && end-i+1 >= COMBSN-index; i++
        )
92      {
93          *(currComb + index) = *(arr + i);
94          combinations(arr, currComb, i+1, end, index+1,
        printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
95      }
96  }
97
98
99  bool even_calc(int *arr, int x1, int x2)
100 {
101     int numEven = 0, i;
102
103     for (i = 0; i < COMBSN; i++)
```

```
104         if (*(arr + i) % 2 == 0) numEven++;
105
106     return (numEven >= x1 && numEven <= x2) ? true : false;
107 }
108
109
110 bool sum_comb_calc(int *arr, int y1, int y2)
111 {
112     int sumNums = 0, i;
113
114     for (i = 0; i < COMBSN; i++)
115         sumNums += *(arr + i);
116
117     return (sumNums >= y1 && sumNums <= y2) ? true : false;
118 }
119
120
121 int frequency()
122 {
123
124 }
125
126
127 long int combinations_count(int N) // wtf ???????
128 {
129     return (factorial(N) / (factorial(COMBSN) * factorial(N
    - COMBSN)));
130 }
131
132
133 long double factorial(int num)
134 {
135     int i;
136     long double fac;
137     if (num == 0) return -1;
138     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
139     return fac;
140 }
141
142
143 void print_other(int N, int unFrstCond, int unScndCondOnly,
    int printed)
144 {
145     printf("\nTotal number of combinations %d to %d: %ld\n",
     N, COMBSN, combinations_count(N));
146     printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
147     printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
148     printf("Printed combinations: %d\n", printed);
```

```
149 }
```

2.3 combinations.h

```
1  #ifndef COMBINATIONS_H
2  #define COMBINATIONS_H
3
4  #include <stdbool.h>
5
6  #define COMBSN 6
7
8  void x_pair(int *, int *);
9  void y_pair(int *, int *);
10
11 void print_combs(int *, int, int, int, int, int);
12 void combinations(int *, int *, int, int, int, int *, int *,
       int *, int, int, int, int);
13
14 bool even_calc(int *, int, int);
15 bool sum_comb_calc(int *, int, int);
16
17 int frequency();
18 long int combinations_count(int);
19 long double factorial(int);
20 void print_other(int, int, int, int); // add freq
21
22 #endif
```

2.4 arrhandler.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "arrhandler.h"
4  #include "combinations.h"
5  #include "ccolors.h"
6
7
8  int *fill_array(int N)
9  {
10     int num, i = 0;
11     int *arr = (int *)malloc(N * sizeof(int));
12
13     if (arr == NULL)
14     {
15         set_color(BOLD_RED);
16         printf("Error! Not enough memory, exiting...\n");
17         exit(EXIT_FAILURE);
18         set_color(STANDARD);
19     }
20     else
```

```
21      {
22          do
23          {
24              printf("arr[%d]: ", i);
25              scanf("%d", &num);
26
27              if (num >= 1 && num <= 49)
28              {
29                  if (i == 0) { *(arr + i) = num; i++; }
30                  else
31                  {
32                      if (!exists_in_array(arr, N, num)) { *(
    arr + i) = num; i++; }
33                      else printf("Give a different number.\n"
    );
34                  }
35              }
36              else printf("Give a number in [1, 49].\n");
37          } while (i < N);
38      }
39
40      return arr;
41  }
42
43
44  bool exists_in_array(int *arr, int N, int num)
45  {
46      int *arrEnd = arr + (N - 1);
47      while (arr <= arrEnd && *arr != num) arr++;
48      return (arr <= arrEnd) ? true : false;
49  }
50
51
52  void quicksort(int *arr, int low, int high)
53  {
54      if (low < high)
55      {
56          int partIndex = partition(arr, low, high);
57          quicksort(arr, low, partIndex - 1);
58          quicksort(arr, partIndex + 1, high);
59      }
60  }
61
62
63  int partition(int *arr, int low, int high)
64  {
65      int pivot = *(arr + high);
66      int i = (low - 1), j;
67
68      for (j = low; j <= high - 1; j++)
```

```
69          if (*(arr + j) < pivot)
70              swap(arr + ++i, arr + j);
71
72      swap(arr + (i + 1), arr + high);
73      return (i + 1);
74  }
75
76
77  void swap(int *a, int *b)
78  {
79      int temp = *a;
80      *a = *b;
81      *b = temp;
82  }
```

2.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include <stdbool.h>
5
6  int *fill_array(int);
7  bool exists_in_array(int *, int, int);
8
9  void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

2.6 Περιγραφή υλοποιήσης

3 kcombinations - συνδυασμοί με K

3.1 main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "kcombinations.h"
4  #include "arrhandler.h"
5
6
7  int main(int argc, char **argv)
8  {
9      int *arr, N, K, x1, x2, y1, y2;
10
11     N = get_n();
12     K = get_k(N);
13
```

