Indian Institute of Technology Kharagpur

EXAMINATION ANSWERSCRIPT

Stamp/Signature of the Invigilator

Spring Semester 2025-26 – End Sem Exam	**SEMESTER (Spring)**

Roll Number											**Section**		**Name**	

Subject Number	**C**	**S**	**1**	**0**	**0**	**0**	**3**	**Subject Name**	**Programming and Data Structures**

Department/Centre/School		**Additional Sheets**	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.
2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.
3. Loose papers, class notes, books or any such materials must not be in your possession; even if they are irrelevant to the subject you are taking examination.
4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.
5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, the exchange of these items or any other papers (including question papers) is not permitted.
6. Write on both sides of the answer-script and do not tear off any page. **Use designated places of the answer-script for rough work**. Report to the invigilator if the answer-script has torn or distorted page(s).
7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.
8. You may leave the Examination Hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.
9. Do not leave the Examination Hall without submitting your answer-script to the invigilator. **In any case, you are not allowed to take away the answer-script with you**. After the completion of the examination, do not leave your seat until the invigilators collect all the answer-scripts.
10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging information with others or any such attempt will be treated as '**unfair means**'. Don't adopt unfair means and also don't indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be Filled by the Examiner

Question Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Marks Obtained											

Marks Obtained (in words)	**Signature of the Examiner**	**Signature of the Scrutineer**

IMPORTANT INSTRUCTIONS

1. **No clarification will be provided during the exam. If you make any additional assumptions in a question, write it clearly beside the question.**

2. **Write your answers only within the boxes/blanks provided. Answers written anywhere else will not be evaluated.**

3. **You can do rough work in any blank space outside the boxes shown in the questions. Do not do any rough work inside the boxes.**

4. **Write final answers with pen only. No answers written in pencil will be evaluated.**

5. **Do NOT declare any new variables in any of the questions.**

6. **Do NOT write more than one statement (simple or compound) in a single line.**

YOU CAN USE THE REST OF THIS PAGE BELOW FOR ROUGH WORK ONLY

Q1. Answer the following questions as directed. (12 × 1 = 12)

(a) What is the hexadecimal equivalent of the decimal number **101.125**?

ANS: 65.2

(b) What is the 8-bit 2's complement representation of the result obtained by adding the 2's complement signed integers **11001001** and **10111101**?

ANS: 10000110

(c) What will be printed by the following program?

```
int n = 10;
void fgh ( int r ) {
    printf("%d, %d\n", n, r);
}
int main ( ) {
    int n = 20;
    fgh(n);
}
```

ANS: 10, 20

(d) What will be the value returned by the call **doit(4)**?

```
unsigned int doit ( unsigned int n ) {
    unsigned int i, s, t;
    s = 0; t = 1;
    for (i = 0; i < n; ++i) {
        s = s + i + n;
        t = t * 2;
    }
    return (s * t);
}
```

ANS: 352

(e) Consider the function below, where A is a null-terminated string.

```
void recFunc ( char A[ ] ) {
    int t;
    t = strlen(A);
    if (t == 0) return;
    else if (t % 2 == 0)
            recFunc(&A[t/2]);
    else recFunc(&A[1]);
}
```

If the function is called with the string "programming" as parameter, total how many times will **recFunc() be** called (include the initial call of **recFunc("programming")** in the count)?

ANS: 7

(f) Consider the structure below, where the rank field stores the class rank of a student. What should be filled in the blank in the **printf** statement so that it prints the class rank of the student A? **Your answer cannot contain the letters A and C, use B to access the ranks of A and C.**

```
struct student {
    int rank;    struct student *ptr;
};
int main( ) {
    struct student A, B, C;
    A.rank = 1; B.rank = 5; C.rank = 10;
    A.ptr = &B; B.ptr = &C; C.ptr = &A;
    printf("%d", _________ );
}
```

ANS: B.ptr->ptr->rank

Other variations are possible and have been given full credit if correct

(g) Consider the following sequence of **push** and **pop** operations on an initially empty stack S of integers. Each operation returns the new stack after the change.

```
S = push(S, 1);
S = pop(S);
S = push(S, 2);
S = push(S, 3);
S = pop(S);
S = push(S, 4);
S = pop(S);
S = pop(S);
```

Write the integers in the order they are popped.

