Name: Section: Rollno:

ESC101A: Fundamentals of Computing (End Semester Exam - A)

24 April, 2015

Total Number of Pages: 16 Total Points 75

Instructions: Read Carefully.

1. Write you roll number on all the pages of the answer book and the extra sheets.
2. Write the answers cleanly in the space provided.
3. **Use pens (blue/black ink) and not pencils** for answering. Do not use red pens.
4. Even if no answers are written, the answer book has to be returned back with name and roll number written.

Question	Points	Score
1	9	
2	12	
3	9	
4	10	
5	10	
6	14	
7	11	
Total:	75	

Helpful hints

1. The questions are *not* arranged according to the increasing order of difficulty. Do a quick first round where you answer the easy ones and leave the difficult ones for the subsequent rounds.
2. All blanks may NOT carry equal marks.
3. You can ask the invigilator for **quick reference sheet** related to C programming. No other C related help will be provided.

I pledge my honour as a gentleman/lady that during the examination I have neither given assistance nor received assistance.

Signature

Space below to be used rechecking requests

Question 1. (9 points) Binary search is an efficient algorithm for searching an item in a sorted list of items. In class we discussed the pseudo code for the binary search. Here, you are provided the following **buggy** iterative code for binary search.

```
1  #include<stdio.h>
2
3  /* Iteratively search for "key" in array "a" having "size" elements */
4
5  int bsearch(int a[], int size, int key) {
6      int left = 0;
7      int right = size - 1;
8      int mid;
9      while (left < right) {
10         mid = (left + right)/2;
11         if (a[mid] == key) return mid;
12         if (a[mid] < key) left = mid+1;
13         else right = mid -1;
14     }
15     return -1;
16 }
17
18 int main()
19 {
20     int B[] = {3,3,3,4,4}, size = 5;
21
22     printf("bsearch(3) = %d\n" , bsearch(B, size, 3)); // prints 2
23     return 0;
24 }
```

Your job is

1. Give an example, i.e., an **array** (all the elements) and a **key**, for which the code will not work.
2. Suggest minimum repair to fix the code. The repair can change multiple lines. You must mention the line numbers and the changes.

Example	**Array:** _____ **Key:** _____
Suggested Repair	**Line #** | **Change**

Solution: Example: Array : [1], Key: 1 (Other also possible)
Repair: while (left <= right) on line 9

Question 2. (12 points) The following program contains an incomplete function `rmbsearch`. It is a modified "Binary Search" that computes the index of the **right-most** occurrence of a **key** in a sorted array **a** of **size** elements.

Complete the function. The function must run in time proportional to log of the size of the array.

(**NOTE:** The printf-s are provided for you to trace your program *manually* on sample inputs. You **DO NOT** have to give the output).

```
#include<stdio.h>

/* Returns the rightmost occurrence of the key in an array.
 * If the key is not found, it returns -1.
 * IDEA: The key is in the region a[left ... right] */

int rmbsearch(int a[], int size, int key) {
    int left = 0, right = size - 1, mid;

    while (____________________) {

        mid = ____________________________________________ ;

        printf("left = %d, right = %d, mid = %d\n",
               left, right, mid); // To help in understanding

        if (a[mid] == key) ____________________________________________ ;

        else if (a[mid] < key) left = ____________________ ;

        else right = ____________________ ;
    }

    if (a[left] == key) return ____________________ ;

    else return -1;
}

int main()
{
    int B[] = {3,4,4}; int size = 3;

    printf("index of 3 = %d\n" , rmbsearch(B, size, 3)); // prints 0
    printf("index of 4 = %d\n" , rmbsearch(B, size, 4)); // prints 2
    printf("index of 2 = %d\n" , rmbsearch(B, size, 2)); // prints -1
    printf("index of 5 = %d\n" , rmbsearch(B, size, 5)); // prints -1

    return 0;
}
```

Solution:

```
#include<stdio.h>
```

```c
/* Returns the rightmost occurrence of the key in an array.
 * If the key not found, it returns -1.
 * IDEA: The key is in the region a[left ... right] */
int rmbsearch(int a[], int size, int key) {
    int left = 0, right = size - 1, mid;

    while (left < right) {

        mid = (left + right+1)/2;

        printf("left = %d, right = %d, mid = %d\n",
               left, right, mid); // DEBUG

        if (a[mid] == key) left = mid;

        else if (a[mid] < key) left = mid+1;

        else right = mid -1;
    }

    if (a[left] == key) return left;

    else return -1;
}

int main()
{
  int B[] = {3,4,4}; int size = 3;

  printf("index of 3 = %d\n" , rmbsearch(B, size, 3)); //prints 0
  printf("index of 4 = %d\n" , rmbsearch(B, size, 4)); //prints 2
  printf("index of 2 = %d\n" , rmbsearch(B, size, 2)); //prints -1
  printf("index of 5 = %d\n" , rmbsearch(B, size, 5)); //prints -1

  return 0;
}
```

Question 3. (9 points) We have discussed merging two sorted arrays in the class. If the data to be merged (the contents of the two sorted arrays) is too huge that it can not fit in arrays, we can use files to store it. In such cases, we can read the data from two files, and store the merged data back in a third file, without using any array.

