

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 υλοποίησης	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 υλοποίησης	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 υλοποίησης	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 υλοποίησης	35

6	Διευκρινήσεις	35
---	---------------	----

7	Εργασία	35
---	---------	----

1 Δομή προγραμμάτων και οδηγίες εκτέλεσης

1.1 Εκτέλεση από Linux

```
1 $ cd path-to-program
2 $ make
3 $ make run
4 $ make run ARGS=txt/data.txt #fcombinations ONLY
5 $ make clean #optional
```

1.2 Δομή φακέλων

2 combinations - συνδυασμοί

2.1 main.c

```
1 #include "combinations.h"
2
3 int main(int argc, char **argv)
4 {
5     int *arr, N, x1, x2, y1, y2;
6
7     N = get_n();
8
9     arr = fill_array(N);
10    quicksort(arr, 0, N-1);
11    x_pair(&x1, &x2);
12    y_pair(&y1, &y2);
13    print_combs(arr, N, x1, x2, y1, y2);
14
15    free(arr);
16
17    return 0;
18 }
```

2.2 combinations.c

```
1 #include "combinations.h"
2
3 int get_n()
4 {
5     int N;
6
7     do
```

```
8     {
9         system("clear||cls");
10        printf("N (6 < N <= 49): ");
11        scanf("%d", &N);
12    } while (N <= 6 || N > 49);
13
14    system("clear||cls");
15
16    return N;
17 }
18
19
20 void x_pair(int *x1, int *x2)
21 {
22     do
23     {
24         printf("x1: ");
25         scanf("%d", x1);
26         printf("x2: ");
27         scanf("%d", x2);
28     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
29 }
30
31
32 void y_pair(int *y1, int *y2)
33 {
34     do
35     {
36         printf("y1: ");
37         scanf("%d", y1);
38         printf("y2: ");
39         scanf("%d", y2);
40     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
41 }
42
43
44 void print_combs(int *arr, int N, int x1, int x2, int y1,
45                 int y2)
46 {
47     int *currComb = (int *)malloc(N * sizeof(int));
48     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
49
50     if (currComb == NULL)
51     {
52         set_color(BOLD_RED);
53         printf("Error! Not enough memory, exiting...\n");
54         exit(EXIT_FAILURE);
55         set_color(STANDARD);
56     }
57     else
```

```

57     {
58         combinations(arr, currComb, 0, N-1, 0, &printed, &
unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
59         print_other(N, unFrstCond, unScndCondOnly, printed);
60     }
61
62     free(currComb);
63 }
64
65
66 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)
67 {
68     int i, j;
69
70     if (index == COMBSN)
71     {
72         for (j = 0; j < COMBSN; j++)
73         {
74             if (even_calc(currComb, x1, x2) && sum_comb_calc
(currComb, y1, y2))
75             {
76                 printf("%d ", *(currComb + j));
77                 if (j == COMBSN - 1) { (*printed)++; printf(
"\n"); }
78             } // add freq
79         }
80         if (!even_calc(currComb, x1, x2) && sum_comb_calc(
currComb, y1, y2)) (*unFrstCond)++;
81         if (!sum_comb_calc(currComb, y1, y2)) (*
unScndCondOnly)++;
82         return;
83     }
84
85     for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
86     {
87         *(currComb + index) = *(arr + i);
88         combinations(arr, currComb, i+1, end, index+1,
printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
89     }
90 }
91
92
93 bool even_calc(int *arr, int x1, int x2)
94 {
95     int numEven = 0, i;
96
97     for (i = 0; i < COMBSN; i++)

```

```
98         if (*(arr + i) % 2 == 0) numEven++;
99
100     return (numEven >= x1 && numEven <= x2) ? true : false;
101 }
102
103
104 bool sum_comb_calc(int *arr, int y1, int y2)
105 {
106     int sumNums = 0, i;
107
108     for (i = 0; i < COMBSN; i++)
109         sumNums += *(arr + i);
110
111     return (sumNums >= y1 && sumNums <= y2) ? true : false;
112 }
113
114
115 int frequency()
116 {
117
118 }
119
120
121 long int combinations_count(int N) // wtf ???????
122 {
123     return (factorial(N) / (factorial(COMBSN) * factorial(N
124 - COMBSN)));
125 }
126
127 long double factorial(int num)
128 {
129     int i;
130     long double fac;
131     if (num == 0) return -1;
132     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
133     return fac;
134 }
135
136
137 void print_other(int N, int unFrstCond, int unScndCondOnly,
138 int printed)
139 {
140     printf("\nTotal number of combinations %d to %d: %ld\n",
141 N, COMBSN, combinations_count(N));
142     printf("Number of combinations not satisfying the first
143 condition: %d\n", unFrstCond);
144     printf("Number of combinations not satisfying the second
145 condition only: %d\n", unScndCondOnly);
146     printf("Printed combinations: %d\n", printed);
```