ANS: 1 3 4 2

(h) Consider the following function.

```
void func1 (int *p) {
    int *a;
    a = (int *) malloc (sizeof(int));
    *a = 10;   *a = (*a)*5;
    p = a;
}
```

Suppose that in the **main()** function, an int type variable *X* has a value 20. If the function is called two times consecutively as **func1(&X)** from **main()**, what will be the value of *X* after the second function call returns?

ANS: 20

(i) What will be printed by the following program?

```
int main( ) {
    int arr[ ] = {10, 20, 30, 40, 50};
    int *ptr = arr;
    printf("%d, ", *ptr++);
    printf("%d, ", (*ptr)++);
    printf("%d, ", *++ptr);
    printf("%d", ++*ptr);
    return 0;
}
```

ANS: 10, 20, 30, 31

(j) We want to sort the array

A = {6, 3, 9, 1, 5, 8, 7, 2}

in ascending order. We use an algorithm that, for $p = 0, 1, 2, \ldots$ (in that sequence), finds the index q of the smallest element in A[$p \ldots n-1$], and then swaps A[p] with A[q], where n is the number of elements in the array. Show the contents of the array just **after** the first four iterations (for $p = 0, 1, 2, 3$) are complete.

ANS: {1, 2, 3, 5, 6, 8, 7, 9}

(k) What will be printed by the following program?

```
int *what ( int *p ) {
    ++p;  *p = 10;  ++p;  return p;
}
int main ( ) {
    int A[ ] = {1, 2, 3, 4, 5}, *p;
    p = what(A);  printf("%d\n", p[0]);
}
```

ANS: 3

(l) Consider the program fragment below. Write a single variable declaration statement for q and r, such that there will be no compilation error/warning for q and r.

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

ANS: int *q, **r;

GRADING GUIDELINE:

For all parts EXCEPT (c), (i), and (l): 1 mark if correct, 0 otherwise (binary)

For part (c): 0.5 each for 10 and 20

For part (i): 0.5 if at least 2 correct, 1 if all correct. Order of values must be correct

For part (l): 0.5 each for declarations of q and r. Ok if you also declare p, but no marks for that.

Q2. The following method is used to check if a positive integer is divisible by 7 or not. Take the two left-most digits of the integer, multiply the left-most digit by 3 and add it to the second digit. Replace these two digits in the number with the result, dropping any leading 0's. Then we can keep repeating this, always dealing with only the two left-most digits, until we end up with a single digit number which is either divisible by 7 (if digit is 0 or 7) or not.

For example, if the input is 249, the sequence of numbers obtained is

249 → 109 (as 3*2 + 4 = 10) → 39 (as 3*1 + 0 = 3) → 18 (as 3 * 3 + 9 = 18) → 11 (as 3 * 1 + 8 = 11) → 4 (as 3 * 1 + 1 = 4), which is not divisible by 7. So 249 is not divisible by 7.

Similarly, if the input is 49, the sequence will be 49 → 21 → 7 (divisible by 7). So 49 is divisible by 7.

The following function takes a positive integer *n* and returns 1 if it is divisible by 7, 0 otherwise, using the above method to check. It is assumed that the number of digits in the integer is not greater than 8. The function first separates and stores the digits of the number in an array, and then works on the digits in the array. Fill in the blanks so that the function does this correctly. Follow the comments shown. (8)

```
int divBy7(int n) {
        int temp = n, i, A[8], numdigit, leftIndex, nextLeftIndex, leftmost, nextLeftmost, val;
        /* Separate out the digits of the number and store in array A, with A[7] holding
           the rightmost digit.  The variable numdigit should hold the number of digits.
           Ex. for 249, A[5] = 2, A[6] = 4, A[7] = 9, and numdigit will be 3 */

        for (numdigit = 0, i = 7; temp > 0; numdigit++, i--, temp = temp/10)          [A]
                A[i] = temp %10;
        while (numdigit > 1) {
                /* get leftmost and 2nd leftmost digits, then compute the value to replace them with */

                leftIndex = 8 – numdigit;                                            [B]
                leftmost = A[leftIndex];

                nextLeftIndex = 9 – numdigit ;        OR    leftIndex + 1          [C]
                nextLeftmost = A[nextLeftIndex];
                val =3*leftmost + nextLeftmost;
                /* replace last two digits in number in array with val */

                A[nextLeftIndex] = val % 10 ;                                        [D]

                if (val > 9) A[leftIndex] = (val/10) ;                               [E]

                else numdigit--;                                                     [F]
        }
        /* only a single digit left in the number in the array. Check directly now */

        if (A[7] % 7 == 0) return 1;   OR    A[7] == 7   OR   A[7] == 7 || A[7] == 0        [G]
                                       Also, A[7] can be replaced with val or A[nextleftIndex]
        else return 0;
}
```

GRADING GUIDELINES:

For all blanks except [A]: 1 mark if fully correct, 0 otherwise (binary)
For blank [A]: 1 if any two of the three things to be updated are correct, 2 if all three correct.
Other variations seen have been given full/partial credit depending on correctness.

