Indian Institute of Technology, Kharagpur

Department of Computer Science and Engineering

End-Semester Examination, Autumn 2013-14

Programming and Data Structures (CS 11001)

Students: 700 **Date**: 25-Nov-13 (FN)

Full marks: 100 **Time**: 3 hours

Name	Roll No.	Section

Question No.	1	2	3	4	5	6	**Total**
Maximum Marks	40	15	10	10	15	10	100
Marks Obtained							
Evaluator	PG	SD	JM	DS	PPD	AG	

This question paper comprises 6 questions in 16 pages (8 sheets) 01

Instructions:

1. Write your name, roll number and section in the space above.
2. On the top of every odd page write your roll number.
3. Answer all questions in the space provided in this paper itself. No extra sheet will be provided.
4. Use the designated spaces for rough work.
5. Marks for every question is shown with the question.
6. No further clarifications to any of the questions will be provided.

1. Answer the following questions:

   (a) Convert `6f2.5bc` (in hexadecimal system) to the octal number system. [**2 marks**]

```
Answer: 3362.2674
```

   Note for Marking: *1 mark for the integral part and 1 mark for the fractional part.*

   (b) Convert `129.5625` (in decimal system) to the hexadecimal system. [**2 marks**]

```
Answer: 81.9
```

   Note for Marking: *1 mark for the integral part and 1 mark for the fractional part.*

   (c) Represent `-77` (in decimal system) in binary 8-bit signed-magnitude, 1's complement, and 2's complement representations. You need not show the steps of conversion. [**1 * 3 = 3 marks**]

```
Answer:
   -77 = 11001101 (Signed Magnitude)
         --------
   -77 = 10110010 (1's Complement)
         --------
   -77 = 10110011 (2's Complement)
         --------
```

   (d) You are asked to generate the following alternating variant of the Fibonacci series $f_a(n)$, $n \geq 0$:

$$0, -1, 1, -2, 3, -5, 8, -13, \cdots$$

   i. Specify the base conditions (on n) and the recursive expression for $f_a(n)$ in every case to complete the recurrence for $f_a(n)$: [**(0.5 + 0.5) + (0.5 + 0.5) + (1.5 + 0.5) = 4 marks**]

```
Answer:
f_a(n) = 0, for n = 0  [Base Condition 1]
         -      -----
       = -1, for n = 1 [Base Condition 2]
         --     -----
       = f_a(n-2) - f_a(n-1), for n > 1 [Recurrence]
         -------------------      -----
```

   Note for Marking: *Conditions (on n) must be exact. Any equivalent expression for $f_a(n)$ (like $f_a(n) = n$, for $n = 0$ or $f_a(n) = -n$, for $n = 1$) is acceptable in every case. No partial marking in case of the expression for $n > 1$.*

ii. Complete the following code segment that prints $f_a(n)$ for $n = 0, 1, 2, \cdots, m$: [**1 + 2 + 1 = 4 marks**]

```
Answer:
void f_a(int m) {
    int n, x = 0, y = -1;
    for(n = 0; n < m; ++n) {
        int t = y;
                -
        printf("f_a(%d) = %d\n", n, x);
        y = x - y;
            -----
        x = t;
            -
    }
}
```

Note for Marking: `y = x - t` *(or any equivalent expression) is acceptable in place for* `y = x - y`.

(e) `FindMinMax` is a function that takes an array `A` of `int` and the number of elements `n` in `A` as input and returns the minimum as well as maximum values in `A` through parameters.

i. Fill up the parameter types in the prototype of `FindMinMax`. [**1 * 2 = 2 marks**]

```
Answer:
    void FindMinMax(int A[], int n, int* min,   // Parameter for minimum value
                                    --------
                                    int* max);  // Parameter for maximum value
                                    --------
```

