INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (Mid Semester / ~~End Semester~~)	SEMESTER (~~Autumn~~ / Spring)

Roll Number										Section		Name	
Subject Number	C	S	1	0	0	0	3	Subject Name					*Programming and Data Structures*
Department / Center of the student												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, exchange of these items of 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 last page(s) 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 the 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**'. Do not adopt unfair means and do not indulge in unseemly behavior.

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

Signature of the Student

To be filled in 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

CS10003 / CS10001 PROGRAMMING AND DATA STRUCTURES
SPRING 2024 – 2025
MID-SEMESTER EXAMINATION
9AM – 11AM, 19-FEBRUARY-2025
MAXIMUM MARKS: 100

Instructions to students

- Write your answers in the question paper itself.
- Answer **all** questions.
- All programs must be written in the **C programming language**.
- Write in the blank / empty spaces provided in the questions. Do rough work in the designated places. For additional rough work, you may ask for supplementary sheets from the invigilators. The answers to all the questions must be written in this question paper only.
- Not all blanks carry equal marks. Evaluation will depend on your overall performance.
- Do not change the intended meaning (as described in the text) of the variables and functions in the questions.
- Unless otherwise specified, you are not allowed to introduce extra variables.
- Do not write anything on this page. Questions start from the next page (Page 3).

1. **(a)** Fill in the blanks for a switch statement equivalent to the following nested if-else statement. You may assume that the user enters a non-negative integer as input. **[8]**

Nested if-else statement

```
int i;
printf("Enter a non-negative integral value for i: ");
scanf("%d", &i);

if ( (i >= 0) && (i < 4) )
  printf("A\n");
else if ( (i > 4) || (i % 4) )
  printf("B\n");
else if (i = 5)
  printf("C\n");
else
  printf("D\n");
```

Equivalent switch statement

```
int i;
printf("Enter a non-negative integral value for i: ");
scanf("%d", &i);

switch (i) {

    case ________0________ :

    case ________1________ :

    case ________2________ :

    case ________3________ :

         printf("________A________\n");

         ________break________ ;

    case ________4________ :

         printf("________C________\n");

         ____i = 5; break____ ;

    default: printf("________B________\n");
}
```

The statement i = 5 preserves the side effects. Students are not required to write this.

(b) Fill in the blanks of the program on the next page, where the user enters an amount of money. Your task is to output the minimum numbers of notes required to realize this amount using the denominations {500, 100, 50}, or say that this amount cannot be realized using these denominations. For example, the minimum numbers of notes with which 3750 INR can be realized are seven 500-INR notes, two 100-INR notes, and one 50-INR note. On the other hand, 3740 INR cannot be realized using the notes of the given denominations. Assume that the user enters a positive amount. **[12]**

```
int main()
{
  int n, copy, a, b, c;
  /* n will store the amount input by the user, and copy is a copy of n.
     The other variables are for storing the numbers of notes for the 3 denominations */

  printf("Enter a positive amount in INR: ");

  /* Assume that the user enters a positive value for n */

  scanf("%_____d_____", _____&n_____ );

  copy = n;

  if ( _____n >= 500_____ ) {   /* Use the variable n in the condition */

    a = _____n / 500_____ ;

    n = n - _____a * 500_____ ;

  } else {

    _____a = 0_____ ;
  }

  if ( _____n >= 100_____ ) {

    b = _____n / 100_____ ;

    n = n - _____b * 100_____ ;

  } else {

    _____b = 0_____ ;
  }

  if ( _____n >= 50_____ ) {

    c = _____n / 50_____ ;

    n = n - _____c * 50_____ ;

  } else {

    _____c = 0_____ ;
  }

  if ( _____n > 0_____ )

    printf("%d cannot be realized with the denominations 500, 100, 50\n", _____copy_____ );

  else

    printf("%d INR is realized by %d 500-INR notes, %d 100-INR notes, and %d 50-INR notes",

          _____copy_____ , _____a_____ , _____b_____ , _____c_____ );

          /* In this printf statement, some of the denominations may have 0 notes being used */
}
```

