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
END-SEMESTER EXAMINATION
9AM – 12PM, 22-APRIL-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. You may or may not stitch the supplementary sheets to this question paper, but any such supplementary sheet must be submitted along with this question paper.
- 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 KGP number is a positive integer n which can be expressed as $n = 7^x + 2^x + 1^x + 3^x + 0^x + 2^x$ (inspired from the IITKGP pincode 721302) or equivalently as $n = 7^x + 3^x + 2^{x+1} + 1$, where x is a positive integer. Given a positive integer n, you want to find out whether n is a KGP number or not, and print the value of x for KGP numbers. Complete the following program by filling in the blanks, to do the task mentioned above. Math library functions (like pow) cannot be used in this program. **[6]**

```
#include <stdio.h>

int main ()
{
    long int pow7 = 1, pow3 = 1, pow2 = 2, num, check = 0, x = 0;

    printf("Enter a positive integer: "); scanf("_____%ld_____", &num);

    /* indicate, in the loop condition below, how long you continue the search for x */

    while ( _____num > check_____ ) {

        /* update x and the powers of 7, 3, 2 */

        ++x;

        pow7 = _____pow7 * 7_____ ;

        pow3 = _____pow3 * 3_____ ;

        pow2 = _____pow2 * 2_____ ;

        /* create the next KGP number as check */

        check = _____pow7 + pow3 + pow2 + 1_____ ;
    }

    if ( _____num == check_____ )

        printf("%d is a KGP number with x = _____%ld_____\n", num, x);

    else
        printf("%d is not a KGP number\n", num);

    return 0;
}
```

Note: It was our oversight to print num as %d. With that, the code compiles with a warning message. In any case, full credit is given if you print num (last blank) or scan num (first blank) as %d or as %ld (%i and %li also work).

2. Consider a sequence $(x_1, x_2, \ldots, x_n)$ of n positive integers. We define a *zigzag* subsequence of length k as a contiguous block $(x_i, x_{i+1}, \ldots, x_{i+k-1})$ which satisfies the condition $x_i \leqslant x_{i+1} \geqslant x_{i+2} \leqslant x_{i+3} \geqslant \ldots$ up to x_{i+k-1}. For example, consider this sequence of 14 integers: (4, 6, 1, 3, 12, 36, 4, 26, 6, 21, 10, 18, 21, 4). Some zigzag subsequences from this sequence are (4, 6, 1, 3), (4, 26, 6, 21, 10), (12, 36, 4, 26, 6, 21, 10, 18), and (18, 21, 4). Your task is to find the length of a/the longest zigzag subsequence. In this example, the longest one is (12, 36, 4, 26, 6, 21, 10, 18) having the length 8. Complete the blanks in the following program to do the above task. In particular, you need to read a sequence of positive integers, and then calculate and print the length of the longest zigzag subsequence. The program accepts input until the user enters 0 (zero) or a negative number. **[6]**

```
#include <stdio.h>

int main ()
{
    int curNum, prevNum, curLen = 1, maxLen = 0, incrementFlag = 1;

    printf("Enter an integer: "); scanf("%d", &curNum);

    while ( ____curNum > 0____ ) {

        prevNum = curNum;

        printf("Enter next integer: "); scanf("%d", &curNum);

        if ( curNum <= 0 ) {

            if ( curLen > maxLen ) maxLen = ____curLen____;

            break;
        }

        if ( incrementFlag ) {

            if ( ____curNum >= prevNum____ ) ++curLen;

            else if ( curLen > maxLen ) {

                maxLen = ____curLen____;

                curLen = ____1____;

                incrementFlag = ____0____;
            }
        }
        else {

            if ( ____curNum <= prevNum____ ) ++curLen;

            else if ( curLen > maxLen ) {

                maxLen = ____curLen____;

                curLen = ____1____;
            }
        }

        incrementFlag = ____!incrementFlag (or 1 - incrementFlag)____;
    }

    printf("Length of longest zigzag subsequence = %d\n", maxLen);

    return 0;
}
```

3. In numerical analysis, the bisection method (also called the root-finding method) is an algorithm for finding the *zeros* (or *roots*) of continuous functions. A zero (or root) of a function f is a real number x such that $f(x) = 0$. Given an interval $[a, b]$ with $f(a)$ and $f(b)$ having opposite signs, the bisection method divides the interval into two equal halves $[a, c]$ and $[c, b]$, and checks whether $f(c)$ is zero or not. If $f(c) = 0$, we are done. Otherwise, we choose the half such that the signs of f are different at the two endpoints. We again divide that subinterval into two equal halves, and so on. Because these operations involve floating-point calculations, and a zero of f need not have an exact representation as a floating-point number, we may never achieve $f(c) = 0$ for any floating-point value c. We stop the search when the width of the search interval falls below a predefined threshold. The middle point of that narrow interval is then declared as an (approximate) zero of f. Fill in the blanks in the following program to find an (approximate) zero of the function $f(x) = x^3 - 18$ by the bisection method. **[9]**