Given below is an incomplete program which merges the contents of two files and stores the result to a third file. The input files are named **first.in** and **second.in** respectively. The output file is named **result.out**.

Each input file consists of integers in sorted (increasing) order. Complete the program so that it works as desired.

```
#include <stdio.h>
int main() {
    // Open the two files for reading, containing a sorted array each

    FILE *inFile1 = fopen(_____, _____);

    FILE* inFile2 = fopen(_____, _____);

    //  Open file to write the result

    FILE *outFile = fopen(_____, _____);

    // We read the numbers from the files one element at a time,
    // do computations and write result to the outFile

    int n1, n2;

    _____;  // Read n1 from first file

    _____;  // Read n2 from second file

    while (!feof(inFile1) && !feof(inFile2)) {

        if (_____) {

            fprintf(outFile, "%d\n", n1);

            _____;
        } else {

            fprintf(outFile, "%d\n", n2);

            _____;
        }
    }

    // Continued On Next Page ...

```

```
39      // If the first file has become empty, then we write all the
40      // elements of the second file to the outFile
41
42      if (feof(inFile1)) {
43
44          while(!feof(inFile2)) {
45
46              fprintf(outFile, "%d\n", n2);
47
48              ____________________________________________________________;
49          }
50      }
51
52      // If the second file has become empty, then we write all the
53      // elements of the first file to the outFile
54
55      if (feof(inFile2)) {
56
57          while(!feof(inFile1)) {
58
59              fprintf(outFile, "%d\n", n1);
60
61              ____________________________________________________________;
62          }
63      }
64
65      // time to be a responsible citizen: release the file resources
66
67      ____________________________________________;
68
69      ____________________________________________;
70
71      ____________________________________________;
72
73      return 0;
74  }
```

Solution:

```c
#include<stdio.h>
int main() {
    /* Open the two files, these files contain
       two sorted arrays which do not fit in memory */
    FILE *inputFile1 = fopen("first.in", "r");
    FILE* inputFile2 = fopen("second.in", "r");

    /** As the result also cannot be held in memory,
        we write the result to a file */
    FILE *outputFile = fopen("result.out", "w");

    /** We scan the files one element at a time, compare them
      and write the smaller of the two numbers to the outputFile */
    int n1, n2;
    fscanf(inputFile1, "%d", &n1);
    fscanf(inputFile2, "%d", &n2);
    while (!feof(inputFile1) && !feof(inputFile2)) {
        if (n1 <= n2) {
            fprintf(outputFile, "%d\n", n1);
            fscanf(inputFile1, "%d", &n1);
        } else {
            fprintf(outputFile, "%d\n", n2);
            fscanf(inputFile2, "%d", &n2);
        }
    }

    /** If the first file has become empty, then we write
        all the elements of the second file to the outputFile */
    if (feof(inputFile1)) {
        while(!feof(inputFile2)) {
            fprintf(outputFile, "%d\n", n2);
            fscanf(inputFile2, "%d", &n2);
        }
    }
    /** If the second file has become empty, then we write
        all the elements of the first file to the outputFile */
    if (feof(inputFile2)) {
        while(!feof(inputFile1)) {
            fprintf(outputFile, "%d\n", n1);
            fscanf(inputFile1, "%d", &n1);
        }
    }

    fclose(inputFile1);
    fclose(inputFile2);
    fclose(outputFile);
    return 0;
}
```

Question 4. (10 points) The h**-index** is a number based on the set of a scientist's most cited papers. It is defined as follows : A scientist has index h if h of his N papers have at least h citations each, and the other $(N - h)$ papers have at most h citations each.