If you have written more than one statement in one line (violating instruction 6), partial credit is given if otherwise correct. Some of you have used comma operator to bypass it, given full credit if correct, though it is not why comma operator is there.

Q3. Consider an array A of n distinct, positive integers that satisfies the following inequalities:

$$A[0] < A[1] < A[2] < \ldots < A[m] > A[m+1] > A[m+2] > \ldots > A[n-1]$$

for some index m in the range $0 < m < n-1$. Let us call such an array a ***hill-valued*** array. The sequence A[0], A[1], A[2], ... A[m] is called the ***ascending part*** of the hill, and the remaining part the ***descending part***. Thus, A[m] is the maximum element in the array. Note that given the range of m specified, both the ascending part and the descending part have at least one element. Complete the following function that takes as input an array A of n elements that satisfies the above inequality, and returns the maximum value in the array using ideas from binary search. (6)

```
int findMax(int A[ ], int n)
{
        int i, L = 0, R = n – 1, M, found = 0;

        while (found == 0) {  /* Loop as long as the maximum is not located */

                /* Compute the middle index M */
                M = (L + R) / 2;

                if ((A[M –1] < A[M]) && (A[M + 1]< A[M])) {
                        /* the maximum is found, */

                                found = 1 ;  OR  found = A[M]   OR   break

                } else if ((A[M – 1]<A[M]) && (A[M] <A[M + 1]) ) {     OR  Any ONE of the two  conditions  ONLY
                                                                           as there is an else-if
                        /* the maximum is in the ascending part */

                        L = M ;        OR   L = M + 1

                } else {  /* the maximum is in the descending part */

                        R = M ;       OR     R = M – 1
                }
        }
        return A[M];
}
```

GRADING GUIDELINES:

For all blanks: 1 mark if fully correct, 0 otherwise (binary)

Ok if you have interchanged the check and setting of L/R for ascending and descending part in the if-else as long as correct.

A few other variations are possible which have been given full credit if correct.

Q4. You are given a null-terminated string containing only the following characters: '0', '1', and '?' (also called the ***wildcard character***). The objective is to find all possible strings that can be formed by replacing each wildcard character present by either a '0' or a '1', while keeping the other '0's and '1's the same. For example, for the input string "0?1", the possible strings are 001 and 011. Similarly, for the input string "100?10?1", the possible strings are 10001001 10001011 10011001 and 10011011.

Complete the **recursive** function given below that takes a null-terminated string as one of the parameters, and prints all possible strings that can be formed following the above description (one in each line, in any order). The broad logic to print all possible strings is as follows: (i) scan the characters in the input string from left to right, (ii) if a '?' character is hit, first replace '?' by 0 and proceed to the rest of the string from left to right recursively, then do the same set of steps by replacing the '?' with 1. The function is called from **main()** as **printAllCombination(S, 0)**, where S is a null-terminated string as above. (6)

```
void printAllCombinations(char pattern[ ], int i) {
        int k;

        if (pattern[i] == '\0' ) {
            /* base case, no characters left */
            printf("%s\n", pattern);
            return;
        }
        if (pattern[i] == '?') {

            for (k = 0; k < 2 ; k++) {

                pattern[i] = k + '0';

                printAllCombinations(pattern, i + 1) ;

                pattern[i] = '?';
            }
            return;
        }
        printAllCombinations(pattern, i + 1) ;
}
```

GRADING GUIDELINES:

For all blanks: 1 mark if fully correct, 0 otherwise (binary)

Note that for the first recursive call, i++ or ++i in place of i+1 is wrong as i should not change as the same i has to be used in the next line to reset to '?'. Given 0.

For the second recursive call, i++ is wrong, ++i is correct (as i is no longer needed after this). Given marks for both if call is otherwise correct.

A few other variations seen have been given full credit if correct.