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

#define f(x) ((x)*(x)*(x)-18)
#define threshold 1e-6

int main ()
{
    double left, right, mid, width;
    int i = 0;

    printf("Enter input interval: "); scanf("%lf%lf", &left, &right);

    if ( (f(left) * f(right)) > 0 ) printf("Invalid Interval!\n");

    else if ( ____(f(left) == 0) || (f(right) == 0)____ ) {

        /* One of the interval boundaries is a zero of f */

        printf("A zero of f is %10.6lf\n", ____(f(left) == 0) ? left : right____ );
    } else {
        printf("ITR \t LEFT \t\t RIGHT \t\t MID \t\t f(MID)\n");
        do {
            ++i;
            mid = ____(left + right) / 2____ ; /* Find the middle point */

            printf("%2d \t %10.6lf \t %10.6lf \t %10.6lf \t %10.6lf\n",
                                                  i, left, right, mid, f(mid));

            if ( ____f(mid) == 0____ ) {

                printf("A zero of f is %10.6lf\n", ____mid____ );
                break;

            } else if ( ____f(left) * f(mid) < 0____ )

                right = ____mid____ ;

            else

                left = ____mid____ ;

            width = right - left;

        } while ( ____width >= threshold____ ) ;

        if ( ____f(mid) != 0____ ) /* if no zero of f is found so far */

            printf("A zero of f is %10.6lf\n", ____(left + right) / 2____ );
    }
    return 0;
}
```

4. Start with a positive integer n. Keep on adding the digits of the number until the sum comes down to a single digit d. If n and d are of the same parity, then n is called an *odd-unity* number if n is odd, or an *even-unity* number if n is even. (By definition, we cannot have odd even-unity numbers and even odd-unity numbers.) For example, 963 is an odd-unity number because $9+6+3=18$, and then $1+8=9$; whereas 965 is not an odd-unity number because $9+6+5=20$, and then $2+0=2$. Similarly 584 is an even-unity number because $5+8+4=17$, and then $1+7=8$; whereas 586 is not an even-unity number because $5+8+6=19$, and then $1+9=10$, and then $1+0=1$. Your task is to complete the following program to list all odd-unity and even-unity numbers within a specified range $[1, R]$, where the end R of the range will be a user input. **[10]**

```
#include <stdio.h>

/* recursive function to calculate the sum of the digits in num */

unsigned int digitSum ( unsigned int num )
{
    if ( ____num > 0____ )

        return ( ____num % 10 + digitSum(num / 10)____ );
    return 0;
}

/* recursive function to compute and print odd/even-unity numbers. flag = 1 means search for
   odd-unity numbers, flag = 0 means search for even-unity numbers */

void recEvenOdd ( unsigned int start, unsigned int end, int flag)
{
    int sum = start;

    if ( ____start <= end____ ) {

        do {

            sum = digitSum( ____sum____ );

        } while (sum > 9);

        if ( ____sum % 2____ == flag )

            printf("  %u", start);

        recEvenOdd( ____start + 2____ , ____end____ , ____flag____ );
    }
}

int main ()
{
    unsigned int range;

    printf("Input range from 1 to "); scanf("%u", &range);

    printf("\nOdd-unity numbers within [1-%u]:\n", range);

    recEvenOdd( ____1____ , ____range____ , ____1____ );

    printf("\n Even-unity numbers within [1-%u]:\n", range);

    recEvenOdd( ____2____ , ____range____ , ____0____ );

    printf("\n");