2. **(a)** The following program computes the reverse of a user-specified positive integer N, after ensuring that N has at least four digits. It begins by taking N as input, validates that N is positive (terminating if negative), and calculates its digit count to verify that it meets the minimum requirement of four digits. If the length condition is not satisfied, the program terminates; otherwise, it proceeds to reverse the digits of N using mathematical operations like remainder and division. The reversed number is then printed. The `long int` data type is used to handle N and its reverse. Fill in the blanks below so that the program works as intended. **[10]**

```
#include <stdio.h>

int main() {
    long int N, reverseN = 0;
    int lengthN = 0;

    printf("Enter a positive integer with more than 4 digits: ");

    scanf(" ____%ld____ ", &N);

    if ( ____N <= 0____ ) {
        printf("The number must be positive.\n");
        return 0;
    }

    long int temp = N;

    while ( ____temp > 0____ ) {
        ____temp /= 10____ ;
        ____lengthN++____ ;
    }

    printf("Length of N is %d\n", lengthN );

    if ( ____lengthN < 4____ ) {
        printf("The number must have at least 4 digits.\n");
        return 0;
    }
    while ( ____N > 0____ ) {

        int digit = ____N % 10____ ;

        reverseN = ____reverseN * 10 + digit____ ;

        ____N /= 10____ ;
    }

    printf("The reversed number is %ld\n", reverseN );

    return 0;
}
```

(b) The following program is meant for finding and printing all perfect numbers between two integers entered by the user. A **perfect number** is one that is equal to the sum of its proper divisors (excluding itself). For example, 6 is a perfect number because 1 + 2 + 3 = 6, and 28 is a perfect number too because 1 + 2 + 4 + 7 + 14 = 28. The program prompts the user to enter a positive lower limit and a positive upper limit, validates the input to ensure the limits are correctly defined, and then uses two for loops to compute the sum of proper divisors for each number in the range. The outer loop goes through each number in the given range, and the inner loop performs the necessary calculations to determine the relevant properties of the current number. If a particular condition is satisfied based on these calculations, the number is identified as a perfect number, and is printed. If no perfect numbers are found in the given range, the program prints an appropriate message. Fill in the blanks in the code below so that the program works as intended. **[10]**

```
#include <stdio.h>

int main() {
    int low, high;

    printf("Enter the lower limit: ");

    scanf("_____%d_____", &low);

    printf("Enter the upper limit: ");

    scanf("_____%d_____", &high);

    if ( ____(low <= 0) || (high <= 0) || (low > high)____ ) {

        printf("Invalid limit(s).\n");
        return 1;
    }

    printf("Perfect numbers between %d and %d are:\n", low, high);

    int found = 0;
    int num, i;

    for ( num = ____low____ ; ____num <= high____ ; num++ ) {

        int sum = 0;

        for ( i = 1 ; ____i <= num / 2____ ; i++ ) {

            if ( ____num % i == 0____ ) {

                ____sum += i____ ;
            }
        }

        if ( ____sum == num____ ) {

            printf("%d\n", num);
            found = 1;
        }
    }

    if ( ____!found____ ) {
        printf("No perfect numbers found in the given range.\n");
    }

    return 0;
}
```

low + 1 and high - 1 are also given credit

3. (a) Consider an array A[ ] of n distinct positive integers which are sorted in strictly increasing order, that is,

$0 < A[0] < A[1] < A[2] < \ldots < A[n-1]$.

The following function takes A and n as its only arguments, and finds all the triples from the array, that form a geometric progression with an integral common ratio r. In other words, it prints all triples (A[i], A[j], A[k]) satisfying $0 \leqslant i < j < k \leqslant n-1$ such that A[j] / A[i] = A[k] / A[j] (call this r), and the common ratio r is an integer. If A[ ] contains no such triples, the output will be blank. Also, if there are less than 3 elements in the array, the function will print that there are not enough elements. As an example, let A[ ] = {1, 2, 6, 8, 12, 15, 16, 18, 30, 32, 54, 64}, and $n = 12$. Then, the triples to be reported are (2,6,18), (2,8,32), (1,8,64), (8,16,32), (6,18,54), and (16,32,64). Notice that 12 / 8 = 18 / 12, so (8, 12, 18) is also a GP triple. But the common ratio for this triple is 3/2 which is not an integer, so this triple should not be printed.