Q5. Consider a function **merge2D( )** that takes an $m \times n$ 2-d integer array A (containing positive integers, all with value less than 10000) as parameter. Each row of the 2-d array A is a sorted (in ascending order) 1-d array of size n (i.e., each row is already sorted, you do not have to sort the rows). It is also given that each row of A contains only distinct, positive integers. However, the same integer may be repeated in different rows. The function merges the m sorted 1-d arrays (corresponding to the m rows of the 2-d matrix A) in a single sorted list, and stores this merged sorted list in a 1-d array B (also passed as parameter to **merge2D**). During the merging step, all repeated elements of A are removed while storing in B, so B contains only distinct integers Finally, the function returns (as return value) the size of the array B. For example, if

$$A = \begin{bmatrix} 1 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 \\ 3 & 5 & 6 & 7 \\ 2 & 3 & 6 & 9 \end{bmatrix}$$ then the resultant 1-d array B = [1 2 3 4 5 6 7 9]

and 8 (the number of integers in B) is returned as return value. Note that each of the elements 1, 2, 3, 4, 5, 6 are repeated across different rows of A and repetitions are removed while storing to B. Complete the **merge2D** function given below so that it correctly implements the above. Assume that B has sufficient space. (8)

```
int merge2D ( int B[ ], int A[ ][100], int m, int n ) {
        int index[100], i, k, min;

        /* index[i] will store the index of the column with the minimum value
           in the i-th row during merge. Initially, it is 0 for all rows. */
        for (i = 0; i < m ; ++i)  index[i] = 0 ;

        k = 0 ;  /* index for iterating over array B */

        while (1) {
                /* find the minimum element among the minimum of each of the rows */
                min = 999999;
                for ( i = 0; i < m; ++i ) {

                        if ( ((index[i] < n)) && (A[i][index[i]] < min) )

                                min = A[i][index[i]];
                }
                if (min == 999999) return k ;      OR   break
                /* Store the minimum in B */

                B[k++] = min ;                  For this line only, ok if they use 2 statements  B[k] = min; k++;
                /* increment the index for those rows where the value at current
                   index is equal to min value */
                for ( i = 0; i < m; ++i ) {

                        if ( (index[i] < n ) && (A[i][index[i]] == min ) )

                                ++index[i] ;
                }
        }
        return k;
}
```

GRADING GUIDELINES:

For all blanks: 1 mark if fully correct, 0 otherwise (binary)

Ok if you have interchanged the two conditions inside a single if's && check, even though it is not fully correct.

A few other variations seen has been given full credit if correct.

Q6. Consider a set of points in a 2-d plane. Each point is defined by the structure

struct point { int x; int y; };

For simplicity, assume that any two points differ in at least one of the two (*x* and *y*) coordinates. A point P **dominates** another point Q if **both** the *x* and *y*-coordinates of P are greater than or equal to the corresponding coordinate of Q. Given a set of points, we wish to find out if there is any point that dominates all other points.

The following function takes as parameter an array of **struct point** type structures and the number of such structures in the array (assume that there are at least two points in the array). If there is any point that dominates all other points, it returns the pointer to the structure holding that point in the array, else it returns NULL. To do this, the function sorts the points in non-increasing order of their *x*-coordinates. If two points have the same *x*-coordinate, they are ordered in non-increasing order of their *y*-coordinates among themselves (i.e., the higher *y*-coordinate coming first. Note that by our assumption, their *y*-coordinates must differ as they have the same *x*-coordinate). After we sort, we check the first point in the list if it dominates all other points. Note that no other point can dominate all other points.

Fill in the blanks for the function to do this. (6)

```
struct point *FindPoint(struct point A[ ], int n)
{
        int i,  j;    struct point temp;
        /* sort as described using insertion sort */
        for (i = 1; i < n; i++) {

                temp = A[i];
                for (j = i - 1; j >= 0; j--) {

                        if ((temp.x > A[j].x) || ((temp.x == A[j].x) && (temp.y > A[j].y))) {        [A]
                                                        OR      (A[i].x > A[j].x) || ……   (first one i, then j)
                                                        OR      (A[ j ].x < A[j+1].x) || ……..  (both j, no i)
                                A[j+1] = A[j];
                                A[j] = temp;
                        }
                        else break;
                }
        }
        /* check to see if A[0] dominates all other points or not */

        for (i = 1; i < n; i++)   if (A[0].y < A[i].y) return NULL;    Ok if they also check for x with an OR condition
                                                                       if correct as A[0].x is >= A[i].x for all I at this point

        return (&A[0]);   OR   A
}
```

GRADING GUIDELINES:

For all blanks except [A]: 1 mark if fully correct, 0 otherwise (binary)
For blank [A]: 1 for each part of the OR condition check (the two parts separated by ||).