Albert Einstein, for example, published 319 papers in scientific journals and has an h-index equal to 46. It means 46 of his papers have received 46 or more citations each, and all of his remaining 273 papers have 46 citations or less each. Given the information of how many citations each paper from a given researcher has received, fill in the following function to return the h-index of the researcher.

```
1  // The function takes an array A as the argument, where A[i]
2  // is the number of citations of the researcher's i^th
3  // paper. The function also takes n as the argument, where n
4  // is the number of papers (== length of array A)
5
6  int compute_h_index(int A[], int n)
7  {
8      int i, h_index;
9
10     sort(A,n); // assume the sort function sorts the array A
11                // in increasing order
12
13     for(i=_____; i>=0; i--)
14     {
15
16         if(_____)
17
18             _____;
19     }
20
21     h_index = _____;
22
23     return h_index;
24 }
```

Solution:

```
1  int compute_h_index(int A[], int n)
2  {
3      sort(A,n); //assume the sort function sorts the array A
4
5      int h_index=0;
6      for(int i=n-1; i>=0; i--)
7      {
8          if(A[i] < n-i)
9              break;
10     }
11     h_index = n-1-i;
12 }
```

(FYI: The h-index was suggested in 2005 by **Jorge E. Hirsch** and is sometimes called the Hirsch index or Hirsch number.)

Question 5. (10 points) Given below is the code of a program which is compiled to create an executable file **foo.out**. The program takes inputs on command line. What is the output that the program produces for the given command line instructions? (You may assume that the executable file is available in the current working directory.)

Note that there are **2 printf**-s in the program. In case the possibility of a run time error exists (including segmentation fault), write **Error**.

```
1  #include<stdio.h>
2  char f(char a, char b, char c)  {
3      printf("f(%c, %c, %c)\n", a, b, c); // PRINTING HERE
4
5      if (a >= b)
6          if (b >= c) return a;
7          else return f(a, c, b);
8      else return f(b, a, c);
9
10 }
11
12 int main(int argc, char** argv)
13 {
14     char res = f(argv[1][0], argv[1][1], argv[1][2]);
15
16     printf("%c\n", res);               // PRINTING HERE
17
18     return 0;
19 }
```

Command	**Output**
./foo.out quick brown fox	
./foo.out	
./foo.out azure	

Solution:

(a)	(b)	(c)
f(q, u, i) f(u, q, i) u	**Error**	f(a, z, u) f(z, a, u) f(z, u, a) z

Question 6. (14 points) Complete the output of the following program.

```
#include<stdio.h>
#include<stdlib.h>
typedef struct _node {
    int data;
    struct _node* next;
} node;

      // Create a new node
node* makeNode(int data)
{
    node *n = malloc(sizeof(node));
    n->data = data;
    n->next = NULL;
    return n;
}

node* processLists(node* first, node* second)
{
    node* res = NULL; // res is the head of the result list
    node *temp, *prev = NULL;
    int x = 0, data;

    while (first || second) { // while atleast one list exists
        data = x + (first ? first->data : 0)
                 + (second? second->data: 0);

           // update x and data
        x = (data >= 16)? 1 : 0;
        data = data % 16;

           // Create a new node with this data, and add to res list
        temp = makeNode(data);
        if(res == NULL)
            res = temp;
        else
            prev->next = temp;

        prev  = temp;

           // Move first and second pointers to next nodes
        if (first) first = first->next;
        if (second) second = second->next;
    }

    if (x > 0)
      temp->next = makeNode(x);  // temp cannot be NULL

    return res;
}
    // Continued On Next Page ...
```

```
51      // Insert a node in a Linked List
52 void push(node** head_ref, int new_data)
53 {
54     node* new_node = makeNode(new_data);
55     new_node->next = (*head_ref);
56     (*head_ref)    = new_node;
57 }
58
59      //  Print a linked list
60 void printList(node *node)
61 {
62     while(node != NULL) {
63         printf("%d ", node->data);
64         node = node->next;
65     }
66     printf("\n");
67 }
68
69 int main()
70 {
71     node *res = NULL, *first = NULL, *second = NULL;
72
73     push(&first, 4);
74     push(&first, 9);
75     push(&first, 5);
76     push(&first, 7);
77     printf("First List is ");  // PRINTING HERE
78     printList(first);          // & HERE
79
80     push(&second, 12);
81     push(&second, 8);
82     printf("Second List is "); // PRINTING HERE
83     printList(second);         // & HERE
84
85     res = processLists(first, second);
86     printf("Result is "); // PRINTING HERE
87     printList(res);       // & HERE
88
89     // free the lists: code not shown
90     return 0;
91 }
```

OUTPUT

First List is _____

Second List is _____

Result is _____

Solution:
First List is 7 5 9 4
Second List is 8 12
Result is 15 1 10 4

Question 7. (11 points) An n-length Gray code $G(n)$ is a list of the 2^n different n-length sequences of 0s and 1s such that each entry in the list differs in precisely one digit from its predecessor. For example, $G(1)$, $G(2)$ and $G(3)$ are :

$G(1)$	$G(2)$	$G(3)$
0	0 0	0 0 0
1	0 1	0 0 1
	1 1	0 1 1
	1 0	0 1 0
		1 1 0
		1 1 1
		1 0 1
		1 0 0

How do we generate a Gray code $G(n)$? A recursive plan to generate n-bit Gray code is as follows:

1. Generate $G(n-1)$, the gray code of $n-1$ length.
2. Attach 0 at the start of each word in $G(n-1)$, followed by
3. Attach 1 at the start of each word in $G(n-1)$ in **reverse order**.