Instead of checking whether A[j] / A[i] = A[k] / A[j], the function checks whether A[i] × A[k] = A[j] × A[j]. The two sides of the last equation are stored in the variables `s` and `t`, respectively. The function has a outer loop on j, implying that the middle element in potential triples is fixed first. The inner loop runs on both i and k, and always maintains $i < j < k$. Each iteration of the inner loop either decrements i or increments k or does both. The given assumptions on A[ ] guarantee the correctness of the function.

Fill in the blanks below so that the function works as intended. **[12]**

```
void findGPT ( int A[], int n )
{
    int i, j, k, s, t;

    if (n <= 2) {
        printf("Not enough elements\n");
        return;
    }

    for (j = 0; j < n; j++) {

        t = A[j] * A[j]; /* t remains constant in the iterations of the inner loop */

        /* Initialize i and k for the inner loop */

        i = ____j - 1____ ;  k = ____j + 1____ ;

        /* Inner loop is on (i,k) */

        while ( ____( i >= 0 ) && ( k < n )____ ) {

            s = A[i] * A[k]; /* s needs to be recomputed for each potential candidate (i, j, k) */

            if (s == t) {
                if ( ____A[j] % A[i] == 0____ )

                    printf("(%d,%d,%d)\n", A[i], A[j], A[k]);

                ____i--; k++;____

            } else if (s < t)

                ____k++;____

            } else {

                ____i--;____

            }
        }
    }
}
```

(b) The following program is meant to count the total number of words in a string. For example, if the input string is "This is programming and data structures course", then the output should be 7. Assume that the string consists only of lower-case and upper-case letters, and two consecutive words are separated by a single space or a single tab. Assume also that the string is non-empty, that is, at least one word exists in the string. Fill in the blanks below so that the program works as intended. **[8]**

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

#define str_size 1000 /* Declare the maximum size of the string */

int main() {
    char str[str_size];
    int i, wrd;

    printf("Input the string : ");
    fgets(str, 1000, stdin);
    /* fgets() reads a line including spaces from the terminal, and appends the new-line character
       followed by the null character at the end of the line, and stores the appended input to str
       as a (null-terminated) string. Assume that the length of the input line is < 1000. */

    i = 0;
    wrd = 1;

    while ( str[i] != '\0' ) {

        if ( (str[i] == ' ') || (str[i] == '\t') || (str[i] == '\n') ) {

            ++wrd ;
        }
        i++;
    }

    printf("Total number of words in the string is %d\n", wrd - 1 );

    return 0;
}
```

4. In this exercise, you need to find the k-th derivative of a univariate polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ of degree n, where a_i are real-valued coefficients. The polynomial $p(x)$ is represented by an $(n + 1)$-element array `poly[]` = $\{a_0, a_1, \ldots, a_n\}$, where the coefficient a_i for x^i ($0 \leqslant i \leqslant n$) is stored in the i-th index of the array, that is, as poly[i]. The function `inputPoly()` builds the array `poly[]` by taking inputs from the user. The function also reads and returns the value of k (the degree of the derivative to be computed later). The function `printPoly()` prints the polynomial by only outputting the non-zero terms of the form of $a_i x^i$ with $a_i \neq 0$. The function `derivePoly()` finds the k-th derivative of the polynomial, reconstructs the `poly[]` array to store the derivative (starting from index 0), and returns the degree of the polynomial after the derivative. For example, if $p(x) = 1 + 3x + 4.5x^5 - 7x^{10}$, and $k = 3$, then $p^{(k)}(x) = p'''(x) = 270x^2 - 5040x^7$. So the array `poly[]` was initially { 1, 3, 0, 0, 0, 4.5, 0, 0, 0, 0, –7 }. After the derivative computation, it changes the array *in place* to { 0, 0, 270, 0, 0, 0, 0, –5040 }, and returns 7.

Fill in the blanks of the following C program that is intended to perform the task stated above.