143 }

2.3 combinations.h

```

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

```

2.4 arrhandler.c

```

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

```

```
16     {
17         do
18         {
19             printf("arr[%d]: ", i);
20             scanf("%d", &num);
21
22             if (num >= 1 && num <= 49)
23             {
24                 if (i == 0) { *(arr + i) = num; i++; }
25                 else
26                 {
27                     if (!exists_in_array(arr, N, num)) { *(
arr + i) = num; i++; }
28                     else printf("Give a different number.\n"
);
29                 }
30             }
31             else printf("Give a number in [1, 49].\n");
32         } while (i < N);
33     }
34
35     return arr;
36 }
37
38
39 bool exists_in_array(int *arr, int N, int num)
40 {
41     int *arrEnd = arr + (N - 1);
42     while (arr <= arrEnd && *arr != num) arr++;
43     return (arr <= arrEnd) ? true : false;
44 }
45
46
47 void quicksort(int *arr, int low, int high)
48 {
49     if (low < high)
50     {
51         int partIndex = partition(arr, low, high);
52         quicksort(arr, low, partIndex - 1);
53         quicksort(arr, partIndex + 1, high);
54     }
55 }
56
57
58 int partition(int *arr, int low, int high)
59 {
60     int pivot = *(arr + high);
61     int i = (low - 1), j;
62
63     for (j = low; j <= high - 1; j++)
```



```
64         if (*(arr + j) < pivot)
65             swap(arr + ++i, arr + j);
66
67     swap(arr + (i + 1), arr + high);
68     return (i + 1);
69 }
70
71
72 void swap(int *a, int *b)
73 {
74     int temp = *a;
75     *a = *b;
76     *b = temp;
77 }
```

2.5 arrhandler.h

```
1 #ifndef ARRHANDLER_H
2 #define ARRHANDLER_H
3
4 #include "combinations.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 Διάγραμμα ροής

2.7 Περιγραφή υλοποίησης

3 kcombinations - συνδυασμοί με K

3.1 main.c

```
1 #include "kcombinations.h"
2
3 int main(int argc, char **argv)
4 {
5     int *arr, N, K, x1, x2, y1, y2;
6
7     N = get_n();
8     K = get_k(N);
9
10    arr = fill_array(N);
11    quicksort(arr, 0, N-1);
```

```
12     x_pair(&x1, &x2);
13     y_pair(&y1, &y2);
14     print_combs(arr, N, K, x1, x2, y1, y2);
15
16     free(arr);
17
18     return 0;
19 }
```

3.2 kcombinations.c

```
1  #include "kcombinations.h"
2
3  int get_n()
4  {
5      int N;
6
7      do
8      {
9          system("clear||cls");
10         printf("N (6 < N <= 49): ");
11         scanf("%d", &N);
12     } while (N <= 6 || N > 49);
13
14     return N;
15 }
16
17
18 int get_k(int N)
19 {
20     int K;
21
22     do
23     {
24         printf("K (K < N <= 49): ");
25         scanf("%d", &K);
26     } while (K >= N || K > 49);
27
28     system("clear||cls");
29
30     return K;
31 }
32
33
34 void x_pair(int *x1, int *x2)
35 {
36     do
37     {
38         printf("x1: ");
39         scanf("%d", x1);
```

```

40     printf("x2: ");
41     scanf("%d", &x2);
42 } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
43 }
44
45
46 void y_pair(int *y1, int *y2)
47 {
48     do
49     {
50         printf("y1: ");
51         scanf("%d", &y1);
52         printf("y2: ");
53         scanf("%d", &y2);
54     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
55 }
56
57
58 void print_combs(int *arr, int N, int K, int x1, int x2, int
59 y1, int y2)
60 {
61     int *currComb = (int *)malloc(N * sizeof(int));
62     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
63
64     if (currComb == NULL)
65     {
66         set_color(BOLD_RED);
67         printf("Error! Not enough memory, exiting...\n");
68         exit(EXIT_FAILURE);
69         set_color(STANDARD);
70     }
71     else
72     {
73         combinations(arr, currComb, 0, N-1, 0, K, &printed,
74 &unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
75         print_other(N, K, unFrstCond, unScndCondOnly,
76 printed);
77     }
78
79     free(currComb);
80 }
81
82 void combinations(int *arr, int *currComb, int start, int
83 end, int index, int K, int *printed, int *unFrstCond, int
84 *unScndCondOnly, int x1, int x2, int y1, int y2)
85 {
86     int i, j;
87
88     if (index == K)

```

```

85     {
86         for (j = 0; j < K; j++)
87         {
88             if (even_calc(currComb, K, x1, x2) &&
89 sum_comb_calc(currComb, K, y1, y2))
90             {
91                 printf("%d ", *(currComb + j));
92                 if (j == K - 1) { (*printed)++; printf("\n")
93 ; }
94             } // add freq
95         }
96         if (!even_calc(currComb, K, x1, x2) && sum_comb_calc
97 (currComb, K, y1, y2)) (*unFrstCond)++;
98         if (!sum_comb_calc(currComb, K, y1, y2)) (*
99 unScndCondOnly)++;
100         return;
101     }
102
103     for (i = start; i <= end && end-i+1 >= K-index; i++)
104     {
105         *(currComb + index) = *(arr + i);
106         combinations(arr, currComb, i+1, end, index+1, K,
107 printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
108     }
109 }
110
111 bool even_calc(int *arr, int K, int x1, int x2)
112 {
113     int numEven = 0, i;
114
115     for (i = 0; i < K; i++)
116         if (*(arr + i) % 2 == 0) numEven++;
117
118     return (numEven >= x1 && numEven <= x2) ? true : false;
119 }
120
121 bool sum_comb_calc(int *arr, int K, int y1, int y2)
122 {
123     int sumNums = 0, i;
124
125     for (i = 0; i < K; i++)
126         sumNums += *(arr + i);
127
128     return (sumNums >= y1 && sumNums <= y2) ? true : false;
129 }
130
131 int frequency()

```

