ECE264 Spring 2014
Exam 2, March 11, 2014

In signing this statement, I hereby certify that the work on this exam is my own and that I have not copied the work of any other student while completing it. I understand that, if I fail to honor this agreement, I will receive a score of ZERO for this exam and will be subject to possible disciplinary action.

Signature:

You must sign here. Otherwise you will receive a **2-point** *penalty.*

Read the questions carefully.
Some questions have conditions and restrictions.

This is an *open-book, open-note* exam. You may use any book, notes, or program printouts. No personal electronic device is allowed. You may not borrow books from other students.

Two learning objectives (recursion and structure) are tested in this exam. To pass an objective, you must receive 50% or more points in this exam.

Contents

Learning Objective 1 (Recursion)	Pass	Fail
Learning Objective 2 (Structure)	Pass	Fail

Total Score:

This page is blank. You can write answers here if you need more space.

This page is blank. You can write answers here if you need more space.

1 Recursion 1 Recursive Equation (5 points)

There are unlimited red (ℝ), green (𝔾), and blue (𝔹) balls. You need to select n balls. Two adjacent balls cannot be both red or both green. The orders matter: ℝ𝔹 and 𝔹ℝ are different. When n is one, there are three options:

1. ℝ
2. 𝔾
3. 𝔹

When n is two, there are seven options. Please notice that ℝ ℝ and 𝔾 𝔾 are invalid options.

1. ℝ 𝔾
2. ℝ 𝔹
3. 𝔾 ℝ
4. 𝔾 𝔹
5. 𝔹 ℝ
6. 𝔹 𝔾
7. 𝔹 𝔹

Q1 Suppose $r(n)$, $g(n)$, and $b(n)$ are the number of options when selecting n balls and the first ball is ℝ, 𝔾, and 𝔹 respectively. Fill this table (3 points, 0.5 point for each answer).

n	1	2	3	4	5
r(n)	1	2			
g(n)	1	2	same as r(3)	same as r(4)	same as r(5)
b(n)	1	3			

Q2 How many options are there when selecting **5** balls? (2 points)
You can write an expression without writing the final answer. For example, you may write 1 + 2 + 9 + 13 without writing 25.

2 Recursion 2: Recursive Function (5 points)

Consider the following incorrect implementation of recursive binary search.

```
1  // search.c
2  int binarysearch(int * arr, int key, int low, int high)
3  {
4    if (low > high)
5      {
6        return -1;
7      }
8    int mid = (low + high) / 2;
9    if (arr[mid] == key)
10     {
11       return mid;
12     }
13   if (arr[mid] > key)
14     {
15       return binarysearch(arr, key, low, mid - 1);
16     }
17   return binarysearch(arr, key, mid, high); // ERROR
18   // ERROR, should be mid + 1 but it is mid
19 }
```

The `main` function is shown below.

```
1  // main.c
2  #include <stdio.h>
3  #include <stdlib.h>
4  int binarysearch(int * arr, int key, int low, int high);
5  #define ARRAYSIZE 10
6  int main(int argc, char * * argv)
7  {
8    int arr[ARRAYSIZE] = {1, 12, 23, 44, 65, 76, 77, 98, 109, 110};
9    int ind;
10   for (ind = 0; ind < ARRAYSIZE; ind ++)
11     {
12       printf("%d\n", binarysearch(arr, arr[ind], 0, ARRAYSIZE));
13
14     }
15   return EXIT_SUCCESS;
16 }
```

One line in `binarysearch` is incorrect, as marked by a comment. If `binarysearch` were correct, the expected output would be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Because of this mistake, the

recursive calls will not end for **some** cases. For each of these cases, the program eventually runs out of the memory for the callstack.