```
#include <stdio.h>

#define MAX 100000

      int
_____________ inputPoly ( ____________double poly[], int d_____________ )
{
  int expt, k;

  printf("Enter the Polynomial:\n");
  /* iterate over every term of the polynomial */

  for ( expt = 0; ____________expt <= d_____________ ; ++expt ) {

    printf("-- Enter coefficient of x^%d: ", expt);
    /* scan and store coefficients of each term of the polynomial */

    scanf("%_____lf_____", ____________&poly[expt]_____________ );
  }

  printf("Enter derivative degree: ");
  scanf("%d", &k);
  return k;
}


void printPoly ( ____________double poly[], int d_____________ )
{
  int expt;

  printf("The Polynomial:\n");
  /* iterate over every term of the polynomial */

  for ( expt = 0; ____________expt <= d_____________; ++expt ) {

    /* check & proceed to print only non-zero coefficient terms */

    if ( ____________poly[expt]_____________ ) {
      /* print coefficients of each term of the polynomial */

      printf(" + (%_____lf_____) ", ____________poly[expt]_____________ );

      if ( expt > 0 ) printf("x");

      if ( expt > 1 ) printf("^%d ", expt);
    }
  }

  printf("\n");
}
```

[6]

[6]

[8]

```
        int                                  double poly[], int d, int k
_________________ derivePoly ( ___________________________________________ )
{
  int expt, i, coef;

  /* prepare initially coef = k! */
  for ( i = k, coef = 1; i > 0; --i ) coef = coef * i;

  /* update the coefficient of terms from x^k, x^(k+1), ... x^d */

  for ( expt = k; expt <= d; ++expt ) {
    /* update the poly[] array coefficients */

    poly[ ________expt - k________ ] = poly[ ________expt________ ] * coef;

    /* prepare coef for next iteration */

    coef = ____coef / (expt - k + 1) * (expt + 1)____ ;
  }

  /* return the final degree after k-th derivative */

  return ( _____________d - k_____________ ) ;
}


int main()
{
  double poly[MAX];
  int degree, derivative;

  /* reads user inputs, and check for erroneous inputs */
  printf("Enter the degree of polynomial: ");
  scanf("%d", &degree);
  if (degree < 0) return 1;

  derivative = inputPoly(poly, degree);
  if (derivative < 0) return 2;

  /* print the polynomial before taking derivative*/
  printf("Before taking derivative: ");
  printPoly(poly, degree);

  /* calculate derivative and final degree */
  degree = derivePoly(poly, degree, derivative);

  /* print the polynomial after taking derivative */
  printf("After taking derivative: ");
  printPoly(poly, degree);

  if ( degree < 0 )
    printf("Degree of polynomial is less than the derivative to be taken!\n");
  else
    printf("Degree of polynomial (after taking %d-th derivative) is %d\n", derivative, degree);

  return 0;
}
```

5. **(a)** Consider the following recursive function. Its return value equals the total number of calls of the function (including the outermost call).

```
int f ( int n )
{
   int r;                          /* variable storing the return value */

   r = 1;                          /* This call counts as 1 */

   if (n > 1) {
      if (n % 2 == 0)              /* if n is even */
         r += f(n/2);              /* one recursive call */
      else                         /* n is odd */
         r += f(n-1) + f(n+1);     /* two recursive calls */
   }

   return r;
}
```

Derive the return value of `f(9)`. Show all your calculations. **[6]**

It is a good idea to compute `f(1), f(2), . . . , f(9)` [although we do not need some of these values].