Note for Marking: *Just the data type* `int*` *is acceptable as an answer. Variable names should be ignored for marking.*

ii. If `int min` and `int max` are variables to hold the minimum and the maximum values respectively of `int X[]`, write a call to `FindMinMax` to match the above prototype. [**0.5 * 2 = 1 mark**]

```
Answer:    FindMinMax(X, n, &min, &max);
                            ----  ----
```

iii. Alternatively, if the return type of `FindMinMax` is `int` as in the following prototype, fill up the data types in the prototype and the details in the call to `FindMinMax`? [**0.5 * 4 = 2 marks**]

```
Answer:
    // Returns the minimum value as function return value
    // and the maximum value through parameter

    int FindMinMax(int A[], int n, int *max);
    ---                            --------

    // In the caller - int min and int max hold the minimum and
    // the maximum values respectively for int X[]

    min = FindMinMax(X, n, &max);
    ---                    ----
```

(f) Consider the following code snippet:

```
p = (int *)malloc(sizeof(int));
q = &p;
r = &q;
```

i. Write the data types of `p`, `q`, and `r`. [**1 * 3 = 3 marks**]

Answer:

```
int *   p;
-----
int **  q;
------
int *** r;
-------
```

ii. Show how to allocate an array of 20 pointers (of appropriate type) to `r`? [**1 * 2 = 2 marks**]

Answer:

```
r = (int ***)malloc(20*sizeof(int **));
     -------        -----------------
```

Note for Marking: *Any equivalent* `sizeof` *expression like* `20*sizeof(*r)` *or* `20*sizeof(r)` *or* `20*sizeof(int *)` *or* `20*sizeof(void *)` *or any expression using the size of a pointer variable would be acceptable. However, machine-dependent expressions like* `20*4` *or* `80` *would not be acceptable.*

(g) You are sorting the following array in ascending order using `InsertionSort`.

6	2	7	1	3

Show the contents of the array after every iteration of the sort (**Iteration 0** is the input array). [**1 * 4 = 4 marks**]

Answer:

	Index				
Iteration	*0*	*1*	*2*	*3*	*4*
0	6	2	7	1	3
1	2	6	7	1	3
2	2	6	7	1	3
3	1	2	6	7	3
4	1	2	3	6	7

Note for Marking: *Award mark if all elements of the array are correct after each iteration.*

(h) Data are pushed to (`PUSH` operation) and popped from (`POP` operation) a stack in the following order:

`PUSH 3; TOP; PUSH 7; TOP; PUSH 6; PUSH 9; TOP; POP; POP; TOP;`

where the `PUSH`, `POP` and `TOP` operations of stack behave as discussed in the class.
Write the values returned by `TOP` for the sequence of operations above. [**0.5 * 4 = 2 marks**]

```
Answer:
3  7  9  7
-  -  -  -
```

(i) A stack of `int` is implemented using an array as the following data type:

```
#define SIZE 20
typedef struct {
    int data[SIZE];
    int top;
} Stack;
```

Fill up the missing codes in the `PUSH`, `POP`, and `TOP` operations of the Stack. [**1 * 3 = 3 marks**]

```
Answer:
void Push(Stack *s, int d) {
    s->data[++s->top] = d;
    -----------------
}

void Pop(Stack *s) {
    --s->top;
    --------
}

int Top(Stack *s) {
    return s->data[s->top];
           ---------------
}
```

Note for Marking: *Equivalent pointer de-referencing expressions like* `(*s).data` *in place of* `s->data` *are acceptable as long as they are correct and are expressed in single expressions. Hence, statements like* `++s->top; s->data[s->top] = d;` *are not acceptable in place of* `s->data[++s->top] = d;`.

(j) Data are enqueued to (`ENQ` operation) and dequeued from (`DEQ` operation) a queue in the following order:

`ENQ 3; FRONT; ENQ 7; FRONT; ENQ 6; ENQ 9; FRONT; DEQ; DEQ; FRONT;`

where the `ENQ`, `DEQ` and `FRONT` operations of queue behave as discussed in the class.
Write the values returned by `FRONT` for the sequence of operations above. [**0.5 * 4 = 2 marks**]

```
Answer:
3  3  3  6
-  -  -  -
```

(k) A queue of `int` is implemented using a circular array as the following data type:

```
#define SIZE 20
typedef struct {
    int data[SIZE];
    int front, rear;
} Queue;
```

Fill up the missing codes in the `IsEmpty` and `IsFull` operations of the Queue. [**2 * 2 = 4 marks**]

```
Answer:
int IsEmpty(Queue *q) {
    return q->front == q->rear;
           --------------------
}

int IsFull(Queue *q) {
    return q->front == (q->rear+1) % SIZE;
           ------------------------------
}
```