    return 0;
}
```

5. Answer the following questions. In all the parts, show your calculations in the respective boxes provided.

(a) Convert the base-9 number $(145.2)_9$ to its equivalent number in base 4. **[2]**

Converting to Decimal:
Integer part: $(145)_9 = 1 \times 9^2 + 4 \times 9^1 + 5 \times 9^0 = 81 + 36 + 5 = (122)_{10}$
Fractional part: $(0.2)_9 = 2/9 \approx (0.2222\ \ldots)_{10}$
So, Combined: $(145.2)_9 = (122.2222)_{10}$

Decimal to Base 4:
Integer part: $122/4 = 30$ (remainder = 2), $30/4 = 7$ (remainder = 2),
$7/4 = 1$ (remainder = 3), $1/4 = 0$ (remainder = 1)
So, $(122)_{10} = (1322)_4$
Fractional part: $0.2222 \times 4 = 0.8888 \rightarrow 0$, $0.8888 \times 4 = 3.5552 \rightarrow 3$,
$0.5552 \times 4 = 2.2208 \rightarrow 2$, $0.2208 \times 4 = 0.8832 \rightarrow 0$,
$0.8832 \times 4 = 3.5328 \rightarrow 3$, $0.5328 \times 4 = 2.1312 \rightarrow 2$
We can stop according to the precision as needed. So, $(0.2222)_{10} = (0.032032032\ \ldots)_4$

So, finally we get, $(145.2)_9 = (1322.032032)_4$

(b) Find value of the base x from the following equation: $(56)_x + (72)_x = (150)_x$. **[2]**

Taking into consideration of x radix, we get the following equation:

$(5 \times x^1 + 6 \times x^0) + (7 \times x^1 + 2 \times x^0) = 1 \times x^2 + 5 \times x^1 + 0 \times x^0$
implies, $5x + 6 + 7x + 2 = x^2 + 5x$
implies, $x^2 + 7x - 8 = 0$
implies, $x = 8$ or -1 (two roots)

Since -1 is impossible as base, so $x = 8$

Another reasoning:

The summation of the two LSD's gives a carry of 0 or 1. In this case, we therefore have $6 + 2 = 00$ or 10 in base x. Clearly, 00 is not possible. That is, $6 + 2 = 10$ in base x, that is, $x = 8$.

One may also say that $6 + 2 = 8$ is a multiple of x. But there should be an argument why x cannot be 4 or 2 (like x-ary digits must be less than x).

(c) Suppose that $(2536\mathbf{Y})_8 = (\mathbf{Y}AF2)_{16}$. What will be the value of **Y**? **[2]**

From LHS,
$(2536Y)_8 = 2 \times 8^4 + 5 \times 8^3 + 3 \times 8^2 + 6 \times 8^1 + Y \times 8^0 = (10992 + Y)_{10}$

From RHS,
$(YAF2)_{16} = Y \times 16^3 + 10 \times 16^2 + 15 \times 16^1 + 2 \times 16^0 = (4096\ Y + 2802)_{10}$
(A) (F)

So, we get, $10992 + Y = 4096\ Y + 2802$, implies that, $Y = 2$

(d) Find the base b of the number system such that the following equality holds: $(321 / 20)_b = (13.3)_b$. **[2]**

Given that, $(321 / 20)_b = (13.3)_b$
implies, $(321)_b = (13.3)_b \times (20)_b$
implies, $3 \times b^2 + 2 \times b^1 + 1 \times b^0 = (1 \times b^1 + 3 \times b^0 + 3 \times b^{-1}) \times (2 \times b^1 + 0 \times b^0)$
implies, $b^2 - 4b - 5 = 0$
implies, b = 5 or –1 (two roots)

Since –1 is impossible as base, so b = 5

(e) State the range of 1's complement and 2's complement numbers for n-bit representation. **[2]**

1's Complement: $-(2^{n-1} - 1)$ to $+(2^{n-1} - 1)$

2's Complement: $-(2^{n-1})$ to $+(2^{n-1} - 1)$

(f) Represent the decimal number **–112** in 8-bit 2's complement form. **[2]**

First, let us represent 112 in 8-bit binary format: 0 1110000

So, 1's complement of the above number: 1 0001111
On adding 1 to it, we get the 2's complement of the above number: 1 0010000

So, –112 is 8-bit 2's complement form: 10010000

(g) Convert the decimal number **−118.8125** into its 32-bit IEEE 754 single-precision floating-point representation. Then, show the HEXADECIMAL value of the floating-point representation for this. **[4]**

The given decimal number is −118.8125

Therefore, the sign bit is 1 (to indicate negative number)

Now, $(118)_{10} = (1110110)_2$ and $(0.8125)_{10} = (0.1101)_2$
So, $(118.8125)_{10} = (1110110.1101)_2$

The above can be written as: 1.1101101101×2^6
$= 1.1101101101 \times 2^{133-127}$
So we get,
Exponent: $(133)_{10} = (10000101)_2$
Mantissa: $(1101101101)_2$

So, the 32-bit IEEE754 single-precision floating point format representation of −118.8125 is as follows:

1	1 0 0 0 0 1 0 1	1 1 0 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0
Sign	Exponent	Mantissa

that is, 1100 0010 1110 1101 1010 0000 0000 0000. Grouping by four each from LSB, we get **C2EDA000.**

Alternate answer to the HEX question. From the floating-point representation, the number is:

111 0110 . 1101 0000 0000 0000 0

or in HEX **–76.D0000**.

(h) Using 8-bit 2's complement representation, perform the subtraction operation: **126 – 37**. Show both 8-bit and decimal representations of the result. **[4]**

First of all,
$(+126)_{10}$ = $(01111110)_2$
$(+37)_{10}$ = $(0100101)_2$

2's complement of +37 will be: $(–37)_{10} = (11011011)_2$

So, 126 – 37 = 126 + (–37)
= 01111110 + 11011011
= 1 01011001

Hence, we get an overflow of 1. Keeping that aside, we get:

Result: $(01011001)_2 = (89)_{10}$

6. You are given an array A of n non-negative integers. For a non-negative integer a, nextprime(a) is defined to be the smallest prime greater than or equal to a. Your task is to update the array A in such a way that every element $A[i]$ of A gets promoted to nextprime($A[i]$). For example, the original array A = {1, 14, 8, 3, 25, 48, 0, 2} should be updated to {2, 17, 11, 3, 29, 53, 2, 2}. Note that 0 and 1 are not considered to be prime numbers, and will be promoted to 2. The following code solves this problem. Its `main()` function is first given.