```
14      arr = fill_array(N);
15      quicksort(arr, 0, N-1);
16      x_pair(&x1, &x2);
17      y_pair(&y1, &y2);
18      print_combs(arr, N, K, x1, x2, y1, y2);
19
20      free(arr);
21
22      return 0;
23  }
```

3.2 kcombinations.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <stdbool.h>
4   #include "kcombinations.h"
5   #include "arrhandler.h"
6   #include "ccolors.h"
7
8
9   int get_n()
10  {
11      int N;
12
13      do
14      {
15          system("clear||cls");
16          printf("N (6 < N <= 49): ");
17          scanf("%d", &N);
18      } while (N <= 6 || N > 49);
19
20      return N;
21  }
22
23
24  int get_k(int N)
25  {
26      int K;
27
28      do
29      {
30          printf("K (K < N <= 49): ");
31          scanf("%d", &K);
32      } while (K >= N || K > 49);
33
34      system("clear||cls");
35
36      return K;
37  }
```

```
38
39
40 void x_pair(int *x1, int *x2)
41 {
42     do
43     {
44         printf("x1: ");
45         scanf("%d", x1);
46         printf("x2: ");
47         scanf("%d", x2);
48     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
49 }
50
51
52 void y_pair(int *y1, int *y2)
53 {
54     do
55     {
56         printf("y1: ");
57         scanf("%d", y1);
58         printf("y2: ");
59         scanf("%d", y2);
60     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
61 }
62
63
64 void print_combs(int *arr, int N, int K, int x1, int x2, int
       y1, int y2)
65 {
66     int *currComb = (int *)malloc(N * sizeof(int));
67     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
68
69     if (currComb == NULL)
70     {
71         set_color(BOLD_RED);
72         printf("Error! Not enough memory, exiting...\n");
73         exit(EXIT_FAILURE);
74         set_color(STANDARD);
75     }
76     else
77     {
78         combinations(arr, currComb, 0, N-1, 0, K, &printed,
       &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
79         print_other(N, K, unFrstCond, unScndCondOnly,
       printed);
80     }
81
82     free(currComb);
83 }
84
```

```
85
86 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)
87 {
88     int i, j;
89
90     if (index == K)
91     {
92         for (j = 0; j < K; j++)
93         {
94             if (even_calc(currComb, K, x1, x2) &&
       sum_comb_calc(currComb, K, y1, y2))
95             {
96                 printf("%d ", *(currComb + j));
97                 if (j == K - 1) { (*printed)++; printf("\n")
       ; }
98             } // add freq
99         }
100        if (!even_calc(currComb, K, x1, x2) && sum_comb_calc
       (currComb, K, y1, y2)) (*unFrstCond)++;
101        if (!sum_comb_calc(currComb, K, y1, y2)) (*
       unScndCondOnly)++;
102        return;
103    }
104
105    for (i = start; i <= end && end-i+1 >= K-index; i++)
106    {
107        *(currComb + index) = *(arr + i);
108        combinations(arr, currComb, i+1, end, index+1, K,
       printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
109    }
110 }
111
112
113 bool even_calc(int *arr, int K, int x1, int x2)
114 {
115    int numEven = 0, i;
116
117    for (i = 0; i < K; i++)
118        if (*(arr + i) % 2 == 0) numEven++;
119
120    return (numEven >= x1 && numEven <= x2) ? true : false;
121 }
122
123
124 bool sum_comb_calc(int *arr, int K, int y1, int y2)
125 {
126    int sumNums = 0, i;
127
```

```
128     for (i = 0; i < K; i++)
129         sumNums += *(arr + i);
130 
131     return (sumNums >= y1 && sumNums <= y2) ? true : false;
132 }
133 
134 
135 int frequency()
136 {
137 
138 }
139 
140 
141 long int combinations_count(int N, int K) // wtf ???????
142 {
143     return (factorial(N) / (factorial(K) * factorial(N - K))
    );
144 }
145 
146 
147 long double factorial(int num)
148 {
149     int i;
150     long double fac;
151     if (num == 0) return -1;
152     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
153     return fac;
154 }
155 
156 
157 void print_other(int N, int K, int unFrstCond, int
    unScndCondOnly, int printed)
158 {
159     printf("\nTotal number of combinations %d to %d: %ld\n",
     N, K, combinations_count(N, K));
160     printf("Number of combinations not satisfying the first
    condition: %d\n", unFrstCond);
161     printf("Number of combinations not satisfying the second
     condition only: %d\n", unScndCondOnly);
162     printf("Printed combinations: %d\n", printed);
163 }
```

3.3 kcombinations.h

```
1 #ifndef COMBINATIONS_H
2 #define COMBINATIONS_H
3 
4 #include <stdbool.h>
5 
6 void x_pair(int *, int *);
```

```
7  void y_pair(int *, int *);
8
9  void print_combs(int *, int, int, int, int, int, int);
10 void combinations(int *, int *, int, int, int, int, int *,
       int *, int *, int, int, int, int);
11
12 bool even_calc(int *, int, int, int);
13 bool sum_comb_calc(int *, int, int, int);
14
15 int frequency();
16 long int combinations_count(int, int);
17 long double factorial(int);
18 void print_other(int, int, int, int, int); // add freq
19
20 #endif
```