Note for Marking: *Allow 1 mark each for the two sides of* `==` *in the expressions. Deduct 0.5 mark if* `=` *is written in place of* `==`*. Deduct 1 mark if* `!=` *is written in place of* `==`*. Equivalent pointer dereferencing expressions like* `(*q).front` *in place of* `q->front` *are acceptable as long as they are correct and are expressed in single expressions. No other partial marking is allowed.*

Roll No: _____

2. Consider the following polynomial $P(x)$ of degree n in x: $P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0 = \sum_{i=0}^{n} a_i x^i$
We can represent $P(x)$ as a pair of variables – degree `n` and an array `a` of its coefficients as:

```
#define MAX_POLY_SIZE 100   // Maximum degree
double a[MAX_POLY_SIZE+1];  // Array of coefficients
unsigned int n;             // Degree of polynomial
```

(a) `EvalPoly` takes a polynomial $P(x)$ (degree `n` and coefficient array `a`) and `x` as input and returns the value of the polynomial $P(x)$ at x. Fill up the missing codes in function `EvalPoly`. [**1 * 2 = 2 marks**]

```
Answer:
double EvalPoly(double a[],     // Array of coefficients
    unsigned int n,             // Degree of polynomial
    double x) {                 // Value of unknown x for evaluation
    double val = 0.0;           // Value of the polynomial
    double pow_x = 1.0;         // Powers of x
    unsigned int i;             // Index
    for(i = 0; i <= n; ++i) {
        val += a[i]*pow_x;      // Accumulate the next term
               ----------
        pow_x *= x;             // Compute the next power of x
              --
    }
    return val;
}
```

Note for Marking: *Any equivalent expression like interchanging of operands of multiplication (like* `val += pow_x*a[i];`*) are acceptable in place of* `val += a[i]*pow_x;`*. Array indexing by pointer expressions like* `*(a+i)` *for* `a[i]` *is also acceptable.* `pow_x = pow_x * x;` *or any expression equivalent to it is also acceptable in place of* `pow_x *= x;`*.*

(b) The above polynomial can be written in a fully parenthesized form (known as *Horner's Form*) as:

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0 = (\cdots(((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x + \cdots)x + a_0$$

Complete the following implementation of `EvalPoly` using the Horner's Form. [**0.5 * 4 + 1 = 3 marks**]

```
Answer:
double EvalPoly(double a[],     // Array of coefficients
    unsigned int n,             // Degree of polynomial
    double x) {                 // Value of unknown x for evaluation
    double val = 0.0;           // Value of the polynomial
    unsigned int i;             // Index
    val = a[n];                 // Initial value
          ----
    for(i = n; i > 0; --i) {
            -  -----  ---
        val = val*x+a[i-1];     // Iterative accumulation of value
              ------------
    }
    return val;
}
```

Note for Marking: *Array indexing by pointer (like* `*(a+i)` *for* `a[i]`*) and re-ordering of operands (like* `0 < i` *for* `i > 0`*), equivalent expressions (like* `i--` *for* `--i`*) are acceptable if correct.*

(c) Complete the function `AddPoly` to add two polynomials `a` and `b` (as degree and coefficient array pair) and generate a sum polynomial `c`: [**1 * 5 = 5 marks**]

```
Answer:
unsigned int                    // Degree of the sum polynomial
AddPoly(
    double a[], unsigned int m, // First polynomial
    double b[], unsigned int n, // Second polynomial
    double c[]) {               // Coefficients of the sum polynomial
    unsigned int i = 0;         // Index

    while (i <= m && i <= n)    // Add common terms
           ------    ------
        c[i++] = a[i]+b[i];
    while (i <= m)              // Handle extra terms of first polynomial
        c[i++] = a[i];
        -------------
    while (i <= n)              // Handle extra terms of second polynomial
           ------
        c[i++] = b[i];

    return (m > n)? m: n;       // Return the degree of sum polynomial
           -------------
}
```

Note for Marking: *Array indexing by pointer (like* `*(a+i)` *for* `a[i]`*), re-ordering of operands (like* `m >= i` *for* `i <= m`*) are acceptable if correct. Return value may be* `i-1` *or* `--i` *in place of* `(m>n)?m:n`*.*

