Indian Institute of Technology, Delhi

FALL, 2016

COL 100: INTRODUCTION TO PROGRAMMING

Minor 2

One Hour

NOTE:

- All answers need to be brief and to the point.
- Please make any assumptions that you deem to be reasonable.
- Follow the *spirit of the question*. Do not immerse yourself in irrelevant details.
- Every answer needs to be written neatly and cleanly in the space provided for it.
- If required, please finalize the code in the space provided for rough work, and then write it cleanly in the space provided for writing your C statements.
- Use proper handwriting, and do not write anything on the margins.
- Note that in some questions, we might not have part marking.
- You are not allowed to carry any electronic gadgets including calculators and mobile phones.
- The closer your answer is to the model answer in terms of the lines of code, the more marks you get.
- The final answer needs to be **written with a pen**.
- There are three additional pages at the end for rough work.
- Even if your final answer is correct, you might not get full marks (or any marks) if the explanation is wrong.
- **NO QUESTIONS WILL BE ANSWERED**
- This question paper **NEEDS TO BE SUBMITTED**. Do not take it with you.

Total Marks: 30
Total Number of Pages : 10

Name:	Group No:	Entry No:

Marks:	1	2	3	4	5	6	Total

1. What is the output of the following program? Explain your answer. (4 marks)

```
int i, j, k = 0;
int n = 100;
for(i = 1; i <= n; i++)
        for(j = 1; j <= i; j++)
                k++;
printf(``%d\n'', k);
```

The output is Ans: 5050

For i=1, the inner loop runs only once and increments k by 1. Then for i=2, the inner loop runs twice and increments k twice. Similarly for i=3, the inner loop runs thrice and increments k by 3. And so on... k therefore is the sum of 1 + 2 + 3 + ... n, = $n(n+1)/2$ in general. For n=100, the answer is 5050.

Marking scheme:

- +1 for the explanation that the programs counts the aggregate number of times the inner loop runs
- +1 for the explanation that the answer is 1 + 2 + 3 + ... n
- +2 for the correct answer, 5050

2. Write the following function in a **recursive** manner. (4 marks)

```
int sum = 0; /* global variable */

void func(int x) {
    while(x > 0) {
        sum += x;
        x--;
    }
}
```

The function ends up calculating the sum of natural numbers for the value of x with which it is invoked. It starts with adding x to sum, then it decrements x by 1 and adds to sum the value of x-1, then adds to sum the value of x-2, and so on, until x is decremented to 0. A recursive version which uses the same logic can be written as follows:

```
int sum = 0; /* global variable */

void func(int x) {
    if(x > 0) {
        sum += x;
        func(x--);
    }
}
```

Marking scheme:

- +1 for a correct application of the concept of recursion by calling the same function from within itself by passing a smaller value of the parameter
- +1 for changing the while loop into an if condition, to serve as a stopping point for the recursion
- +2 for overall correctness of the program

Some students may have changed the return type of the function to return the sum, which is also correct. Essentially any form which uses recursion correctly, and ends up putting the result in sum.

```
int sum = 0; /* global variable */

int func(int x) {
    if(x > 0) {
        sum = x + func(x--);
        return sum;
    }
}
```

3. Complete the following program to express a number in terms of its prime factors and their exponents. For example the number 24 can be expressed as $2^3 \times 3^1$. In general a number(N) can be represented as: $N = p_1^{e_1} \times p_2^{e_2} \times \ldots$. Here, $p_1, p_2, \ldots$ are prime numbers. Note that all the prime numbers and exponents $(e_1, e_2, \ldots)$ are greater than 1. Now, complete the following program to print the prime factorization of a number. For example, for 24, you should print $2**3*3**1$, where $**$ is the exponential operator. (6 marks)

```
#define NUM_PRIMES 10000

/* primes is an array with the first 10,000 prime numbers */
int primes[] = {2, 3, 5, 7, 11, 13, 17, 19, ... };

void print_prime_factorization(int n);

void main() {
        int n;
        /* Assumption: n > 1, and all of its prime factors
         are elements of the primes array */
        scanf(``%d'', &n);
        print_prime_factorization(n);
}

/* complete this function */
void print_prime_factorization(int n){
    int i = 0, j, firstprime = 1;

    while(n > 1 && i < NUM_PRIMES) {
        if(n % primes[i] == 0) {          // check for divisibility by the ith prime
            j = 0;
            while(n % primes[i] == 0) {
                n /= primes[i];           // remove the ith prime from n, j times
                j++;                      // j ends up as the exponent for the prime
            }
            if(firstprime == 0)           // fiddly work to not print * at the start
                printf(" * ");
            else
                firstprime = 0;
            printf("%d ** %d", primes[j], j);
        }
        i++;                              // move to the next prime
    }
    printf("\n");
}
```

The logic is to test for divisibility by all the primes, one by one. Upon finding a prime factor, keep dividing n by it until the prime is no longer a factor and keep track of how

many such factor divisions were performed. This is the exponent for the prime. After removing all the prime factors from n, it eventually reduces to 1 and there is no need to test for more factors.