3.4 arrhandler.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "arrhandler.h"
4  #include "kcombinations.h"
5  #include "ccolors.h"
6
7
8  int *fill_array(int N)
9  {
10     int num, i = 0;
11     int *arr = (int *)malloc(N * sizeof(int));
12
13     if (arr == NULL)
14     {
15         set_color(BOLD_RED);
16         printf("Error! Not enough memory, exiting...\n");
17         exit(EXIT_FAILURE);
18         set_color(STANDARD);
19     }
20     else
21     {
22         do
23         {
24             printf("arr[%d]: ", i);
25             scanf("%d", &num);
26
27             if (num >= 1 && num <= 49)
28             {
29                 if (i == 0) { *(arr + i) = num; i++; }
30                 else
31                 {
32                     if (!exists_in_array(arr, N, num)) { *(
```

```
    arr + i) = num; i++; }
33                      else printf("Give a different number.\n"
    );
34                  }
35              }
36              else printf("Give a number in [1, 49].\n");
37          } while (i < N);
38      }
39
40      return arr;
41  }
42
43
44  bool exists_in_array(int *arr, int N, int num)
45  {
46      int *arrEnd = arr + (N - 1);
47      while (arr <= arrEnd && *arr != num) arr++;
48      return (arr <= arrEnd) ? true : false;
49  }
50
51
52  void quicksort(int *arr, int low, int high)
53  {
54      if (low < high)
55      {
56          int partIndex = partition(arr, low, high);
57          quicksort(arr, low, partIndex - 1);
58          quicksort(arr, partIndex + 1, high);
59      }
60  }
61
62
63  int partition(int *arr, int low, int high)
64  {
65      int pivot = *(arr + high);
66      int i = (low - 1), j;
67
68      for (j = low; j <= high - 1; j++)
69          if (*(arr + j) < pivot)
70              swap(arr + ++i, arr + j);
71
72      swap(arr + (i + 1), arr + high);
73      return (i + 1);
74  }
75
76
77  void swap(int *a, int *b)
78  {
79      int temp = *a;
80      *a = *b;
```

```
81      *b = temp;
82  }
```

3.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include <stdbool.h>
5
6  int *fill_array(int);
7  bool exists_in_array(int *, int, int);
8
9  void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

3.6 Περιγραφή υλοποιήσης

4 fcombinations - συνδυασμοί από αρχείο

4.1 main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "fcombinations.h"
4  #include "arrhandler.h"
5
6
7  int main(int argc, char **argv)
8  {
9      int N, K;
10     int *arr;
11     int x1, x2, y1, y2;
12
13     read_file(argv);
14
15     return 0;
16 }
```

4.2 fcombinations.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <string.h>
5  #include "fcombinations.h"
6  #include "ccolors.h"
```

```
7
8  #define COMBSN 6
9
10
11 void read_file(char **argv)
12 {
13     FILE *dataFile = fopen(argv[1], "r");
14
15     if (dataFile == NULL)
16     {
17         set_color(BOLD_RED);
18         printf("Error! Not enough memory, exiting...\n");
19         exit(EXIT_FAILURE);
20         set_color(STANDARD);
21     }
22     else
23     {
24         printf("Cool\n");
25         // fscanf();
26     }
27
28     fclose(dataFile);
29 }
30
31
32 void x_pair(int *x1, int *x2)
33 {
34     do
35     {
36         printf("x1: ");
37         scanf("%d", x1);
38         printf("x2: ");
39         scanf("%d", x2);
40     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
41 }
42
43
44 void y_pair(int *y1, int *y2)
45 {
46     do
47     {
48         printf("y1: ");
49         scanf("%d", y1);
50         printf("y2: ");
51         scanf("%d", y2);
52     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
53 }
54
55
56 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++
    )
```

```
98      {
99          *(currComb + index) = *(arr + i);
100         combinations(arr, currComb, i+1, end, index+1,
        printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
101     }
102 }
103
104
105 bool even_calc(int *arr, int x1, int x2)
106 {
107     int numEven = 0, i;
108
109     for (i = 0; i < COMBSN; i++)
110         if (*(arr + i) % 2 == 0) numEven++;
111
112     return (numEven >= x1 && numEven <= x2) ? true : false;
113 }
114
115
116 bool sum_comb_calc(int *arr, int y1, int y2)
117 {
118     int sumNums = 0, i;
119
120     for (i = 0; i < COMBSN; i++)
121         sumNums += *(arr + i);
122
123     return (sumNums >= y1 && sumNums <= y2) ? true : false;
124 }
125
126
127 int frequency()
128 {
129
130 }
131
132
133 long int combinations_count(int N) // wtf ???????
134 {
135     return (factorial(N) / (factorial(COMBSN) * factorial(N
        - COMBSN)));
136 }
137
138
139 long double factorial(int num)
140 {
141     int i;
142     long double fac;
143     if (num == 0) return -1;
144     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
145     return fac;
```

```
146 }
147
148
149 void print_other(int N, int unFrstCond, int unScndCondOnly,
        int printed)
150 {
151     printf("\nTotal number of combinations %d to %d: %ld\n",
          N, COMBSN, combinations_count(N));
152     printf("Number of combinations not satisfying the first
        condition: %d\n", unFrstCond);
153     printf("Number of combinations not satisfying the second
         condition only: %d\n", unScndCondOnly);
154     printf("Printed combinations: %d\n", printed);
155 }
```

4.3 fcombinations.h