```
int main ()
{
   int n, m, i, A[MAX_SIZE], P[MAX_SIZE], pcnt;

   printf("n = "); scanf("%d", &n);          /* Read the size of A */
   for (i=0; i<n; ++i) scanf("%d", &A[i]);   /* Read the elements of A (assumed non-negative) */
   prnarray(A, n);                           /* Print the original A */
   m = findmax(A,n);                         /* Find the maximum element of A */
   pcnt = primelist(m,P);                    /* Store in P all the primes up to nextprime(m) */
                                             /* pcnt is the number of primes stored in P */
   for (i=0; i<n; ++i)
      A[i] = nextprime(A[i], P, pcnt);       /* Promote every element of A to its nextprime */
   prnarray(A, n);                           /* Print the updated A */
   return 0;
}
```

In the rest of this exercise, you complete an implementation of the functions `nextprime()` and `primelist()`. The function `nextprime()` is given first. The function assumes that P is a sorted array of the first *pcnt* primes. A second assumption is that for every a sent to the function, P stores `nextprime(a)`. Note that a may or may not itself be a prime. The function returns the smallest element of P, that is greater than or equal to a. To that end, it runs a binary search in P. Unlike the implementations of binary search covered in the class and the slides, an index in P (or –1) is not returned here. Instead (as the name of the function suggests), the function returns the element of P, equal to nextprime(a) (which exists in P under the second assumption mentioned above). After the initialization of the search interval $[L, R]$, the binary-search loop should always maintains the inequality $P[L-1] < a \leqslant P[R]$. **[5]**

```
int nextprime ( int a, int P[], int pcnt )
{
   int L, R, M;

   if ( a <= P[0] ) return P[0];

   L = ____1____ ; R = ____pcnt - 1____ ;

   while ( ____L < R____ ) {

      M = ____(L + R) / 2____ ;

      if ( ____a <= P[M]____ ) R = M;

      else L = M + 1;
   }

   return P[L];
}
```

The function `primelist()` comes next. This is based on the *sieve of Eratosthenes* (known since the third century BC). Suppose that we want to find all the prime numbers $\leqslant$ a positive limit L. To start with, we mark all the numbers 2, 3, . . . , L as prime. Then, we run a loop for i = 2, 3, 4, ... If i is already recognized as composite (not prime), we continue to the next iteration. Otherwise, that is, if i is still marked as a prime, it is indeed a prime. But all proper multiples of i, that is, 2i, 3i, 4i, ... cannot be prime, so we mark these multiples as not primes. The iterations of the sieve are illustrated below for L = 30. Numbers recognized as non-prime are struck through.

Initialization	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Iteration i = 2	2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	21	~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~
Iteration i = 3	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	25	~~26~~	~~27~~	~~28~~	29	~~30~~
Iteration i = 5	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~	~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~

In this example, the loop for $i = 4$ is not shown (because 4 is already marked as composite, so there are no changes in the markings in that iteration). The example illustrates that it suffices to run the iterations for $2 \leqslant i \leqslant \sqrt{L}$ (because all composite number $\leqslant L$ must have a prime divisors $\leqslant \sqrt{L}$). Note also that some numbers are marked multiple times (like 15 is marked twice, first as a multiple of 3, and once more as a multiple of 5), but this does not matter.

A problem we need to address here is that we have computed the maximum m in A. But m itself need not be prime, and the function `nextprime()` requires nextprime(m) to reside in P. So how do we set the limit L? Assume that $m \geqslant 25$. It is known (since 1952) that in this case, there exists at least one prime p in the range $m < p < 6m/5$. Therefore we can take $L = 6m/5$. We will however stop recording primes as soon as the first prime $\geqslant m$ is located.

Complete the following implementation of `primelist()` based on the above ideas. **[10]**