```

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

```

3.3 kcombinations.h

```

1 #ifndef COMBINATIONS_H
2 #define COMBINATIONS_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7
8 #include "arrhandler.h"
9 #include "ccolors.h"
10
11 void x_pair(int *, int *);
12 void y_pair(int *, int *);
13
14 void print_combs(int *, int, int, int, int, int, int);

```

```

15 void combinations(int *, int *, int, int, int, int, int *,
    int *, int *, int, int, int, int);
16
17 bool even_calc(int *, int, int, int);
18 bool sum_comb_calc(int *, int, int, int);
19
20 int frequency();
21 long int combinations_count(int, int);
22 long double factorial(int);
23 void print_other(int, int, int, int, int); // add freq
24
25 #endif

```

3.4 arrhandler.c

```

1 #include "arrhandler.h"
2
3 int *fill_array(int N)
4 {
5     int num, i = 0;
6     int *arr = (int *)malloc(N * sizeof(int));
7
8     if (arr == NULL)
9     {
10         set_color(BOLD_RED);
11         printf("Error! Not enough memory, exiting...\n");
12         exit(EXIT_FAILURE);
13         set_color(STANDARD);
14     }
15     else
16     {
17         do
18         {
19             printf("arr[%d]: ", i);
20             scanf("%d", &num);
21
22             if (num >= 1 && num <= 49)
23             {
24                 if (i == 0) { *(arr + i) = num; i++; }
25                 else
26                 {
27                     if (!exists_in_array(arr, N, num)) { *(arr + i) = num; i++; }
28                     else printf("Give a different number.\n");
29                 }
30             }
31             else printf("Give a number in [1, 49].\n");
32         } while (i < N);
33     }

```

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

3.5 arrhandler.h

```
1 #ifndef ARRHANDLER_H
2 #define ARRHANDLER_H
3
```

```
4 #include "kcombinations.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 Διάγραμμα ροής

3.7 Περιγραφή υλοποίησης

4 fcombinations - συνδυασμοί από αρχείο

4.1 main.c

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

4.2 fcombinations.c

```
1 #include "fcombinations.h"
2
3 void read_file(char **argv)
4 {
5     FILE *dataFile = fopen(argv[1], "r");
6
7     if (dataFile == NULL)
8     {
9         set_color(BOLD_RED);
10        printf("Error! Not enough memory, exiting...\n");
11        exit(EXIT_FAILURE);
12        set_color(STANDARD);
13    }
14    else
15    {
16        printf("Cool\n");
```



```
17         // fscanf();
18     }
19
20     fclose(dataFile);
21 }
22
23
24 void x_pair(int *x1, int *x2)
25 {
26     do
27     {
28         printf("x1: ");
29         scanf("%d", x1);
30         printf("x2: ");
31         scanf("%d", x2);
32     } while (*x1 < 0 || *x1 > *x2 || *x2 > 6);
33 }
34
35
36 void y_pair(int *y1, int *y2)
37 {
38     do
39     {
40         printf("y1: ");
41         scanf("%d", y1);
42         printf("y2: ");
43         scanf("%d", y2);
44     } while (*y1 < 21 || *y1 > *y2 || *y2 > 279);
45 }
46
47
48 void print_combs(int *arr, int N, int x1, int x2, int y1,
49                 int y2)
50 {
51     int *currComb = (int *)malloc(N * sizeof(int));
52     int unFrstCond = 0, unScndCondOnly = 0, printed = 0;
53
54     if (currComb == NULL)
55     {
56         set_color(BOLD_RED);
57         printf("Error! Not enough memory, exiting...\n");
58         exit(EXIT_FAILURE);
59         set_color(STANDARD);
60     }
61     else
62     {
63         combinations(arr, currComb, 0, N-1, 0, &printed, &
64                     unFrstCond, &unScndCondOnly, x1, x2, y1, y2);
65         print_other(N, unFrstCond, unScndCondOnly, printed);
66     }
67 }
```

```

65     free(currComb);
66 }
67
68
69
70 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)
71 {
72     int i, j;
73
74     if (index == COMBSN)
75     {
76         for (j = 0; j < COMBSN; j++)
77         {
78             if (even_calc(currComb, x1, x2) && sum_comb_calc
                (currComb, y1, y2))
79             {
80                 printf("%d ", *(currComb + j));
81                 if (j == COMBSN - 1) { (*printed)++; printf(
                    "\n"); }
82             } // add freq
83         }
84         if (!even_calc(currComb, x1, x2) && sum_comb_calc(
            currComb, y1, y2)) (*unFrstCond)++;
85         if (!sum_comb_calc(currComb, y1, y2)) (*
            unScndCondOnly)++;
86         return;
87     }
88
89     for (i = start; i <= end && end-i+1 >= COMBSN-index; i++)
90     {
91         *(currComb + index) = *(arr + i);
92         combinations(arr, currComb, i+1, end, index+1,
            printed, unFrstCond, unScndCondOnly, x1, x2, y1, y2);
93     }
94 }
95
96
97 bool even_calc(int *arr, int x1, int x2)
98 {
99     int numEven = 0, i;
100
101     for (i = 0; i < COMBSN; i++)
102         if (*(arr + i) % 2 == 0) numEven++;
103
104     return (numEven >= x1 && numEven <= x2) ? true : false;
105 }
106