Marking scheme:

- +1 for correctly checking for divisibility of n by each prime, one by one
- +1 for correctly calculating the exponent with which a prime divides n
- +2 for a good termination condition that after all the prime factors of n have been found, no further checks are performed
- +2 for overall program correctness. Ignore fiddly things like not printing extra *s

4. You can store a 2D 100 x 100 matrix in a 1D array as follows, row by row. (6 marks)

```
int matarray[10000];

/* To access the (row i, column j)th element */
int get_element(int i, int j) {
        return matarray[i * 100 + j];
}
```

Now, if you know that the $(i, j)^{th}$ element is equal to the $(j, i)^{th}$ element, for all i and j, ie. the matrix is symmetric around its diagonal, then we can make some modifications. We need not store all the 10,000 elements in *matarray*. We need not store the duplicate elements. Answer the following questions:

(a) How many elements do we need to store such that duplicate elements are not stored?

Ans: 5050

We only need to store the upper or lower triangle of the matrix. The number of elements to store therefore, assuming we store the lower triangle only, will be 1 + 2 + 3 + ... n. That is, we store the first element of the first row, the first two elements of the second row, the first three elements of the third row, and so on. For n = 100, the answer is 5050.
Marking scheme: +1 for the correct answer

(b) Describe how you would store the elements in a space efficient manner in the 1D *matarray* array.

Answer: Explanation given above.
Marking scheme: +1 for explaining that only the upper or lower triangle needs to be stored

(c) Modify the *get_element* function to reflect this change.

```
int get_element(int i, int j){
    int row, col, pos;

    // to store the lower triangle, the min of (i, j) is the row
    // and max of (i, j) is the column in the lower triangle
    row = (i < j) ? i : j;
    col = (i < j) ? j : i;

    // calculate the position in the 1D array
    pos = (row * (row - 1)) / 2 + col;

    return matarray[pos];
}
```

Assuming we only store the lower triangle, the logic therefore is to store in the 1D array the first element of the first row, the first two elements of the second row, the first three elements of the third row, and so on. The ith row starts from the position which is the sum of elements in the preceding rows, equal to 1 + 2 + 3 + ... (i-1), which is equal to i(i-1)/2. The jth element then in the ith row is stored another j positions down at i(i-1)/2 + j.

This is what the program calculates. Note however that the (i, j)th element in the lower triangle can also be queried as (j, i) for the corresponding duplicate element in the upper triangle. We need to use the minimum value of (i, j) for the row, and the maximum value of (i, j) for the column, to address the lower triangle.

Marking scheme:

- +1 for the correct method of storage in the 1D array
- +1 for the correct calculation of the position of the (i, j)th element
- +1 for assigning the row as the minimum of (i, j) and column as the maximum of (i, j), to store the lower triangle
- +1 for overall correctness

Note that some students may have stored the upper triangle, which leads to a slightly more complex way to calculate the position as n + (n-1) + (n-2) + ... (n - row), plus column. The marking scheme remains the same as above.

5. Write a program using **recursion** to compute the n^{th} power of a number x (compute x^n). Here, $n \geq 0$. (4 marks)

```
int get_nth_power (int x, int n){
    if(n > 1)
        return x * get_nth_power(x, n-1);
    else
        return x;
}
```

The logic is to compute x^n as x times $x^{(n-1)}$, where $x^{(n-1)}$ is calculated through the same function. The terminating condition for the recursion is when n = 1, at which point we have to return x^1.

Marking scheme:

- +1 for writing the recursion as x times the function for (x, n-1)
- +1 for the termination condition
- +2 for overall correctness

6. What will be the output of this program? (6 marks)

```
void swap1(char x, char y) {
        char t;
        t = x; x = y; y = t;
}
void swap2(char str[], int x, int y) {
        char t;
        t = str[x]; str[x] = str[y]; str[y] = t;
}
void swap3(char *x, char *y) {
        char t;
        t = *x; *x = *y; *y = t;
}
void main() {
        char str[] = "hello";
        char str1[] = "hello";
        char str2[] = "hello";

        swap1(str[0], str[4]); printf("%s\n", str);
        swap2(str1, 0, 4); printf("%s\n", str1);
        swap3(&str2[0], &str2[4]); printf("%s\n", str2);
}
```

Will the output be different if swap3 is invoked as swap3(str, str + 4)? Please explain all your answers.

The output for the first printf is Ans: hello
The parameters passed to the swap function are just the values of the characters of the array and therefore while the swap of the character values occurs inside the swap function, the actual array is not changed.

Marking scheme: +1 for the correct answer and explanation

The output for the second printf is Ans: oellh
This values of the array do get swapped this time because the array address is passed to the function, and therefore the values of the original array get changed.

Marking scheme: +1 for the correct answer, +1 for the correct explanation

The output for the third printf Ans: oellh
This time the actual addresses of the characters in the array are passed to the function, and therefore the values of the original array get changed as well.

Marking scheme: +1 for the correct answer, +1 for the correct explanation

Invoking swap3(str, str + 4) is effectively the same as invoking swap3 with the addresses of the characters in the array, and therefore the output will be

Ans: oellh

Marking scheme: +1 for the correct answer and explanation