```
1  #ifndef COMBINATIONS_H
2  #define COMBINATIONS_H
3
4  #include <stdbool.h>
5
6  #define COMBSN 6
7
8  void read_file();
9
10 void x_pair(int *, int *);
11 void y_pair(int *, int *);
12
13 void print_combs(int *, int, int, int, int, int);
14 void combinations(int *, int *, int, int, int, int *, int *,
       int *, int, int, int, int);
15
16 bool even_calc(int *, int, int);
17 bool sum_comb_calc(int *, int, int);
18
19 int frequency();
20 long int combinations_count(int);
21 long double factorial(int);
22 void print_other(int, int, int, int); // add freq
23
24 #endif
```

4.4 arrhandler.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "arrhandler.h"
4  #include "fcombinations.h"
5  #include "ccolors.h"
```

```
6
7
8  int *fill_array(int N)
9  {
10     int num, i = 0;
11     int *arr = (int *)malloc(N * sizeof(int));
12
13     if (arr == NULL)
14     {
15         set_color(BOLD_RED);
16         printf("Error! Not enough memory, exiting...\n");
17         exit(EXIT_FAILURE);
18         set_color(STANDARD);
19     }
20     else
21     {
22         do
23         {
24             printf("arr[%d]: ", i);
25             scanf("%d", &num);
26
27             if (num >= 1 && num <= 49)
28             {
29                 if (i == 0) { *(arr + i) = num; i++; }
30                 else
31                 {
32                     if (!exists_in_array(arr, N, num)) { *(
       arr + i) = num; i++; }
33                     else printf("Give a different number.\n"
       );
34                 }
35             }
36             else printf("Give a number in [1, 49].\n");
37         } while (i < N);
38     }
39
40     return arr;
41 }
42
43
44 bool exists_in_array(int *arr, int N, int num)
45 {
46     int *arrEnd = arr + (N - 1);
47     while (arr <= arrEnd && *arr != num) arr++;
48     return (arr <= arrEnd) ? true : false;
49 }
50
51
52 void quicksort(int *arr, int low, int high)
53 {
```

```
54      if (low < high)
55      {
56          int partIndex = partition(arr, low, high);
57          quicksort(arr, low, partIndex - 1);
58          quicksort(arr, partIndex + 1, high);
59      }
60  }
61
62
63  int partition(int *arr, int low, int high)
64  {
65      int pivot = *(arr + high);
66      int i = (low - 1), j;
67
68      for (j = low; j <= high - 1; j++)
69          if (*(arr + j) < pivot)
70              swap(arr + ++i, arr + j);
71
72      swap(arr + (i + 1), arr + high);
73      return (i + 1);
74  }
75
76
77  void swap(int *a, int *b)
78  {
79      int temp = *a;
80      *a = *b;
81      *b = temp;
82  }
```

4.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include <stdbool.h>
5
6  int *fill_array(int);
7  bool exists_in_array(int *, int, int);
8
9  void quicksort(int *, int, int);
10 int partition(int *, int, int);
11 void swap(int *, int *);
12
13 #endif
```

4.6 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

```
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "minesweeper.h"
#include "gameplay.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();
```

```
30      wrefresh(menuWin);
31      keypad(menuWin, true);
32
33      set_mode(menuWin);
34
35      int WIDTH = set_width(menuWin, xMax);
36      int HEIGHT = set_height(menuWin, yMax);
37      int NMINES = set_nmines(menuWin, WIDTH * HEIGHT);
38
39      game_win(WIDTH, HEIGHT, NMINES);
40      getchar();
41  }
42
43
44  void set_mode(WINDOW *menuWin) // loop
45  {
46      char mode;
47      mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
        );
48      wrefresh(menuWin);
49      scanw("%c", &mode);
50      mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
        ): ") + 1, "%c", mode);
51      wrefresh(menuWin);
52      mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
        spyrosROUM!!!! :-DDDD
53      wrefresh(menuWin);
54
55      switch (mode)
56      {
57          case 'k':
58          case 'K':
59              mvwprintw(menuWin, 2, 1, "Keyboard mode");
60              wrefresh(menuWin);
61              break;
62          case 't':
63          case 'T':
64              mvwprintw(menuWin, 2, 1, "Text mode");
65              wrefresh(menuWin);
66              break;
67          default:
68              break;
69      }
70  }
71
72
73  int set_width(WINDOW *menuWin, int xMax)
74  {
75      int WIDTH;
76
```