```

```

107
108 bool sum_comb_calc(int *arr, int y1, int y2)
109 {
110     int sumNums = 0, i;
111
112     for (i = 0; i < COMBSN; i++)
113         sumNums += *(arr + i);
114
115     return (sumNums >= y1 && sumNums <= y2) ? true : false;
116 }
117
118
119 int frequency()
120 {
121
122 }
123
124
125 long int combinations_count(int N) // wtf ???????
126 {
127     return (factorial(N) / (factorial(COMBSN) * factorial(N
128 - COMBSN)));
129 }
130
131 long double factorial(int num)
132 {
133     int i;
134     long double fac;
135     if (num == 0) return -1;
136     else for (i = 1, fac = 1; i <= num; i++) fac *= i;
137     return fac;
138 }
139
140
141 void print_other(int N, int unFrstCond, int unScndCondOnly,
142                 int printed)
143 {
144     printf("\nTotal number of combinations %d to %d: %ld\n",
145           N, COMBSN, combinations_count(N));
146     printf("Number of combinations not satisfying the first
147 condition: %d\n", unFrstCond);
148     printf("Number of combinations not satisfying the second
149 condition only: %d\n", unScndCondOnly);
150     printf("Printed combinations: %d\n", printed);
151 }

```

4.3 fcombinations.h

```

1 #ifndef COMBINATIONS_H

```

```
2 #define COMBINATIONS_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <stdbool.h>
7 #include <string.h>
8
9 #include "arrhandler.h"
10 #include "ccolors.h"
11
12 #define COMBSN 6
13
14 void read_file();
15
16 void x_pair(int *, int *);
17 void y_pair(int *, int *);
18
19 void print_combs(int *, int, int, int, int, int);
20 void combinations(int *, int *, int, int, int, int *, int *,
    int *, int, int, int, int);
21
22 bool even_calc(int *, int, int);
23 bool sum_comb_calc(int *, int, int);
24
25 int frequency();
26 long int combinations_count(int);
27 long double factorial(int);
28 void print_other(int, int, int, int); // add freq
29
30 #endif
```

4.4 arrhandler.c

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

```
18     {
19         printf("arr[%d]: ", i);
20         scanf("%d", &num);
21
22         if (num >= 1 && num <= 49)
23         {
24             if (i == 0) { *(arr + i) = num; i++; }
25             else
26             {
27                 if (!exists_in_array(arr, N, num)) { *(
arr + i) = num; i++; }
28                 else printf("Give a different number.\n"
);
29             }
30         }
31         else printf("Give a number in [1, 49].\n");
32     } while (i < N);
33 }
34
35 return arr;
36 }
37
38
39 bool exists_in_array(int *arr, int N, int num)
40 {
41     int *arrEnd = arr + (N - 1);
42     while (arr <= arrEnd && *arr != num) arr++;
43     return (arr <= arrEnd) ? true : false;
44 }
45
46
47 void quicksort(int *arr, int low, int high)
48 {
49     if (low < high)
50     {
51         int partIndex = partition(arr, low, high);
52         quicksort(arr, low, partIndex - 1);
53         quicksort(arr, partIndex + 1, high);
54     }
55 }
56
57
58 int partition(int *arr, int low, int high)
59 {
60     int pivot = *(arr + high);
61     int i = (low - 1), j;
62
63     for (j = low; j <= high - 1; j++)
64         if (*(arr + j) < pivot)
65             swap(arr + ++i, arr + j);
```

```
66
67     swap(arr + (i + 1), arr + high);
68     return (i + 1);
69 }
70
71
72 void swap(int *a, int *b)
73 {
74     int temp = *a;
75     *a = *b;
76     *b = temp;
77 }
```

4.5 arrhandler.h

```
1  #ifndef ARRHANDLER_H
2  #define ARRHANDLER_H
3
4  #include "fcombinations.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 Διάγραμμα ροής

4.7 Περιγραφή υλοποίησης

5 minesweeper - ναρκαλιευτής

5.1 main.c

```
1  #include "minesweeper.h"
2
3  int main(int argc, char **argv)
4  {
5      main_win();
6      start();
7      endwin();
8
9      return 0;
10 }
```

5.2 minesweeper.c