```
int primelist ( int m, int P[] )
{
   int *A, L, i, imax, j;

   L = 6 * m / 5;  /* Set the limit up to which primes need to be generated */

   /* The array A stores the marking of elements as prime or composite. A[i] = 1 means i is (still)
      classified as prime. A[i] = 0 is means i is known to be composite (not prime). We are not
      interested about i = 0, 1. For simplicity, we however have indices of A ranging from 0 to L,
      although we will never use A[0] and A[1]. Allocate memory to A[] accordingly. */

   A = ( ____int *____ )malloc( ____(L + 1) * sizeof(int)____ );

   /* Initially mark all i of interest as primes */

   for ( ____i = 2; i <= L; ++i____ ) A[i] = 1;
   imax = sqrt(L);  /* The maximum value √L of i to run the following iteration */
   for (i = 2; i <= imax; ++i) {

      /* if i is now proved to be prime */

      if ( ____A[i] == 1____ ) {
         /* Mark all proper multiples of i as not prime (write a loop on j) */

         ____for (j = 2*i; j <= L; j += i) A[j] = 0;____
      }
   }

   /* Now store in P the i's marked as prime. i is for reading from A, and j is for writing to P. */
   j = 0;
   for (i=2; i<=L; ++i) {

      if ( ____A[i] == 1____ ) { /* if i is prime */
         /* Record i in P */

         ____P[j] = i; ++j; (or P[j++] = i;)____

         if ( ____i >= m____ ) break; /* break as soon as nextprime(m) is stored */
      }
   }

   /* A is no longer needed, so release the memory allocated to A */

   ____free(A);____

   return ____j____ ; /* number of primes stored in P */
}
```

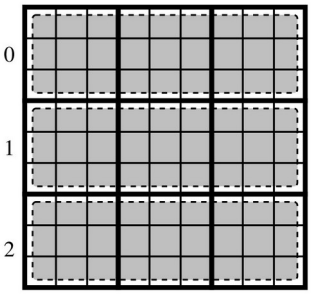
7. Sudoku is a board game conventionally played on a 9×9 grid consisting of nine 3×3 blocks. Each cell in the grid contains a single digit in the range 1 – 9 such that no row nor column nor block contains a repetition of digits. In order to convert the board to a puzzle, entries in some of the cells are deleted. Players are required to put the missing digits in the empty cells in order to complete the solution. A good Sudoku puzzle offers a unique solution. The following figure shows a Sudoku puzzle (Part (a)) and its solution (Part (b)). The *blocks* are shown as thick-edged squares. Three blocks appearing side by side from left to right constitute a *row band* (Part (c)). Likewise, three blocks appearing side by side from top to bottom constitute a *column band* (Part (d)). The numbering of the rows, columns, and bands is also explained in the figure below.

	0	1	2	3	4	5	6	7	8
0					2	4		8	
1	4	8	7			1	9		
2	9		3			6		1	
3	3	9			8	7	2		
4			8		1	3	4		9
5	1	7	2					3	6
6		5	9						8
7					4			2	
8			6	1	5		7		

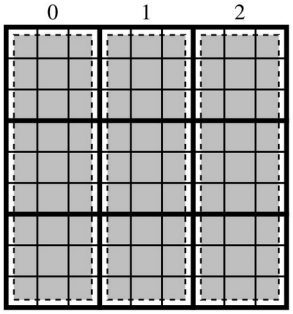
(a) A Sudoku puzzle

6	1	5	9	2	4	3	8	7
4	8	7	5	3	1	9	6	2
9	2	3	8	7	6	5	1	4
3	9	4	6	8	7	2	5	1
5	6	8	2	1	3	4	7	9
1	7	2	4	9	5	8	3	6
7	5	9	3	6	2	1	4	8
8	3	1	7	4	9	6	2	5
2	4	6	1	5	8	7	9	3

(b) Solution of the puzzle



(c) Row bands



(d) Column bands

A strategy to generate random Sudoku puzzles is to start with a base solution (like the one given below), and then make random permutations of the rows and columns, that do not violate the no-repetition property of Sudoku boards. Finally, some cells in the 9×9 grid are carefully chosen for deletion so that the solution remains unique.

The `main()` function of a random Sudoku puzzle generator is given below. Your task is to complete the functions called from `main()`, so that the program works as intended. You do not have to write the function `prn_mat()`.