```
77      do
78      {
79          mvwprintw(menuWin, 1, 1, "Width (Max = %d): ", xMax-
    12);
80          wrefresh(menuWin);
81          scanw("%d", &WIDTH);
82          mvwprintw(menuWin, 1, strlen("Width (Max = XXX): ")
    + 1, "%d", WIDTH);
83          wrefresh(menuWin);
84      } while (WIDTH < 5 || WIDTH > xMax - 12);
85
86      return WIDTH;
87  }
88
89
90  int set_height(WINDOW *menuWin, int yMax)
91  {
92      int HEIGHT;
93
94      do
95      {
96          mvwprintw(menuWin, 2, 1, "Height (Max = %d): ", yMax
    -12);
97          wrefresh(menuWin);
98          scanw("%d", &HEIGHT);
99          mvwprintw(menuWin, 2, strlen("Height (Max = YYY): ")
     + 1, "%d", HEIGHT);
100         wrefresh(menuWin);
101     } while (HEIGHT < 5 || HEIGHT > yMax - 12);
102
103     return HEIGHT;
104 }
105
106
107 int set_nmines(WINDOW *menuWin, int DIMENSIONS)
108 {
109     int NMINES;
110
111     do
112     {
113         mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
    DIMENSIONS-10); // -10 so the player has a chance to win
114         wrefresh(menuWin);
115         scanw("%d", &NMINES);
116         mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ")
     + 1, "%d", NMINES);
117         wrefresh(menuWin);
118     } while (NMINES < 1 || NMINES > DIMENSIONS-10);
119
120     return NMINES;
```

```
121 }
122
123
124 void game_win(int WIDTH, int HEIGHT, int NMINES)
125 {
126     int yMax, xMax;
127     getmaxyx(stdscr, yMax, xMax);
128
129     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
        // fix 43
130     box(gameWin, 0, 0);
131     refresh();
132     wrefresh(gameWin);
133     keypad(gameWin, true);
134
135     char **dispboard = init_dispboard(gameWin, WIDTH, HEIGHT
        );
136     char **mineboard = init_mineboard(gameWin, WIDTH, HEIGHT
        , NMINES);
137
138     selection(gameWin, dispboard, mineboard, WIDTH, HEIGHT);
139
140     free(dispboard);
141     free(mineboard);
142 }
143
144
145 char **init_dispboard(WINDOW *gameWin, int WIDTH, int HEIGHT
        )
146 {
147     int i;
148     char **dispboard = (char **)malloc(WIDTH * sizeof(char *
        ));
149     for (i = 0; i < WIDTH; i++)
150         dispboard[i] = (char *)malloc(HEIGHT);
151
152     if (dispboard == NULL)
153     {
154         mvprintw(1, 1, "Error, not enough memory, exiting...
        ");
155         exit(EXIT_FAILURE);
156     }
157     else
158     {
159         fill_dispboard(dispboard, WIDTH, HEIGHT);
160         print_board(gameWin, dispboard, WIDTH, HEIGHT);
161         getchar();
162     }
163
164     return dispboard;
```

```
165 }
166
167 void fill_dispboard(char **dispboard, int WIDTH, int HEIGHT)
168 {
169     int i, j;
170
171     for (i = 0; i < WIDTH; i++)
172         for (j = 0; j < HEIGHT; j++)
173             dispboard[i][j] = HIDDEN;
174 }
175
176
177 char **init_mineboard(WINDOW *gameWin, int WIDTH, int HEIGHT
        , int NMINES)
178 {
179     int i;
180     char **mineboard = (char **)malloc(WIDTH * sizeof(char *
        ));
181     for (i = 0; i < WIDTH; i++)
182         mineboard[i] = (char *)malloc(HEIGHT);
183
184     if (mineboard == NULL)
185     {
186         mvprintw(1, 1, "Error, not enough memory, exiting...
        ");
187         exit(EXIT_FAILURE);
188     }
189     else
190     {
191         place_mines(mineboard, WIDTH, HEIGHT, NMINES);
192         add_adj(mineboard, WIDTH, HEIGHT);
193         fill_spaces(mineboard, WIDTH, HEIGHT, NMINES);
194     }
195
196     return mineboard;
197 }
198
199
200 void place_mines(char **mineboard, int WIDTH, int HEIGHT,
        int NMINES)
201 {
202     int i, wRand, hRand;
203
204     srand(time(NULL));
205
206     for (i = 0; i < NMINES; i++)
207     {
208         wRand = rand() % WIDTH;
209         hRand = rand() % HEIGHT;
210         mineboard[wRand][hRand] = MINE;
```

```
211     }
212 }
213
214
215 void add_adj(char **mineboard, int WIDTH, int HEIGHT)
216 {
217     int i, j;
218
219     for (i = 0; i < WIDTH; i++)
220         for (j = 0; j < HEIGHT; j++)
221             if (!is_mine(mineboard, i, j))
222                 mineboard[i][j] = adj_mines(mineboard, i, j,
     WIDTH, HEIGHT) + '0';
223 }
224
225
226 bool is_mine(char **mineboard, int row, int col)
227 {
228     return (mineboard[row][col] == MINE) ? true : false;
229 }
230
231 bool outof_bounds(int row, int col, int WIDTH, int HEIGHT)
232 {
233     return (row < 0 || row > WIDTH-1 || col < 0 || col >
    HEIGHT-1) ? true : false;
234 }
235
236
237
238 int8_t adj_mines(char **mineboard, int row, int col, int
    WIDTH, int HEIGHT)
239 {
240     int8_t numAdj = 0;
241
242     if (!outof_bounds(row, col - 1, WIDTH, HEIGHT)      &&
    mineboard[row][col-1]    == MINE) numAdj++; // North
243     if (!outof_bounds(row, col + 1, WIDTH, HEIGHT)      &&
    mineboard[row][col+1]    == MINE) numAdj++; // South
244     if (!outof_bounds(row + 1, col, WIDTH, HEIGHT)      &&
    mineboard[row+1][col]    == MINE) numAdj++; // East
245     if (!outof_bounds(row - 1, col, WIDTH, HEIGHT)      &&
    mineboard[row-1][col]    == MINE) numAdj++; // West
246     if (!outof_bounds(row + 1, col - 1, WIDTH, HEIGHT)  &&
    mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
247     if (!outof_bounds(row - 1, col - 1, WIDTH, HEIGHT)  &&
    mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
248     if (!outof_bounds(row + 1, col + 1, WIDTH, HEIGHT)  &&
    mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
249     if (!outof_bounds(row - 1, col + 1, WIDTH, HEIGHT)  &&
    mineboard[row-1][col+1]  == MINE) numAdj++; // South-West
```