```
1 #include "minesweeper.h"
2
3 void main_win()
4 {
5     initscr();
6     noecho();
7     cbreak();
8
9     WINDOW *mainWin = newwin(0, 0, 0, 0);
10    box(mainWin, 0, 0);
11    refresh();
12    wrefresh(mainWin);
13    keypad(mainWin, true);
14 }
15
16
17 void start()
18 {
19     int yMax, xMax;
20     int numSettings = 3;
21     getmaxyx(stdscr, yMax, xMax);
22
23     WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
24                               5);
25     box(menuWin, 0, 0);
26     refresh();
27     wrefresh(menuWin);
28     keypad(menuWin, true);
29
30     set_mode(menuWin);
31
32     int COLS = set_cols(menuWin, xMax);
33     int ROWS = set_rows(menuWin, yMax);
34     int NMINES = set_nmines(menuWin, COLS * ROWS);
35
36     game_win(COLS, ROWS, NMINES);
37     getchar();
38 }
39
40 void game_win(int COLS, int ROWS, int NMINES)
41 {
42     int yMax, xMax;
43     getmaxyx(stdscr, yMax, xMax);
44
45     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
46     // fix 43
47     box(gameWin, 0, 0);
48     refresh();
49     wrefresh(gameWin);
```

```
49     keypad(gameWin, true);
50
51     char **dispboard = init_dispboard(gameWin, COLS, ROWS);
52     char **mineboard = init_mineboard(gameWin, COLS, ROWS,
53     NMINES);
54
55     selection(gameWin, dispboard, mineboard, COLS, ROWS,
56     NMINES);
57
58     free(dispboard);
59     free(mineboard);
60 }
61
62 char **init_dispboard(WINDOW *gameWin, int COLS, int ROWS)
63 {
64     int i;
65     char **dispboard = (char **)malloc(COLS * sizeof(char *))
66 );
67     for (i = 0; i < COLS; i++)
68         dispboard[i] = (char *)malloc(ROWS);
69
70     if (dispboard == NULL)
71     {
72         mvprintw(1, 1, "Error, not enough memory, exiting...
73 ");
74         exit(EXIT_FAILURE);
75     }
76     else
77     {
78         fill_dispboard(dispboard, COLS, ROWS);
79         print_board(gameWin, dispboard, COLS, ROWS);
80         getchar();
81     }
82
83     return dispboard;
84 }
85
86 void fill_dispboard(char **dispboard, int COLS, int ROWS)
87 {
88     int i, j;
89
90     for (i = 0; i < COLS; i++)
91         for (j = 0; j < ROWS; j++)
92             dispboard[i][j] = HIDDEN;
93 }
94
95 char **init_mineboard(WINDOW *gameWin, int COLS, int ROWS,
```



```
    int NMINES)
95 {
96     int i;
97     char **mineboard = (char **)malloc(COLS * sizeof(char *))
    );
98     for (i = 0; i < COLS; i++)
99         mineboard[i] = (char *)malloc(ROWS);
100
101     if (mineboard == NULL)
102     {
103         mvprintw(1, 1, "Error, not enough memory, exiting...
    ");
104         exit(EXIT_FAILURE);
105     }
106     else
107     {
108         place_mines(mineboard, COLS, ROWS, NMINES);
109         add_adj(mineboard, COLS, ROWS);
110         fill_spaces(mineboard, COLS, ROWS, NMINES);
111
112         // tests
113         //print_board(gameWin, mineboard, COLS, ROWS);
114         //filewrite(mineboard, COLS, ROWS, 1, 2);
115     }
116
117     return mineboard;
118 }
119
120
121 void place_mines(char **mineboard, int COLS, int ROWS, int
    NMINES)
122 {
123     int i, wRand, hRand;
124
125     srand(time(NULL));
126
127     for (i = 0; i < NMINES; i++)
128     {
129         wRand = rand() % COLS;
130         hRand = rand() % ROWS;
131         mineboard[wRand][hRand] = MINE;
132     }
133 }
134
135
136 void add_adj(char **mineboard, int COLS, int ROWS)
137 {
138     int i, j;
139
140     for (i = 0; i < COLS; i++)
```

```

141         for (j = 0; j < ROWS; j++)
142             if (!is_mine(mineboard, i, j))
143                 mineboard[i][j] = adj_mines(mineboard, i, j,
144                     COLS, ROWS) + '0';
145     }
146
147     bool is_mine(char **mineboard, int row, int col)
148     {
149         return (mineboard[row][col] == MINE) ? true : false;
150     }
151
152     bool outof_bounds(int row, int col, int COLS, int ROWS)
153     {
154         return (row < 0 || row > COLS-1 || col < 0 || col > ROWS
155             -1) ? true : false;
156     }
157
158
159     int8_t adj_mines(char **mineboard, int row, int col, int
160         COLS, int ROWS)
161     {
162         int8_t numAdj = 0;
163
164         if (!outof_bounds(row, col - 1, COLS, ROWS) &&
165             mineboard[row][col-1] == MINE) numAdj++; // North
166         if (!outof_bounds(row, col + 1, COLS, ROWS) &&
167             mineboard[row][col+1] == MINE) numAdj++; // South
168         if (!outof_bounds(row + 1, col, COLS, ROWS) &&
169             mineboard[row+1][col] == MINE) numAdj++; // East
170         if (!outof_bounds(row - 1, col, COLS, ROWS) &&
171             mineboard[row-1][col] == MINE) numAdj++; // West
172         if (!outof_bounds(row + 1, col - 1, COLS, ROWS) &&
173             mineboard[row+1][col-1] == MINE) numAdj++; // North-East
174         if (!outof_bounds(row - 1, col - 1, COLS, ROWS) &&
175             mineboard[row-1][col-1] == MINE) numAdj++; // North-West
176         if (!outof_bounds(row + 1, col + 1, COLS, ROWS) &&
177             mineboard[row+1][col+1] == MINE) numAdj++; // South-East
178         if (!outof_bounds(row - 1, col + 1, COLS, ROWS) &&
179             mineboard[row-1][col+1] == MINE) numAdj++; // South-West
180
181         return numAdj;
182     }
183
184
185     void fill_spaces(char **mineboard, int COLS, int ROWS, int
186         NMINES)
187     {
188         int i, j;

```