```
int main () {
   int A[9][9] = { { 9, 3, 2, 8, 6, 7, 1, 5, 4 },
                   { 4, 7, 8, 5, 1, 3, 6, 9, 2 },
                   { 6, 5, 1, 9, 4, 2, 8, 3, 7 },
                   { 5, 8, 6, 2, 3, 1, 7, 4, 9 },
                   { 3, 4, 9, 6, 7, 8, 5, 2, 1 },
                   { 1, 2, 7, 4, 5, 9, 3, 8, 6 },
                   { 2, 6, 4, 1, 8, 5, 9, 7, 3 },
                   { 8, 1, 3, 7, 9, 4, 2, 6, 5 },
                   { 7, 9, 5, 3, 2, 6, 4, 1, 8 } };     /* A base solution */

   row_permute(A);      /* permute rows */
   col_permute(A);      /* permute columns */
   del_cells(A);        /* delete digits from cells */
   prn_mat(A);          /* print the puzzle */
   return 0;
}
```

In order to generate random numbers, we use the function `rand()` which returns a random integer r in the range $[0, 2^{31} - 1]$. If we take the remainder of division of r by a (small) positive integer m, we get a random integer in the range 0, 1, 2, . . . , $m - 1$. For example, the statement `i1 = rand() % 3` (in the function `row_permute()` below) assigns to `i1` a random integer from 0, 1, 2.

First, complete the function `row_permute()` below. Note that permuting the rows in a row band is safe (that is, does not violate the constraints of Sudoku boards). The function below chooses a pair of distinct rows in each row band, and swaps them element by element. We only pass the two-dimensional array A to the function, because we know its size beforehand. **[3]**

```
void row_permute ( ____________________int A[][9]____________________ ) {
   int rowband, i1, i2, j, t;

   for (rowband=0; rowband<3; ++rowband) {  /* for each row band, do */

      /* Generate two distinct rows i1 and i2 from 0, 1, 2 in that band */
      i1 = rand() % 3;
      do { i2 = rand() % 3; } while (i1 == i2);
```

```
        /* Convert i1 and i2 to row indices in A[][] */
        i1 += rowband * 3 ;
        i2 += rowband * 3 ;

        /* Swap the i1-th and the i2-th row of A[][]. Use the variable t for swapping values. */

        for ( ____j=0; j<9; ++j____ ) {  /* loop on j */

            ____t = A[i1][j]; A[i1][j] = A[i2][j]; A[i2][j] = t;____
        }
    }
}
```

The function `col_permute()` is similar. In each column band, two distinct columns `j1` and `j2` are chosen, and swapped element by element. Use a logic similar to the function `row_permute()`. The only parameter to the function is the two-dimensional array A. **[3]**

```
void col_permute ( ____int A[][9]____ ) {
   int colband, i, j1, j2, t;

   for (colband=0; colband<3; ++colband) {
      /* Generate distinct j1 and j2 from 0, 1, 2 */
      j1 = rand() % 3;
      do { j2 = rand() % 3; } while (j1 == j2);

      /* Convert j1 and j2 to column indices in A */
      j1 += colband * 3;
      j2 += colband * 3;