(d) Consider the function `EvalPolyDerivative` that takes a polynomial $P(x)$ (degree `n` and coefficient array `a`) and `x` as input and returns the value of the derivative $dP(x)/dx$ at x. For example, if $P(x) = 7x^3 - 2x^2 + 5x + 2$, `EvalPolyDerivative` will evaluate $P'(x) = 21x^2 - 4x + 5$. Fill up the missing codes in `EvalPolyDerivative`. You may use the functions written above. [**1 + 2 + 1 * 2 = 5 marks**]

```
Answer:
double EvalPolyDerivative(
    double a[],                 // Array of coefficients
    unsigned int n,             // Degree of polynomial
    double x) {                 // Value of unknown for evaluation
    unsigned int i;             // Index
    double b[MAX_POLY_SIZE];    // Array of coefficients of
                                // derivative polynomial
    for(i = 0; i < n; ++i) {
               -----
        b[i] = a[i+1]*(i+1);    // Perform derivative
               -------------
    }
    return EvalPoly(b, n-1, x); // Evaluate derivative polynomial
           --------    ---
}
```

Note for Marking: *Array indexing by pointer (like* `*(a+i+1)` *for* `a[i+1]`*), re-ordering of operands (like* `n > i` *for* `i < n`*) are acceptable if correct.*

Roll No: _____

3. It is required to fill up a 2D array `A` with a zigzag pattern that starts from the left-top corner with `00`. The numbers are filled up sequentially along the anti-diagonals with the direction alternating between *up* (from south-west (SW) to north-east (NE): `01 --> 02`) and *down* (from NE to SW: `03 --> 04 --> 05`) as follows:

```
Answer:

00     02     03     09     10     XX     21
                                   (20)   --
01     04     08     11     XX
                           (19)
05     07     12     XX
                    (18)
06     13     XX
              (17)
14     XX
      (16)
15
--
```

ROUGH WORK

Use this space for your rough work, if any.
This part will not be evaluated.

(a) Study the above pattern carefully and fill up the dashed values. You need to compute the values marked as `XX`, but you do not need to write them. There is no separate credit for writing `XX`'s. [**1 * 2 = 2 marks**]

(b) Fill up the missing codes below to generate the above pattern in array `A` in the order mentioned above: [**1 * 8 = 8 marks**]

```
Answer:
    int A[20][20];  // The 2D Array
    int n = 10;     // Number of anti-diagonals to fill
    int val = 0;    // Next value to fill up. Starts with 0
    int dir = 1;    // Direction of fill up. dir = 1 means direction is downward
                    // from north-east (NE) to south-west (SW), 0 otherwise
    int i, j;       // Indices
    int k;          // Anti-Diagonal control index

    for(k = 0; k < n; ++k) {        // Control the diagonal to fill
               -----
        for(i = 0; i <= k; ++i) {   // One index of A
                  ------
            j = k - i;              // Other index of A
                -----
            if (dir)                // Decide direction and choose fill location
                A[i][j] = val++;    // Downward fill. From north-east to south-west
                  -  -
            else
                A[j][i] = val++;    // Upward fill. From south-west to north-east
                  -  -
        }
        dir = 1 - dir;              // Change direction
              --------
    }
```

Note for Marking: *Equivalent expressions like* `(dir)?0:1` *that takes* `0` *to* `1` *and* `1` *to* `0` *are acceptable in place of* `dir = 1 - dir`. *Re-ordering of operands (like* `k >= i` *for* `i <= k`*) are acceptable if correct.*

4. The following questions use a linked list consisting of nodes as:

```
typedef struct Node_ {
    char data;
    struct Node_ *next;
} Node;
```

Variable `Node *head;` holds the header of the list.