```

179
180     for (i = 0; i < COLS; i++)
181         for (j = 0; j < ROWS; j++)
182             if (mineboard[i][j] != MINE && mineboard[i][j] =
= '0')
183                 mineboard[i][j] = '-';
184 }

```

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 <stdlib.h>
12 #include <string.h>
13 #include <time.h>
14
15 #include "settings.h"
16 #include "gameplay.h"
17 #include "outputs.h"
18
19 #define HIDDEN '#'
20 #define MINE '*'
21 #define CLEAR "
"
22
23 void main_win();
24 void start();
25
26 void game_win(int, int, int);
27 char **init_dispboard(struct _win_st*, int, int);
28 void fill_dispboard(char **, int, int);
29 char **init_mineboard(struct _win_st*, int, int, int);
30
31 void place_mines(char **, int, int, int);
32 void add_adj(char **, int, int);
33 bool is_mine(char **, int, int);
34 bool outof_bounds(int, int, int, int);
35 int8_t adj_mines(char **, int, int, int, int);
36 void fill_spaces(char **, int, int, int);
37
38 #endif

```

5.4 gameplay.c

```
1 #include "minesweeper.h"
2
3 void main_win()
4 {
5     initscr();
6     noecho();
7     cbreak();
8
9     WINDOW *mainWin = newwin(0, 0, 0, 0);
10    box(mainWin, 0, 0);
11    refresh();
12    wrefresh(mainWin);
13    keypad(mainWin, true);
14 }
15
16
17 void start()
18 {
19     int yMax, xMax;
20     int numSettings = 3;
21     getmaxyx(stdscr, yMax, xMax);
22
23     WINDOW *menuWin = newwin(numSettings+2, xMax-10, yMax-7,
24                               5);
25     box(menuWin, 0, 0);
26     refresh();
27     wrefresh(menuWin);
28     keypad(menuWin, true);
29
30     set_mode(menuWin);
31
32     int COLS = set_cols(menuWin, xMax);
33     int ROWS = set_rows(menuWin, yMax);
34     int NMINES = set_nmines(menuWin, COLS * ROWS);
35
36     game_win(COLS, ROWS, NMINES);
37     getchar();
38 }
39
40 void game_win(int COLS, int ROWS, int NMINES)
41 {
42     int yMax, xMax;
43     getmaxyx(stdscr, yMax, xMax);
44
45     WINDOW *gameWin = newwin(43, xMax-10, (yMax/2) - 24, 5);
46     // fix 43
47     box(gameWin, 0, 0);
```

```
47     refresh();
48     wrefresh(gameWin);
49     keypad(gameWin, true);
50
51     char **dispboard = init_dispboard(gameWin, COLS, ROWS);
52     char **mineboard = init_mineboard(gameWin, COLS, ROWS,
53     NMINES);
54
55     selection(gameWin, dispboard, mineboard, COLS, ROWS,
56     NMINES);
57
58     free(dispboard);
59     free(mineboard);
60 }
61
62 char **init_dispboard(WINDOW *gameWin, int COLS, int ROWS)
63 {
64     int i;
65     char **dispboard = (char **)malloc(COLS * sizeof(char *))
66 );
67     for (i = 0; i < COLS; i++)
68         dispboard[i] = (char *)malloc(ROWS);
69
70     if (dispboard == NULL)
71     {
72         mvprintw(1, 1, "Error, not enough memory, exiting...
73 ");
74         exit(EXIT_FAILURE);
75     }
76     else
77     {
78         fill_dispboard(dispboard, COLS, ROWS);
79         print_board(gameWin, dispboard, COLS, ROWS);
80         getchar();
81     }
82
83     return dispboard;
84 }
85
86 void fill_dispboard(char **dispboard, int COLS, int ROWS)
87 {
88     int i, j;
89
90     for (i = 0; i < COLS; i++)
91         for (j = 0; j < ROWS; j++)
92             dispboard[i][j] = HIDDEN;
```

```
93
94 char **init_mineboard(WINDOW *gameWin, int COLS, int ROWS,
    int NMINES)
95 {
96     int i;
97     char **mineboard = (char **)malloc(COLS * sizeof(char *)
    );
98     for (i = 0; i < COLS; i++)
99         mineboard[i] = (char *)malloc(ROWS);
100
101     if (mineboard == NULL)
102     {
103         mvprintw(1, 1, "Error, not enough memory, exiting...
    ");
104         exit(EXIT_FAILURE);
105     }
106     else
107     {
108         place_mines(mineboard, COLS, ROWS, NMINES);
109         add_adj(mineboard, COLS, ROWS);