Q1 Please write down the **smallest** value of `ind` in `main` when the program runs out of the memory for the callstack. (2 points)

Q2 Write down the values of `key`, `low`, and `high` at the top frame when the program runs out the memory for the callstack. (3 points, 1 point for each answer)

3 Structure 1: Read and Write Data (5 points)

Consider the following structure for a two-dimensional array.

```
 1 #ifndef ARRAY_H
 2 #define ARRAY_H
 3 // tell compiler not to pad any space
 4 #pragma pack(1)
 5 typedef struct
 6 {
 7   int length;
 8   int * data;
 9 } Array;
10 #endif
```

The following is a program using the structure.

```
 1 // for simplicity, this program does not handle errors
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5 #include "array.h"
 6 int main(int argc, char **argv)
 7 {
 8   int length = 10;
 9   char * filename = "data";
10   // create an object
11   Array * arrptr1 = NULL;
12   printf("sizeof(arrptr1) = %d\n", (int) sizeof(arrptr1));
13   arrptr1 = malloc(sizeof(Array));
14   printf("sizeof(arrptr1) = %d, sizeof(Array) = %d\n",
15          (int) sizeof(arrptr1), (int) sizeof(Array));
16   // allocate memory for the data
17   arrptr1 -> length = length;
18   arrptr1 -> data = malloc(sizeof(int) * (arrptr1 -> length));
19   printf("sizeof(arrptr1) = %d, sizeof(arrptr1 -> data) = %d\n",
20          (int) sizeof(arrptr1), (int) sizeof(arrptr1 -> data));
21   // initialize the values of the array
22   int ind;
23   for (ind = 0; ind < (arrptr1 -> length); ind ++)
24     {
25       arrptr1 -> data[ind] = ind;
26     }
27   // save the data to a file
28   FILE * fptr = fopen(filename, "w");
```

```
29    // write the data to the file
30    if (fwrite(arrptr1, sizeof(Array), 1, fptr) != 1)
31      {
32        // fwrite fail
33        return EXIT_FAILURE;
34      }
35    printf("ftell(fptr) = %d\n", (int) ftell(fptr));
36    fclose (fptr);
37
38    // fill the array with random numbers
39    // ensure the heap contains garbage before releasing it
40    srand(time(NULL)); // set the seed of the random number
41    for (ind = 0; ind < (arrptr1 -> length); ind ++)
42      {
43        arrptr1 -> data[ind] = rand();
44      }
45
46    // release memory
47    free(arrptr1 -> data);
48    free(arrptr1);
49    // read the data from the file
50    Array * arrptr2 = NULL;
51    arrptr2 = malloc(sizeof(Array));
52    fptr = fopen(filename, "r");
53    if (fread(arrptr2, sizeof(Array), 1, fptr) != 1)
54      {
55        // fread fail
56        return EXIT_FAILURE;
57      }
58    // add the data
59    int sum = 0;
60    for (ind = 0; ind < (arrptr2 -> length); ind ++)
61      {
62        sum += arrptr2 -> data[ind];
63      }
64    printf("sum = %d\n", sum);
65    // release memory
66    free(arrptr2);
67    return EXIT_SUCCESS;
68  }
```

Assume this program runs on a 64-bit (8 bytes) processor and one integer uses 4 bytes. Assume that the program never returns `EXIT_FAILURE`.

Q1 What is the output of this program? (3.5 points, 0.5 each answer)
If the program crashes before printing a particular line, please write "Do not reach here". If a variable is not initialized, please write "garbage".

```
sizeof(arrptr1) =

sizeof(arrptr1) =          , sizeof(Array) =

sizeof(arrptr1) =          , sizeof(arrptr1 -> data) =

ftell(fptr) =

sum =
```

Q2 `Valgrind` reports one (or more) memory error. Which line causes this error? (0.5 point) Explain the reason (1 point).