      /* Swap the two column in A[][] using the variable t */

      for ( ____i=0; i<9; ++i____ ) {  /* loop on i */

          ____t = A[i][j1]; A[i][j1] = A[i][j2]; A[i][j2] = t;____
      }
   }
}
```

Your final task is to complete the digit-deletion function `del_cells()` given below. A deleted cell in A stores 0 (that is, deleting a cell storing the digit d in the range 1 – 9 replaces that d by 0). The function makes a predefined maximum number of attempts. In each attempt, a random row r and a random column c are chosen. If A[r][c] is already 0, that cell has been deleted earlier, so we continue to the next attempt. So suppose that A[r][c] ≠ 0. Define the *neighborhood* of the cell A[r][c] to consist of all the digits appearing in the r-th row, the c-th column, and the same block as A[r][c]. We denote this set as nbd(A[r][c]). If all the digits 1 – 9 appear in nbd(A[r][c]), it is safe to delete A[r][c], that is, to set A[r][c] = 0 (if this is the case, the other members of nbd(A[r][c]) uniquely determines A[r][c]). Otherwise, deleting A[r][c] may be unsafe, and we do not delete A[r][c].

The figure to the right gives examples of when it is safe and when it is unsafe to make a deletion. In each part, the neighborhood of a cell is shown as a set of shaded cells. In Part (a), the neighborhood contains all the digits 1 – 9 (it also contains 0, because there are deleted cells in the neighborhood, although this does not matter), so it is safe to delete the highlighted 8. In Part (b), the neighborhood does not contain 2, so it may be unsafe to delete the highlighted 5, that is, the deletion is not carried out in this case.

6	1	5		2	4	3	8	7
4		7	5	3	1		6	
9	2	3	8	7				4
		4	6	8	7		5	
5	6		2	1		4		
		2	4	9		8	3	
7	5	9		6		1		
8		1	7	4		6		5
			1	5		7	9	3

(a) Deletion of 8 is safe

6	1	5		2	4	3	8	7
4		7	5	3	1		6	
9	2	3	8	7				4
		4	6	8	7		5	
5	6		2	1		4		
		2	4	9			3	
7	5	9		6		1		
8		1	7	4		6		**5**
			1	5		7	9	3

(b) Deletion of 5 is not safe

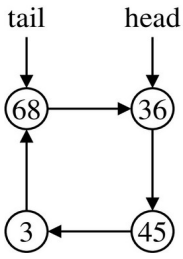
In order to decide whether the deletion of a cell is safe, we use an array of 10 flags initialized to all zeros (implying an empty neighborhood). Then, we look at the row, the cell, and the block containing A[r][c], and mark all digits present there by setting the corresponding flag to 1. Finally, we make a check whether the neighborhood contains all the digits 1 – 9. If so, we delete A[r][c]. Otherwise, we continue to our next attempt.

Complete the function below to delete cells. The input is only the array A. Assume that `MAX_ATTEMPT` is defined earlier. **[11]**

```
void del_cells ( ____int A[][9]____ ) {
   int i, j, r, c, attempt, rowband, colband, nbd[10];

   for (attempt=0; attempt<MAX_ATTEMPT; ++attempt) {
      r = rand() % 9; c = rand() % 9;


      if ( ____A[r][c] != 0____ ) { /* if cell is not already deleted */

         for ( i=0; i<=9; ++i ) nbd[i] = 0;  /* Initialize neighborhood */
         /* Look at the r-th row */

         for ( j=0; j<9; ++j ) nbd[ ____A[r][j]____ ] = 1;
         /* Look at the c-th column */


         for ( i=0; i<9; ++i ) nbd[ ____A[i][c]____ ] = 1;
         /* Compute the row band and the column band of the cell A[r][c] */


         rowband = ____r / 3____ ; colband = ____c / 3____ ;
         /* Look at all cells (i,j) in the block of A[r][c] */


         for ( ____i=3*rowband; i<3*(rowband+1); ++i____ ) {   /* Loop on i */


            for ( ____j=3*colband; j<3*(colband+1); ++j____ ) { /* Loop on j */


               nbd[ ____A[i][j]____ ] = 1;
            }
         }
         /* Final check whether it is safe to delete A[r][c] */

         for ( i=1; i<=9; ++i ) {


            if ( ____nbd[i] == 0____ ) break;
         }

         /* if it is safe to delete A[r][c] */


         if ( ____i == 10____ ) A[r][c] = 0;
      }
   }
}
```

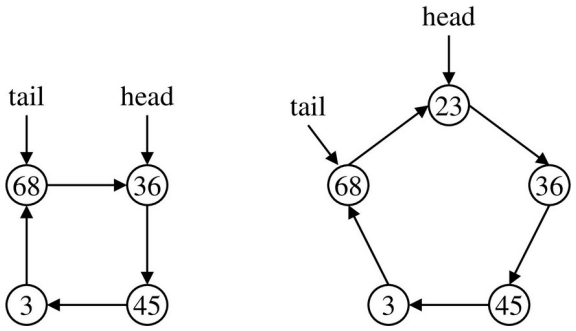
This program generates random puzzles from the same base solution. But players may eventually detect the pattern of the base solution. A real-life application may start with a database of several predetermined base solutions, and pick one randomly for each puzzle. Another approach is to randomly generate the base solution too. But this involves some algorithms well beyond the scope of this exercise (perhaps of this course too).

8. A *circular linked list* **(***CLL***)** is a data structure where the *next* pointer of the last (*tail*) node points back to the first (*head*) node, forming a circular loop. We represent a CLL by its tail pointer, that is, the head pointer is not explicitly stored. Whenever needed, the head pointer can be accessed by following the pointer at the tail node. A circular linked list is shown in the adjacent figure. Although the head pointer is shown, we refer to this list by its tail pointer. In this exercise, you are asked to write two functions to do the following: **(a)** insert a node at the head of a CLL, and **(b)** reverse a CLL (with respect to the head). The structure of a node is defined as follows.