110         fill_spaces(mineboard, COLS, ROWS, NMINES);
111
112         // tests
113         //print_board(gameWin, mineboard, COLS, ROWS);
114         //filewrite(mineboard, COLS, ROWS, 1, 2);
115     }
116
117     return mineboard;
118 }
119
120
121 void place_mines(char **mineboard, int COLS, int ROWS, int
    NMINES)
122 {
123     int i, wRand, hRand;
124
125     srand(time(NULL));
126
127     for (i = 0; i < NMINES; i++)
128     {
129         wRand = rand() % COLS;
130         hRand = rand() % ROWS;
131         mineboard[wRand][hRand] = MINE;
132     }
133 }
134
135
136 void add_adj(char **mineboard, int COLS, int ROWS)
137 {
138     int i, j;
```

```

139
140     for (i = 0; i < COLS; i++)
141         for (j = 0; j < ROWS; j++)
142             if (!is_mine(mineboard, i, j))
143                 mineboard[i][j] = adj_mines(mineboard, i, j,
144                     COLS, ROWS) + '0';
145 }
146
147 bool is_mine(char **mineboard, int row, int col)
148 {
149     return (mineboard[row][col] == MINE) ? true : false;
150 }
151
152 bool outof_bounds(int row, int col, int COLS, int ROWS)
153 {
154     return (row < 0 || row > COLS-1 || col < 0 || col > ROWS
155         -1) ? true : false;
156 }
157
158
159 int8_t adj_mines(char **mineboard, int row, int col, int
160     COLS, int ROWS)
161 {
162     int8_t numAdj = 0;
163
164     if (!outof_bounds(row, col - 1, COLS, ROWS) &&
165         mineboard[row][col-1] == MINE) numAdj++; // North
166     if (!outof_bounds(row, col + 1, COLS, ROWS) &&
167         mineboard[row][col+1] == MINE) numAdj++; // South
168     if (!outof_bounds(row + 1, col, COLS, ROWS) &&
169         mineboard[row+1][col] == MINE) numAdj++; // East
170     if (!outof_bounds(row - 1, col, COLS, ROWS) &&
171         mineboard[row-1][col] == MINE) numAdj++; // West
172     if (!outof_bounds(row + 1, col - 1, COLS, ROWS) &&
173         mineboard[row+1][col-1] == MINE) numAdj++; // North-East
174     if (!outof_bounds(row - 1, col - 1, COLS, ROWS) &&
175         mineboard[row-1][col-1] == MINE) numAdj++; // North-West
176     if (!outof_bounds(row + 1, col + 1, COLS, ROWS) &&
177         mineboard[row+1][col+1] == MINE) numAdj++; // South-East
178     if (!outof_bounds(row - 1, col + 1, COLS, ROWS) &&
179         mineboard[row-1][col+1] == MINE) numAdj++; // South-West
180
181     return numAdj;
182 }
183
184
185 void fill_spaces(char **mineboard, int COLS, int ROWS, int
186     NMINES)

```

```

177 {
178     int i, j;
179
180     for (i = 0; i < COLS; i++)
181         for (j = 0; j < ROWS; j++)
182             if (mineboard[i][j] != MINE && mineboard[i][j] =
183                 = '0')
184                 mineboard[i][j] = '-';
185 }

```

5.5 gameplay.h

```

1 #ifndef GAMEPLAY_H
2 #define GAMEPLAY_H
3
4 #include "minesweeper.h"
5
6 #define DEFUSEKEY 'd'
7
8 void selection(struct _win_st*, char **, char **, int, int,
9               int);
10 bool transfer(char **, char **, int, int, int, int *);
11 void reveal(struct _win_st*, char **, int, int);
12 bool defused(char **, char **, int, int, int, int *);
13 #endif

```

5.6 settings.c

```

1 #include "settings.h"
2
3 void set_mode(WINDOW *menuWin) // loop
4 {
5     char mode;
6     mvwprintw(menuWin, 1, 1, "Keyboard or text mode (k/t): "
7 );
8     wrefresh(menuWin);
9     scanw("%c", &mode);
10    mvwprintw(menuWin, 1, strlen("Keyboard or text mode (k/t
11 ): ") + 1, "%c", mode);
12    wrefresh(menuWin);
13    mvwprintw(menuWin, 1, 1, CLEAR); // thanks stefastra &&
14    spyrosROUM!!!! :-DDDD
15    wrefresh(menuWin);
16
17    switch (mode)
18    {
19        case 'k':
20        case 'K':
21            mvwprintw(menuWin, 2, 1, "Keyboard mode");