```
250
251     return numAdj;
252 }
253
254
255 void fill_spaces(char **mineboard, int WIDTH, int HEIGHT,
    int NMINES)
256 {
257     int i, j;
258
259     for (i = 0; i < WIDTH; i++)
260         for (j = 0; j < HEIGHT; j++)
261             if (mineboard[i][j] != MINE && mineboard[i][j] =
    = '0')
262                 mineboard[i][j] = '-';
263 }
264
265
266 void print_board(WINDOW *gameWin, char **mineboard, int
    WIDTH, int HEIGHT)
267 {
268     int i, j;
269
270     for (i = 0; i < WIDTH; i++)
271     {
272         for (j = 0; j < HEIGHT; j++)
273         {
274             mvwaddch(gameWin, j + 1, i + 1, mineboard[i][j])
    ;
275             wrefresh(gameWin);
276         }
277     }
278 }
279
280
281 void filewrite(char **mineboard, int WIDTH, int HEIGHT, int
    hitRow, int hitCol)
282 {
283     int i, j;
284     FILE *mnsOut = fopen("mnsout.txt", "w");
285
286     if (mnsOut == NULL)
287     {
288         mvprintw(1, 1, "Error opening file, exiting...");
289         exit(EXIT_FAILURE);
290     }
291     else
292     {
293         fprintf(mnsOut, "Mine hit at position (%d, %d)\n\n",
     hitRow, hitCol);
```

```
294         fprintf(mnsOut, "Board overview\n\n");
295
296         for (i = 0; i < WIDTH; i++) // fix inversion
297         {
298             for (j = 0; j < HEIGHT; j++)
299                 fprintf(mnsOut, "%c ", mineboard[i][j]);
300             fprintf(mnsOut, "\n");
301         }
302
303         mvprintw(1, 1, "Session written to file");
304         refresh();
305     }
306
307     fclose(mnsOut);
308 }
```

5.3 minesweeper.h

```
1  #ifndef MINESWEEPER_H
2  #define MINESWEEPER_H
3
4  #if defined linux || defined __unix__
5  #include <ncurses.h>
6  #elif defined _WIN32 || defined _WIN64
7  #include <pdcurses.h>
8  #include <stdint.h>
9  #endif
10
11 #include <stdbool.h>
12
13 #define HIDDEN '#'
14 #define MINE '*'
15 #define CLEAR "
                          "
16
17 void main_win();
18 void start();
19 void set_mode(struct _win_st*);
20
21 int set_width(struct _win_st*, int);
22 int set_height(struct _win_st*, int);
23 int set_nmines(struct _win_st*, int);
24
25 void game_win(int, int, int);
26 char **init_dispboard(struct _win_st*, int, int);
27 void fill_dispboard(char **, int, int);
28 char **init_mineboard(struct _win_st*, int, int, int);
29 void place_mines(char **, int, int, int);
30 void add_adj(char **, int, int);
31 bool is_mine(char **, int, int);
```