```
typedef struct _node {
    int info;
    struct _node *next;
} node;
```

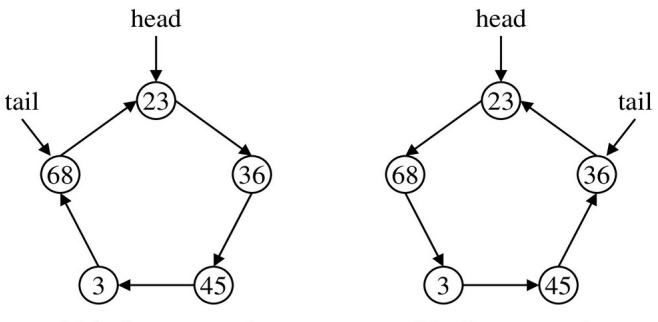
First, complete the following function for inserting an integer item `x` at the head of a CLL. The `main()` function calls this as `insertcll(x,tail)`. The new node will be the new head, and the pointer of the tail node will point to this new head. However, if the list passed to the function is empty (tail pointer is NULL), then both head and tail will point to the new node created. In either case, the tail pointer is returned (it changes from NULL to a new node pointer during the first insertion). The adjacent figure shows a CLL before (left side of the figure) and after (right side) the insertion of 23.



[7]

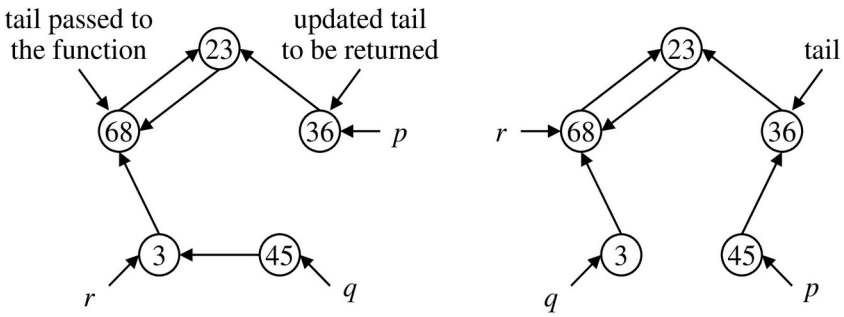
```
node *insertcll ( int x, node *tail )
{
   node *p;

   /* Allocate memory to p */

   p = ( ____node *____ )malloc( ____sizeof(node)____ );
   p -> info = x;

   if ( ____tail == NULL____ ) {    /* insertion in an empty list */

      p -> next = ____p____ ;

      return ____p____ ;
   }

   p -> next = ____tail -> next____ ;

   ____tail -> next____ = p;

   return ____tail____ ;
}
```

Next, complete the reversal function. The reversal is relative to the head, and is illustrated in the adjacent figure. Initially, the pointers in the nodes point in the clockwise direction. After the reversal, all these pointers point in the counterclockwise direction. We pass only the tail pointer of the list to the reversal function. As the links are reversed, the head pointer (not maintained explicitly though) should not change. But the tail pointer changes, and this new tail pointer needs to be returned.



(a) before reversal (b) after reversal

The function `reversecll()` below carries out this reversal by modifying the pointers in the nodes (not by swapping the `info` values stored in the nodes). The function uses a reversal loop which maintains three pointers *p*, *q*, and *r* pointing to three consecutive nodes in the original list. Each iteration reverses the pointer of the node pointed to by *q*. After that, the three pointers move one node down the original list. An iteration of the reversal loop is explained in the figure below. Note again that the tail pointer changes by the reversal process. You should store the new tail pointer before the reversal loop starts. **[10]**



(a) before an iteration (b) after the iteration

```
node *reversecll ( node *tail )
{
   node *p, *q, *r;   /* no other variables are permitted */

   if (tail == NULL) return ____NULL____ ;

   /* Using the tail pointer passed to the function, initialize p, q, and r so that in the first
      iteration below, the pointer in the head node is reversed to point to the old tail node */

   p = ____tail____ ;

   q = ____p -> next (or tail -> next)____ ;

   r = ____q -> next (or p -> next -> next
          or tail -> next -> next)____ ;

   tail = ____r (or tail -> next -> next or q -> next or
                p -> next -> next )____ ; /* update the tail pointer for returning */

   while ( ____p != r  (or tail -> next != q)____ ) { /* Repeat until all pointers are reversed */

      ____q -> next = p____ ; /* Reverse the pointer of *q */
      /* advance p, q, r down by one node */

      p = ____q____ ;

      q = ____r____ ;

      r = ____r -> next____ ;
   }

   return tail;
}
```

Space for rough work

Space for rough work

Space for rough work

Space for rough work