(a) Compute `FindSpecialChar(str)` for `str = "merge", "memory",` and `"recursion"` to get an idea for what the function `FindSpecialChar(str)` does. State the output (return value) of `FindSpecialChar(str)` in words for a given `str`. [**1 + 1 + 1 + 2 = 5 marks**]

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

char FindSpecialChar(char *str) {
    Node *head = (Node *)malloc(sizeof(Node));
    Node *p = head;
    Node *q = 0, *r = 0;
    int i, n = strlen(str);
    // Form a list with the characters of the string
    for(i = 0; i < n - 1; ++i) {
        p->data = str[i];
        p->next = (Node *)malloc(sizeof(Node));
        p = p->next;
        p->next = 0;
    }
    p->data = str[n - 1];
    // Find a special character
    q = p = head;
    while (p) {
        p = p->next;
        if (p) {
            p = p->next;
            q = q->next;
        }
    }
    return q->data;
}
```

ROUGH WORK

Use this space for your rough work, if any.
This part will not be evaluated.

```
Answer:
For input "merge", the output is:     r
                                      -
For input "memory", the output is:    o
                                      -
For input "recursion", the output is: r
                                      -
Given str, the output is: the middle character in str
                          ---------------------------
```

Note for Marking: *Separate explanations for strings of odd and even lengths are acceptable as long as* middle character *is mentioned.*

(b) You need to write `Node *ReverseList(Node *p)` to reverse the nodes in a given linked list. That is,

`head = ReverseList(head);`

will take a list held by `head`, reverse it and put back to `head`.
Fill up the missing codes in `Node *ReverseList(Node *p)` below. [**1 * 5 = 5 marks**]

Answer:

```
Node *ReverseList(Node *p) {
    if (!p) return 0;

    if (p->next) {
        Node *q = ReverseList(p->next);
                              ----------
        p->next->next = p;
                        ---
        p->next = 0;
                  ---
        return q;
              ---
    }
    return p;
          ---
}
```

Note for Marking: `NULL` *in place of* `0` *is acceptable. De-referencing expressions like* `(*p).next` *in place of* `p->next` *are acceptable if correct.*

5. The function `int *Place(int *left, int *right);` takes two pointers `left` and `right` to two locations in an array, say `A`, of `int` such that the index value `lIndex` of the element pointed to by `left` is less than the index value `rIndex` of the element pointed to by `right`. Clearly, `left = &A[lIndex]`, `right = &A[rIndex]` and `lIndex < rIndex`. Assume that the elements in `A[]` are all distinct.

   `int *Place(int *left, int *right)` places the element `num = A[lIndex]` pointed to by `left` at a position `pIndex` (`lIndex <= pIndex <= rIndex`) such that all elements between `A[lIndex]` and `A[pIndex-1]` are less than `A[pIndex]` and all elements between `A[pIndex+1]` and `A[rIndex]` are greater than `A[pIndex]`. The function `int *Place(int *left, int *right)` does not disturb any element outside the range of `A[lIndex] .. A[rIndex]` and returns `&A[pIndex]` after the placement.

   For example, for `A[] =`

0	1	2	3	4	5	6	7	8	9	10	11
5	4	10	7	3	9	12	11	6	8	13	14

   `lIndex = 2` and `rIndex = 8` (`left = &A[2]` and `right = &A[8]`), `int *Place(int *left, int *right)` places `A[2] = 10` at `&A[6]`, sets `A[]` as

0	1	2	3	4	5	6	7	8	9	10	11
5	4	6	7	3	9	10	11	12	8	13	14

   computes `pIndex` as `6` and returns `&A[6]`.

   (a) Express the following in terms of `left`, `right`, and `A` **only**. [**1 * 4 = 4 marks**]

```
Answer:

lIndex    = left - &A[0]
            ------------
rIndex    = right - &A[0]
            --------------
A[lIndex] = *left
            -----
A[rIndex] = *right
            ------
```

   Note for Marking: *Equivalent address expressions like* `left - A` *in place of* `left - &A[0]` *are acceptable if correct.*

   (b) Given the bounds `left` and `right`, write an expression of them for the number of elements of `A` that lie between these bounds (both ends included). For example, for the above example, if (`left = &A[2]` and `right = &A[8]`), the number of elements between them (counting both ends) is `7`. [**2 marks**]

```
Answer:

Number of elements = right - left + 1
                     -----------------
```

(c) Fill up the missing codes in the function below where `void Swap(int *p, int *q);` is a function that takes two pointers to `int` variables and swaps their values. [**2 + 2 + 1 + 1 + 1 = 7 marks**]

Answer:

```
void Swap(int *p, int *q);

int *Place(int *left, int *right)
{
    int num = *left; // Element to place
    int *l = left;   // Left scanning pointer
    int *r = right;  // Right scanning pointer

    // Repeat from two ends till the ends meet
    while (l < r){

        // Scan from left to find the first element greater than num
        while(*l <= num && l <= right) ++l;
              ---------
        // Scan from right to find the first element smaller than num
        while(*r > num) --r;
              --------
        if (l < r) {
            // Interchange the two wrongly placed elements
            Swap(l, r);
                 --  --
            --r;
        }
    }

    // Final interchange
    Swap(r, left);

    // Return position
    return r;
           --
}
```

Note for Marking: *Re-ordering of operands (like* `num >= *l` *for* `*l <= num`*) are acceptable if correct.*

(d) What property should be satisfied by the elements of `A` so that every call to `int *Place(int *left, int *right)` would have no effect on `A` and would simply return `left`? Write this property in words. [**2 marks**]

Answer:

`A` is sorted in increasing order.

6. Answer the following questions:

   (a) You are given two text files with the following contents:

File Name	File Contents	Remarks
INPUT.txt	29 48	Contains one line
OUTPUT.txt	97 6 124	Contains two lines Second line is blank

   These files are used for IO with the following program.

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

int main() {
    int i = 5, j = 17;

    FILE *ifp = 0, *ofp = 0;

    // Open and read file
    if (ifp = fopen("INPUT.txt", "r")) {
        printf("Reading file\n");
        fscanf(ifp, "%d %d", &i, &j);
    }
    else { printf("Cannot read INPUT.txt"); exit(0); }

    fclose(ifp);

    // Open and write file
    if (ofp = fopen("OUTPUT.txt", "a")) {
        printf("Writing file\n");
        fprintf(ofp, "%d", i);
    }
    else { printf("Cannot write to OUTPUT.txt"); exit(0); }

    // Code Point A

    fclose(ofp);

    // Open and write file
    if (ofp = fopen("OUTPUT.txt", "w")) {
        printf("Writing file\n");
        fprintf(ofp, "%d\n%d", j, i);
    }
    else { printf("Cannot write to OUTPUT.txt"); exit(0); }

    // Code Point B

    fclose(ofp);

    return 0;
}
```

Roll No: _____

i. Write the contents of file `OUTPUT.txt` at the marked points. [**1 * 4 = 4 marks**]

```
Answer:

@ Code Point A

97 6 124
--------
29
--

@ Code Point B

48
--
29
--
```

ii. What will be the output to the console if the file `INPUT.txt` is missing? [**1 mark**]

```
Answer:

Cannot read INPUT.txt
---------------------
```

(b) Consider the following program that accepts arguments from the command line:

```
Answer:

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

// main with Command line parameters
void main(int argc, char *argv[]) {
          ---       ------
    int i;
    double d;
    char s[20];

    // Print the number of arguments
    printf("argc = %d\n", argc);

    // Print the 1st argument
    printf("argv[0] = %s\n", argv[0]);

    // Set the 2nd argument (int) to i
    sscanf(argv[1], "%d", &i);
    -------------------------
}
```

Note for Marking: *Solution using* `atoi`*, that is,* `i = atoi(argv[1]);` *in place of* `sscanf` *is also acceptable. If* `sscanf` *is used, award 1 mark for* `argv[1]`*, 0.5 mark for* `%d`*, and 0.5 mark for* `&i`*. If* `atoi` *is used, award 1 mark for* `argv[1]`*, and 1 mark for* `i =` *. No other solution is acceptable.*

This program is compiled into an executable file `"C:\Programs\CommandLine.exe"` and is executed from `C:\Programs>` folder as

```
CommandLine.exe 27
```

i. Fill up the missing codes in the program. [**0.5 * 2 + 2 = 3 marks**]

ii. Write the output of the program. [**1 * 2 = 2 marks**]

```
Answer:
argc = 2
       -
argv[0] = C:\Programs\CommandLine.exe
          --------------------------
```

Note for Marking: `argv[0]` *is acceptable as long as* `CommandLine.exe` *is mentioned. Writing the path is optional.*