```
32 bool outof_bounds(int, int, int, int);
33 int8_t adj_mines(char **, int, int, int, int);
34 void fill_spaces(char **, int, int, int);
35
36 void print_board(struct _win_st*, char **, int, int);
37 void filewrite(char **, int, int, int, int);
38
39 #endif
```

5.4 gameplay.c

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <time.h>
4  #include "minesweeper.h"
5  #include "gameplay.h"
6
7  void main_win()
8  {
9      initscr();
10     noecho();
11     cbreak();
12
13     WINDOW *mainWin = newwin(0, 0, 0, 0);
14     box(mainWin, 0, 0);
15     refresh();
16     wrefresh(mainWin);
17     keypad(mainWin, true);
18 }
19
20
21 void start()
22 {
23     int yMax, xMax;
24     int numSettings = 3;
25     getmaxyx(stdscr, yMax, xMax);
26
27     WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
       5);
28     box(menuWin, 0, 0);
29     refresh();
30     wrefresh(menuWin);
31     keypad(menuWin, true);
32
33     set_mode(menuWin);
34
35     int WIDTH = set_width(menuWin, xMax);
36     int HEIGHT = set_height(menuWin, yMax);
37     int NMINES = set_nmines(menuWin, WIDTH * HEIGHT);
38
```

```
39      game_win(WIDTH, HEIGHT, NMINES);
40      getchar();
41  }
42
43
44  void set_mode(WINDOW *menuWin) // loop
45  {
46      char mode;
47      mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
        );
48      wrefresh(menuWin);
49      scanw("%c", &mode);
50      mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
        ): ") + 1, "%c", mode);
51      wrefresh(menuWin);
52      mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
        spyrosROUM!!!! :-DDDD
53      wrefresh(menuWin);
54
55      switch (mode)
56      {
57          case 'k':
58          case 'K':
59              mvwprintw(menuWin, 2, 1, "Keyboard mode");
60              wrefresh(menuWin);
61              break;
62          case 't':
63          case 'T':
64              mvwprintw(menuWin, 2, 1, "Text mode");
65              wrefresh(menuWin);
66              break;
67          default:
68              break;
69      }
70  }
71
72
73  int set_width(WINDOW *menuWin, int xMax)
74  {
75      int WIDTH;
76
77      do
78      {
79          mvwprintw(menuWin, 1, 1, "Width (Max = %d): ", xMax-
        12);
80          wrefresh(menuWin);
81          scanw("%d", &WIDTH);
82          mvwprintw(menuWin, 1, strlen("Width (Max = XXX): ")
        + 1, "%d", WIDTH);
83          wrefresh(menuWin);
```

```
84      } while (WIDTH < 5 || WIDTH > xMax - 12);
85
86      return WIDTH;
87  }
88
89
90  int set_height(WINDOW *menuWin, int yMax)
91  {
92      int HEIGHT;
93
94      do
95      {
96          mvwprintw(menuWin, 2, 1, "Height (Max = %d): ", yMax
        -12);
97          wrefresh(menuWin);
98          scanw("%d", &HEIGHT);
99          mvwprintw(menuWin, 2, strlen("Height (Max = YYY): ")
         + 1, "%d", HEIGHT);
100         wrefresh(menuWin);
101     } while (HEIGHT < 5 || HEIGHT > yMax - 12);
102
103     return HEIGHT;
104 }
105
106
107 int set_nmines(WINDOW *menuWin, int DIMENSIONS)
108 {
109     int NMINES;
110
111     do
112     {
113         mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
        DIMENSIONS-10); // -10 so the player has a chance to win
114         wrefresh(menuWin);
115         scanw("%d", &NMINES);
116         mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ")
         + 1, "%d", NMINES);
117         wrefresh(menuWin);
118     } while (NMINES < 1 || NMINES > DIMENSIONS-10);
119
120     return NMINES;
121 }
122
123
124 void game_win(int WIDTH, int HEIGHT, int NMINES)
125 {
126     int yMax, xMax;
127     getmaxyx(stdscr, yMax, xMax);
128
129     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
```

```
     // fix 43
130      box(gameWin, 0, 0);
131      refresh();
132      wrefresh(gameWin);
133      keypad(gameWin, true);
134
135      char **dispboard = init_dispboard(gameWin, WIDTH, HEIGHT
         );
136      char **mineboard = init_mineboard(gameWin, WIDTH, HEIGHT
         , NMINES);
137
138      selection(gameWin, dispboard, mineboard, WIDTH, HEIGHT);
139
140      free(dispboard);
141      free(mineboard);
142  }
143
144
145  char **init_dispboard(WINDOW *gameWin, int WIDTH, int HEIGHT
         )
146  {
147      int i;
148      char **dispboard = (char **)malloc(WIDTH * sizeof(char *
         ));
149      for (i = 0; i < WIDTH; i++)
150          dispboard[i] = (char *)malloc(HEIGHT);
151
152      if (dispboard == NULL)
153      {
154          mvprintw(1, 1, "Error, not enough memory, exiting...
         ");
155          exit(EXIT_FAILURE);
156      }
157      else
158      {
159          fill_dispboard(dispboard, WIDTH, HEIGHT);
160          print_board(gameWin, dispboard, WIDTH, HEIGHT);
161          getchar();
162      }
163
164      return dispboard;
165  }
166
167  void fill_dispboard(char **dispboard, int WIDTH, int HEIGHT)
168  {
169      int i, j;
170
171      for (i = 0; i < WIDTH; i++)
172          for (j = 0; j < HEIGHT; j++)