Q7. Consider a singly linked list, each of whose nodes are defined by the following structure:

struct node { int data; struct node *next; };

Consider the function **removeDupl()** below that takes as parameter the head pointer of a singly linked list, and returns the head pointer of a modified singly linked list that is obtained from the input list by removing all duplicate integers from it. More precisely, if an integer occurs more than once, only the node with the first (counting from head) occurrence is kept, the other ones are deleted. No new node should be dynamically allocated. The logic to remove duplicates is simple. For each element from the beginning, the function traverses the rest of the list till the end, deleting any duplicate of that element found. Fill in the blanks so that the function does this. For simplicity, the code below ignores free'ing memory (so do not use any free() function call). (7)

```
struct node *removeDupl(struct node *head) {

        struct node *check, *prev, *curr, *temp;     int val;
        /* check elements one by one for duplicates */
        check = head;
        while (check  != NULL) {
                val = check->data;

                prev = check;

                curr = check->next;        OR      prev->next
                while (curr != NULL) {

                        if (curr->data == val) {
                                /* duplicate found. Delete and move on. */

                                prev->next = curr->next ;
                                curr = curr->next;

                                continue ;
                        }
                        prev = curr;     curr = curr->next;
                }
                check = check->next ;
        }
        return head ;
}
```

GRADING GUIDELINES:

For all blanks: 1 mark if fully correct, 0 otherwise (binary).

Small variations seen in some blanks have been given full credit if correct.

Q8. Consider the following typedef to define a **STACK** datatype to hold integers.

typedef struct { int data[100]; int top; } STACK;

The STACK datatype supports the following methods. For simplicity, assume that the stack is never full.

void init(STACK *S) : initializes the stack pointed to by S before use, always called once at the beginning.
int top(STACK *S) : returns (but does not remove) the integer at the top of stack pointed to by S.
int pop(STACK *S) : removes (pops) the top element from stack pointed to by S and returns it.
void push(STACK *S, int k) : pushes integer *k* in the stack pointed to by S.
int isEmpty(STACK *S): returns 1 if stack pointed to by S is empty, 0 otherwise.

The following function **sortStack()** takes as parameter the pointer *IP* to a stack *S*, and returns the pointer to a **new** stack that contains all elements of *S* in descending order from the top (i.e. if we keep popping elements from the new stack returned, the first element popped will be the maximum element in *S*, the second element popped will be the second maximum element in *S* and so on, with the last element popped being the minimum element in *S*). For example, if *S* = [30, 20, 2, 1, 38, –1] (with 30 being the top element and –1 the bottom element), then the stack returned will be [38, 30, 20, 2, 1, –1]. The original stack *S* should be left unchanged.

The sorting is done in the following manner. The stack pointed to by *IP* is first copied to a temporary stack *tempstack* (since the original stack cannot be changed) and the memory for the new stack is dynamically allocated. The algorithm then repeatedly does the following until *tempstack* is empty: (i) pop the top element from *tempstack*, call it *iptop*, (ii) for all elements in the new stack that are greater than *iptop*, pop them from new stack and push them in *tempstack*, (iii) finally, push *iptop* in new stack and repeat the process.

Complete the function by filling in the blanks properly. **You can access the stacks only through the functions given**, do not access the array directly. (7)

```
STACK *sortStack(STACK *IP) {
        STACK *new, tempstack;      int iptop, temp;

        tempstack = (*IP);
        /* Create and initialize the new stack */

        new = (STACK *) malloc (sizeof(STACK));                    [A]
        init(new);
        while ( isEmpty(&tempstack) == 0) {

                iptop = pop(&tempstack);                           [B]
                while (isEmpty(new) == 0) {
                        /* if top of new stack is greater than iptop */

                        if (top(new) > iptop) {
                                temp = pop(new);

                                push(&tempstack, temp );           [C]
                        }
                        else break;
                }

                push(new, iptop) ;
        }
        return new;
}
```

GRADING GUIDELINES:

For all blanks: 1 mark if fully correct, 0 otherwise (see below for exception)

For blank [A], give 0.5 if right hand side is fully correct, but not assigned to new correctly

For blanks [B] and [C], deduct 0.5 marks if & not given in call (deduct for each one separately if not given)