```



```
19         wrefresh(menuWin);
20         break;
21     case 't':
22     case 'T':
23         mvwprintw(menuWin, 2, 1, "Text mode");
24         wrefresh(menuWin);
25         break;
26     default:
27         break;
28 }
29 }
30
31
32 int set_cols(WINDOW *menuWin, int xMax)
33 {
34     int COLS;
35
36     do
37     {
38         mvwprintw(menuWin, 1, 1, "Columns (Max = %d): ",
39 xMax-12);
40         wrefresh(menuWin);
41         scanw("%d", &COLS);
42         mvwprintw(menuWin, 1, strlen("Columns (Max = XXX): ")
43 ) + 1, "%d", COLS);
44         wrefresh(menuWin);
45     } while (COLS < 5 || COLS > xMax - 12);
46
47     return COLS;
48 }
49
50 int set_rows(WINDOW *menuWin, int yMax)
51 {
52     int ROWS;
53
54     do
55     {
56         mvwprintw(menuWin, 2, 1, "Rows (Max = %d): ", yMax-1
57 2);
58         wrefresh(menuWin);
59         scanw("%d", &ROWS);
60         mvwprintw(menuWin, 2, strlen("Rows (Max = YYY): ") +
61 1, "%d", ROWS);
62         wrefresh(menuWin);
63     } while (ROWS < 5 || ROWS > yMax - 12);
64
65     return ROWS;
66 }
```

```

65
66 int set_nmines(WINDOW *menuWin, int DIMENSIONS)
67 {
68     int NMINES;
69
70     do
71     {
72         mvwprintw(menuWin, 3, 1, "Mines (Max = %d): ",
73                     DIMENSIONS-10); // -10 so the player has a chance to win
74         wrefresh(menuWin);
75         scanw("%d", &NMINES);
76         mvwprintw(menuWin, 3, strlen("Mines (Max = MMMM): ")
77                     + 1, "%d", NMINES);
78         wrefresh(menuWin);
79     } while (NMINES < 1 || NMINES > DIMENSIONS-10);
80
81     return NMINES;
82 }

```

5.7 settings.h

```

1 #ifndef SETTINGS_H
2 #define SETTINGS_H
3
4 #include "minesweeper.h"
5
6 void set_mode(struct _win_st*);
7 int set_cols(struct _win_st*, int);
8 int set_rows(struct _win_st*, int);
9 int set_nmines(struct _win_st*, int);
10
11 #endif

```

5.8 outputs.c

```

1 #include "outputs.h"
2
3 void game_won(WINDOW *gameWin, char **mineboard, int yMiddle,
4               int xMiddle)
5 {
6     wclear(gameWin);
7     mvwprintw(gameWin, yMiddle-11, xMiddle-18, "You defused
8     all the mines!");
9     mvwprintw(gameWin, yMiddle-10, xMiddle-10, "You won :)");
10    ;
11    wrefresh(gameWin);
12    wclear(gameWin);
13 }

```

```
13 void game_over(WINDOW *gameWin, char **mineboard, int
    yMiddle, int xMiddle)
14 {
15     wclear(gameWin);
16     mvwprintw(gameWin, yMiddle-11, xMiddle-11, "You hit a
mine!");
17     mvwprintw(gameWin, yMiddle-10, xMiddle-10, "Game over :(");
18     wrefresh(gameWin);
19     wclear(gameWin);
20 }
21
22
23 void print_board(WINDOW *gameWin, char **mineboard, int COLS
    , int ROWS)
24 {
25     int i, j;
26
27     for (i = 0; i < ROWS; i++)
28     {
29         for (j = 0; j < COLS; j++)
30         {
31             mvwaddch(gameWin, j+1, i+1, mineboard[i][j]);
32             wrefresh(gameWin);
33         }
34     }
35 }
36
37
38 void filewrite(char **mineboard, int COLS, int ROWS, int
    hitCol, int hitRow, const char *status)
39 {
40     int i, j;
41     FILE *mnsOut = fopen("mnsout.txt", "w");
42
43     if (mnsOut == NULL)
44     {
45         mvprintw(1, 1, "Error opening file, exiting...");
46         exit(EXIT_FAILURE);
47     }
48     else
49     {
50         strcmp(status, "won")
51         ? fprintf(mnsOut, "Mine hit at position (%d, %d)
\n\n", hitCol, hitRow)
52         : fprintf(mnsOut, "Last mine defused at position
(%d, %d)\n\n", hitCol, hitRow);
53         fprintf(mnsOut, "Board overview\n\n");
54
55         for (i = 0; i < ROWS; i++)
```

```
56     {
57         for (j = 0; j < COLS; j++)
58             fprintf(mnsOut, "%c ", mineboard[j][i]);
59         fprintf(mnsOut, "\n");
60     }
61
62     mvprintw(1, 1, "Session written to file");
63     refresh();
64 }
65
66 fclose(mnsOut);
67 }
```

5.9 outputs.h

```
1  #ifndef OUTPUTS_H
2  #define OUTPUTS_H
3
4  #include "minesweeper.h"
5
6  void game_won(struct _win_st*, char **, int, int);
7  void game_over(struct _win_st*, char **, int, int);
8
9  void print_board(struct _win_st*, char **, int, int);
10 void filewrite(char **, int, int, int, int, const char *);
11
12 #endif
```

5.10 Διάγραμμα ροής

5.11 Περιγραφή υλοποίησης

6 Διευκρινήσεις

7 Εργαλεία

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