173              dispboard[i][j] = HIDDEN;
```

```
174 }
175
176
177 char **init_mineboard(WINDOW *gameWin, int WIDTH, int HEIGHT
        , int NMINES)
178 {
179     int i;
180     char **mineboard = (char **)malloc(WIDTH * sizeof(char *
        ));
181     for (i = 0; i < WIDTH; i++)
182         mineboard[i] = (char *)malloc(HEIGHT);
183
184     if (mineboard == NULL)
185     {
186         mvprintw(1, 1, "Error, not enough memory, exiting...
        ");
187         exit(EXIT_FAILURE);
188     }
189     else
190     {
191         place_mines(mineboard, WIDTH, HEIGHT, NMINES);
192         add_adj(mineboard, WIDTH, HEIGHT);
193         fill_spaces(mineboard, WIDTH, HEIGHT, NMINES);
194     }
195
196     return mineboard;
197 }
198
199
200 void place_mines(char **mineboard, int WIDTH, int HEIGHT,
        int NMINES)
201 {
202     int i, wRand, hRand;
203
204     srand(time(NULL));
205
206     for (i = 0; i < NMINES; i++)
207     {
208         wRand = rand() % WIDTH;
209         hRand = rand() % HEIGHT;
210         mineboard[wRand][hRand] = MINE;
211     }
212 }
213
214
215 void add_adj(char **mineboard, int WIDTH, int HEIGHT)
216 {
217     int i, j;
218
219     for (i = 0; i < WIDTH; i++)
```

```
220         for (j = 0; j < HEIGHT; j++)
221             if (!is_mine(mineboard, i, j))
222                 mineboard[i][j] = adj_mines(mineboard, i, j,
     WIDTH, HEIGHT) + '0';
223 }
224
225
226 bool is_mine(char **mineboard, int row, int col)
227 {
228     return (mineboard[row][col] == MINE) ? true : false;
229 }
230
231 bool outof_bounds(int row, int col, int WIDTH, int HEIGHT)
232 {
233     return (row < 0 || row > WIDTH-1 || col < 0 || col >
    HEIGHT-1) ? true : false;
234 }
235
236
237
238 int8_t adj_mines(char **mineboard, int row, int col, int
    WIDTH, int HEIGHT)
239 {
240     int8_t numAdj = 0;
241
242     if (!outof_bounds(row, col - 1, WIDTH, HEIGHT)      &&
    mineboard[row][col-1]    == MINE) numAdj++; // North
243     if (!outof_bounds(row, col + 1, WIDTH, HEIGHT)      &&
    mineboard[row][col+1]    == MINE) numAdj++; // South
244     if (!outof_bounds(row + 1, col, WIDTH, HEIGHT)      &&
    mineboard[row+1][col]    == MINE) numAdj++; // East
245     if (!outof_bounds(row - 1, col, WIDTH, HEIGHT)      &&
    mineboard[row-1][col]    == MINE) numAdj++; // West
246     if (!outof_bounds(row + 1, col - 1, WIDTH, HEIGHT)  &&
    mineboard[row+1][col-1]  == MINE) numAdj++; // North-East
247     if (!outof_bounds(row - 1, col - 1, WIDTH, HEIGHT)  &&
    mineboard[row-1][col-1]  == MINE) numAdj++; // North-West
248     if (!outof_bounds(row + 1, col + 1, WIDTH, HEIGHT)  &&
    mineboard[row+1][col+1]  == MINE) numAdj++; // South-East
249     if (!outof_bounds(row - 1, col + 1, WIDTH, HEIGHT)  &&
    mineboard[row-1][col+1]  == MINE) numAdj++; // South-West
250
251     return numAdj;
252 }
253
254
255 void fill_spaces(char **mineboard, int WIDTH, int HEIGHT,
    int NMINES)
256 {
257     int i, j;
```

```
258
259     for (i = 0; i < WIDTH; i++)
260         for (j = 0; j < HEIGHT; j++)
261             if (mineboard[i][j] != MINE && mineboard[i][j] =
    = '0')
262                 mineboard[i][j] = '-';
263 }
264
265
266 void print_board(WINDOW *gameWin, char **mineboard, int
    WIDTH, int HEIGHT)
267 {
268     int i, j;
269
270     for (i = 0; i < WIDTH; i++)
271     {
272         for (j = 0; j < HEIGHT; j++)
273         {
274             mvwaddch(gameWin, j + 1, i + 1, mineboard[i][j])
    ;
275             wrefresh(gameWin);
276         }
277     }
278 }
279
280
281 void filewrite(char **mineboard, int WIDTH, int HEIGHT, int
    hitRow, int hitCol)
282 {
283     int i, j;
284     FILE *mnsOut = fopen("mnsout.txt", "w");
285
286     if (mnsOut == NULL)
287     {
288         mvprintw(1, 1, "Error opening file, exiting...");
289         exit(EXIT_FAILURE);
290     }
291     else
292     {
293         fprintf(mnsOut, "Mine hit at position (%d, %d)\n\n",
     hitRow, hitCol);
294         fprintf(mnsOut, "Board overview\n\n");
295
296         for (i = 0; i < WIDTH; i++) // fix inversion
297         {
298             for (j = 0; j < HEIGHT; j++)
299                 fprintf(mnsOut, "%c ", mineboard[i][j]);
300             fprintf(mnsOut, "\n");
301         }
302
```

```
303         mvprintw(1, 1, "Session written to file");
304         refresh();
305     }
306
307     fclose(mnsOut);
308 }
```

5.5 gameplay.h

```
1  #ifndef GAMEPLAY_H
2  #define GAMEPLAY_H
3
4  #if defined linux || defined __unix__
5  #include <ncurses.h>
6  #elif defined _WIN32 || defined _WIN64
7  #include <pdcurses.h>
8  #include <stdint.h>
9  #endif
10
11 #include <stdbool.h>
12
13 void selection(struct _win_st*, char **, char **, int, int);
14 bool transfer(char **, char **, int, int);
15 void reveal(struct _win_st*, char **, int, int);
16 void game_over(struct _win_st*, char **, int, int);
17
18 #endif
```

5.6 Περιγραφή υλοποιήσης

6 Διευκρινήσεις

7 Εργαλεία

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