The 0-length code is defined to be **empty**, and the 1-length code is a list of 2 numbers: 0 followed by 1.

Complete the following code to compute Gray code according to the given plan. The function

```
void gray(int n, char** prefix)
```

computes $G(n)$ and stores it in the array of strings **prefix**.

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<stdlib.h>
4  #include<math.h>
5
6  void gray(int n, char** prefix); // declaration
7
8  int main() {
9      int n;
10     scanf("%d", &n);
11     int i;
12     int rows = (int) pow(2, n); // rows in G(n) = 2^n
13
14     char** graycode = (char**)_____________________________________________;
15     for(i = 0; i < rows; i++) {
16
17         graycode[i] = _____________________________________________________;
18     }
19     gray(n, graycode);
20     for(i = 0; i < rows; i++) {
21         printf("%s\n", graycode[i]);
22     }
23     return 0;
24 }
25
26 // Continued On Next Page ...
27
```

```
28
29 void gray(int n, char** prefix) {
30
31     if (_____________________) { // done, return empty string
32         prefix[0][0] = '\0';
33         return;
34     }
35     int i;
36
37     // compute the number of rows required to store G(n-1) = 2^(n-1)
38
39     int rows = (int) pow(2, n-1);
40
41     // Allocate space in tmpCode to hold G(n-1)
42
43     char** tmpCode = (char**)_____________________________________________;
44     for(i = 0; i < rows; i++) {
45
46         tmpCode[i] = ____________________________________________________;
47     }
48
49     // Compute G(n-1)
50     gray(n-1, tmpCode);
51
52     // now create G(n) from G(n-1), using step 2 and 3 of the plan.
53
54     for (i = 0; i < rows; i++) {
55         strcpy(prefix[i], "0");
56         strcat(prefix[i], tmpCode[i]);
57     }
58
59     for (i = 0; i < rows; i++) {
60
61         strcpy(_____________________, _____________________);
62
63         strcat(_____________________, _____________________);
64     }
65
66     // free the temporary memory
67     for(i = 0; i < rows; i++) {
68         free(tmpCode[i]);
69     }
70     free(tmpCode);
71     return;
72 }
```

Solution:

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

```
#include<stdlib.h>
#include<math.h>

void gray(int n, char** prefix) {
    if (n == 0) { // done, return empty string
        prefix[0][0] = '\0';
        return;
    }
    int i;
    // compute the number of rows required to store gray(n-1) = 2^(n-1)
    int rows = (int) pow(2, n-1);
    char** tmpCode = (char**)malloc(rows*sizeof(char*));
    for(i = 0; i < rows; i++) {
        tmpCode[i] = (char*)malloc(n*sizeof(char));
    }

    // Compute n-1 gray
    gray(n-1, tmpCode);

    // now create gray(n) from gray(n-1), in two steps.
    // First: Prefix '0' in the i^th row followed by i^th
    // row of gray(n-1)
    for (i = 0; i < rows; i++) {
        strcpy(prefix[i], "0");
        strcat(prefix[i], tmpCode[i]);
    }
    // Second: In next rows rows, 1 followed by the row of
    //         gray(n-1) in reverse order
    for (i = 0; i < rows; i++) {
        strcpy(prefix[rows+i], "1");
        strcat(prefix[rows+i], tmpCode[rows-1-i]);
    }

    // free the temporary memory
    for(i = 0; i < rows; i++) {
        free(tmpCode[i]);
    }
    free(tmpCode);
    return;
}

int main() {
  int n;
  scanf("%d", &n);
  int i;
  int rows = (int) pow(2, n);
  char** graycode = (char**)malloc(rows*sizeof(char*));
  for(i = 0; i < rows; i++) {
      graycode[i] = (char*)malloc((n+1)*sizeof(char));
  }
```

```
53    gray(n, graycode);
54    for(i = 0; i < rows; i++) {
55        printf("%s\n", graycode[i]);
56    }
57    return 0;
58 }
```