```
f(1) = 1
f(2) = 1 + f(1) = 2
f(3) = 1 + f(2) + f(4) = 1 + 2 + 1 + f(2) = 1 + 2 + 1 + 2 = 6
f(4) = 1 + f(2) = 1 + 1 + f(1) = 3
f(5) = 1 + f(4) + f(6) = 1 + 3 + 1 + f(3) = 1 + 3 + 1 + 6 = 11
f(6) = 1 + f(3) = 1 + 6 = 7
f(7) = 1 + f(6) + f(8) = 1 + 7 + 1 + f(4) = 1 + 7 + 1 + 3 = 12
f(8) = 1 + f(4) = 1 + 3 = 4
f(9) = 1 + f(8) + f(10) = 1 + 4 + 1 + f(5) = 1 + 4 + 1 + 11 = 17
```

(b) Let n be a positive integer. A *composition* of n is a way of expressing n as a sum of positive integer-valued summands (terms). For example, 2 + 1 + 2 + 5 is a composition of 10. Compositions are called *ordered* if we treat the order of the summands in the sum as important. For example, the ordered composition 2 + 1 + 2 + 5 of 10 is not the same as the ordered composition 2 + 5 + 1 + 2 of 10. An ordered composition is called *palindromic* if the summands read the same both forward and backward. Two examples are given below.

All palindromic ordered compositions of 5	All palindromic ordered compositions of 6
1 + 1 + 1 + 1 + 1	1 + 1 + 1 + 1 + 1 + 1
1 + 3 + 1	1 + 1 + 2 + 1 + 1
2 + 1 + 2	1 + 2 + 2 + 1
5	1 + 4 + 1
	2 + 1 + 1 + 2
	2 + 2 + 2
	3 + 3
	6

In this part, we develop a recursive function `palin()` to print all palindromic ordered compositions of a positive integer n supplied by the user in the `main()` function. The function works as follows.

The function uses an array $A[\,]$ for storing the summands to the left of the center. Since we are interested in palindromic compositions only, the summands to the right of the center appear in the reverse order as they appear in $A[\,]$. Suppose that the first k summands are selected so far. The array $A[\,]$ stores these as $A[0], A[1], A[2], \ldots, A[k-1]$. This corresponds to a (partially prepared) composition of the form:

$$n = A[0] + A[1] + A[2] + \ldots + A[k-1] + \boxed{\text{remaining sum } r \text{ to be realized}} + A[k-1] + \ldots + A[2] + A[1] + A[0].$$

The remaining sum r at the center, to be broken down further, is $r = n - 2\times\big(A[0] + A[1] + A[2] + \ldots + A[k-1]\big)$. There are three possibilities. If $r = 0$, then we already have a composition of n. Then, consider $r > 0$. We can choose the next summand a as any of the values $1, 2, 3, \ldots, \lfloor r/2 \rfloor$. For each such value of a, we set $A[k] = a$, and make a recursive call. Finally, we can also choose $a = r$, and obtain a composition of n. Fill out the details of the program below, that uses this idea. The variables n, A, k, and r should be used as described above. **[14]**

```
void palin ( int n , int A[] , int k , int r )
{
   int a, i;

   /* Choose a from the allowed values, and make a recursive call for each choice of a */

   for ( ____a = 1____ ; ____a <= r / 2____ ; ____++a____ ) {   /* loop on a */

      A[k] = a ;

      palin( ____n____ , ____A____ , ____k + 1____ , ____r - 2 * a____ );

   }

   /* Now, handle the two cases r = 0 and a = r, and print the composition obtained.               */
   /* A single + sign should be used between two consecutive summands, but not at beginning or end. */

   a = r;   /* Choose the next summand as the entire of r (r may be 0) */

   if ( ____k > 0____ ) {   /* print A[] forward */

        printf("%d", A[0]);
        for (i = 1; i < k; ++i) printf(" + %d", A[i]);
        if (a) printf(" + ");

   }
   if ( ____a > 0____ ) printf("%d", a);

   if ( ____k > 0____ ) {   /* print A[] backward */

        for (i = k-1; i >= 0; --i) printf(" + %d", A[i]);

   }
   printf("\n");
}

int main ( )
{
   int n, A[100];

   printf("Enter a positive integer: "); scanf("%d", &n);  /* Assume that n is positive */

   palin( ____n____ , ____A____ , ____0____ , ____n____ );
   return 0;
}
```

This line was missing in the question paper.
Students are not required to write this statement.

Space for rough work

Space for rough work

Space for rough work

Space for rough work