The following information is for your reference.

```
long ftell(FILE *stream);

       The ftell() function obtains the current value  of  the  file  position
       indicator for the stream pointed to by stream.
```

4 Structure 2: Objects and Pointers (5 points)

Fill in the missing numbers in the program's output.

```
swapT1----------------------------
t: x =  2, y =  6
u: x = -4, y = -3
t: x =   , y =
u: x =   , y =
swapT2----------------------------
t: x =  2, y =  6
u: x = -4, y = -3
t: x =   , y =
u: x =   , y =
swapT3----------------------------
t: x =  2, y =  6
u: x = -4, y = -3
t: x =   , y =
u: x =   , y =
swapT4----------------------------
t: x =  2, y =  6
u: x = -4, y = -3
t: x =   , y =
u: x =   , y =
```

Please see the code below and fill in the missing numbers.

```
1 #include <stdio.h>
2 #include <stdlib.h>
```

```
 3
 4 typedef struct
 5 {
 6   int * x;
 7   int * y;
 8 } VectorT;
 9
10 void printT1(const char * label, VectorT a)
11 {
12   printf("%s: x = %2d, y = %2d\n",
13          label, * (a.x), *(a.y));
14 }
15
16 void printT2(const char * label, VectorT * a)
17 {
18   printf("%s: x = %2d, y = %2d\n",
19          label, * (a -> x), *(a -> y));
20 }
21
22 void swapT1(VectorT a, VectorT b)
23 {
24   VectorT tmp = a;
25   a = b;
26   b = tmp;
27 }
28
29 void swapT2(VectorT *a, VectorT *b)
30 {
31   VectorT tmp = *a;
32   *a = *b;
33   *b = tmp;
34 }
35
36 void swapT3(VectorT a, VectorT b)
37 {
38   int * tmp = a.x;
39   a.x = b.x;
40   b.x = tmp;
41   tmp = a.y;
42   a.y = b.y;
43   b.y = tmp;
44 }
```

```
45
46 void swapT4(VectorT *a, VectorT *b)
47 {
48   int tmp = * (a -> x);
49   * (a -> x) = * (b -> x);
50   * (b -> x) = tmp;
51   tmp = * (a -> y);
52   * (a -> y) = * (b -> y);
53   * (b -> y) = tmp;
54 }
55
56 int main(int argc, char **argv)
57 {
58   printf("swapT1----------------------------\n");
59   VectorT t;
60   VectorT u;
61   t.x = malloc(sizeof(int));
62   t.y = malloc(sizeof(int));
63   u.x = malloc(sizeof(int));
64   u.y = malloc(sizeof(int));
65   * (t.x) = 2;
66   * (t.y) = 6;
67   * (u.x) = -4;
68   * (u.y) = -3;
69   printT1("t", t);
70   printT1("u", u);
71   swapT1(t, u);
72   printT1("t", t);
73   printT1("u", u);
74
75   printf("swapT2----------------------------\n");
76   * (t.x) = 2;
77   * (t.y) = 6;
78   * (u.x) = -4;
79   * (u.y) = -3;
80   printT1("t", t);
81   printT1("u", u);
82   swapT2(& t, & u);
83   printT1("t", t);
84   printT1("u", u);
85
86   printf("swapT3----------------------------\n");
```

```
 87    * (t.x) = 2;
 88    * (t.y) = 6;
 89    * (u.x) = -4;
 90    * (u.y) = -3;
 91    printT1("t", t);
 92    printT1("u", u);
 93    swapT3(t, u);
 94    printT1("t", t);
 95    printT1("u", u);
 96
 97    printf("swapT4----------------------------\n");
 98    * (t.x) = 2;
 99    * (t.y) = 6;
100    * (u.x) = -4;
101    * (u.y) = -3;
102    printT1("t", t);
103    printT1("u", u);
104    swapT4(& t, & u);
105    printT1("t", t);
106    printT1("u", u);
107
108    // release memory
109    free (t.x);
110    free (t.y);
111    free (u.x);
112    free (u.y);
113
114    return EXIT_SUCCESS;
115 }
```