1. François Viète (1540–1603) proposes the following formula for the computation of π:

$$\pi = 2\left[\left(\frac{2}{\sqrt{2}}\right)\left(\frac{2}{\sqrt{2+\sqrt{2}}}\right)\left(\frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}}\right)\left(\frac{2}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}}\right)\cdots\right].$$

Let $d_n = \sqrt{2+\sqrt{2+\sqrt{2+\cdots\sqrt{2}}}}$ with n occurrences of 2 on the right side. For $n = 0, 1, 2, 3, \ldots$, the n-th approximation of π is given by

$$\pi_n = 2\left[\left(\frac{2}{d_1}\right)\left(\frac{2}{d_2}\right)\cdots\left(\frac{2}{d_n}\right)\right].$$

We have $\pi = \lim_{n\to\infty} \pi_n$. The denominator d_n is calculated as $d_n = \sqrt{2+d_{n-1}}$ for $n \geqslant 2$, and $d_1 = \sqrt{2}$. The following C program implements this idea. The loop in the program stops when two successive approximations differ by a very small value, that is, $\pi_n - \pi_{n-1} < \epsilon$, where ϵ is a pre-defined error limit (like 10^{-15}). Fill out the missing parts of the following C code. Use the math library call **`sqrt()`**. Use no other facility provided by the math library. Do not use arrays. Do not introduce new variables.

```
#include <stdio.h>

#include _____________________

#define ERROR_LIMIT 1e-15

main ()
{
   double d;      /* d stores the denominator d_n */
   double pi;     /* pi stores the approximation for π */
   double nextpi; /* next approximation for π */
   double error;  /* difference of two consecutive approximations of π */

   /* Start with the approximation π_1 for pi. Notice that π_0 = 2. */

   d = _________________ ; pi = _________________ ; error = _________________ ;

   /* Iterate until two consecutive approximations differ by a small value */

   while ( ____________________________________ ) {   /* Condition on error */

      d = ____________________________ ;   /* Compute next value of denominator */

      nextpi = ____________________________ ;    /* Compute next approximation */

      error = ____________________________ ;   /* Difference of approximations */

      pi = ____________________________ ;    /* Prepare for the next iteration */
   }
   printf("Approximate value of pi = %lf\n", pi);
}
```

2. Let f and g be two polynomials in x. We want to compute their product $h = fg$. Suppose that each of f and g has n terms. Write $f = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0 = x^m f_1 + f_0$, where $m = n/2$ (assume n is even), and where the half polynomials $f_1 = a_{2m-1}x^{m-1} + a_{2m-2}x^{m-2} + \cdots + a_{m+1}x + a_m$ and $f_0 = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \cdots + a_1x + a_0$ have m terms each. Likewise, write $g = x^m g_1 + g_0$. We have $fg = x^{2m} f_1 g_1 + (f_1 g_0 + f_0 g_1)x^m + f_0 g_0$. Since $f_1 g_0 + f_0 g_1 = (f_1 + f_0)(g_1 + g_0) - f_1 g_1 - f_0 g_0$, we can compute fg by making only three recursive calls on half polynomials ($f_0 g_0$, $f_1 g_1$ and $(f_1 + f_0)(g_1 + g_0)$).

Complete the following recursive C function to implement this multiplication algorithm (known as the *Karatsuba-Ofman* algorithm). A polynomial $f = a_{n-1}x^{n-1} + \cdots + a_1x + a_0$ with n terms is stored in an array of size MAX $\geqslant n$ as follows. Blank cells mean *memory not in use*.

Array element	a_0	a_1	$\cdots$	a_{n-1}			$\cdots$	
Array index	0	1		$n-1$	n	$n+1$		MAX -1

```
void polyMul ( int h[], int f[], int g[], int n )
/* f and g are the input polynomials, h is the output (product) */
/* n is the number of terms (not the degree) in each input polynomial */
{
   int m, i;
   int f1[MAX], f0[MAX], g1[MAX], g0[MAX];    /* Copies of half polynomials */
   int f0g0[MAX], f1g1[MAX];              /* Local storage for f0g0 and f1g1 */
   int f1f0[MAX], g1g0[MAX], f1f0g1g0[MAX];  /* f1+f0, g1+g0, (f1+f0)(g1+g0) */

   if (n == 1) { h[0] = ____________________ ; return; }  /* Recursion basis */

   /* Pad with zero to make the number of terms even */
   if (n % 2 == 1) { f[n] = g[n] = 0 ; ++n; }
   m = n / 2; /* Number of terms in each half polynomial */

   for (i=0; i<m; ++i) {       /* Make local copies of the half polynomials */

      f0[i] = f[i]; g0[i] = g[i]; f1[i] = ____________ ; g1[i] = ____________ ;
   }

   for (i=0; i<m; ++i) {             /* Loop for computing f1+f0 and g1+g0 */

      f1f0[i] = ____________________ ; g1g0[i] = ____________________ ;
   }

   /* Three recursive calls */

   polyMul(f0g0, __________ , __________ , __________ );              /* f0g0 */

   polyMul(f1g1, __________ , __________ , __________ );              /* f1g1 */

   polyMul(f1f0g1g0, __________ , __________ , __________ );  /* (f1+f0)(g1+g0) */

   for (i=0; i<=4*m-2; ++i) h[i] = 0; /* Initialize h to zero */

   /* Add/subtract the products of half polynomials at appropriate places */
   for (i=0; i<=2*m-2; ++i) {

      h[i]     += ____________________________________________ ;

      h[m+i]   += ____________________________________________ ;

      h[2*m+i] += ____________________________________________ ;
   }
}
```

3. In this exercise, we compute the binomial coefficient $\binom{n}{r}$ by repeatedly using the formula $\binom{n}{r} = \frac{n}{r}\binom{n-1}{r-1}$. We compute n/r as a floating-point value. Finally, the accumulated product is rounded to the nearest integer. In both the following parts, you are not allowed to use any math library function.

(a) Fill in the blanks to complete the following C function that takes a floating-point value x as its only argument and returns the rounded value of x. The rounded value of x is the integer nearest to x. When x is mid-way between two consecutive integers, we follow the convention "round half away from zero", that is, round$(2.5) = 3$ and round$(-2.5) = -3$.

```
int roundit ( double x )
{
   int r;          /* The rounded integer to return */
   double fpart;   /* Fractional part */

   /* Store in r the truncated value of |x| */

   _____________________________________________________________

   /* Store in fpart the fractional part of |x| */

   _____________________________________________________________

   /* Modify r based conditionally upon fpart */

   if (fpart >= 0.5) _____________________ ;

   /* Return r after sign adjustment */

   return ____________________________________________________ ;
}
```

(b) Complete the following C function to compute $\binom{n}{r}$. The function starts by initializing an empty product, and subsequently multiplies the product by $\frac{n}{r}$, $\frac{n-1}{r-1}$, $\frac{n-2}{r-2}$, and so on, in a loop, until the denominator reduces to 0. After the loop terminates, the product is almost ready to be returned (since $\binom{n-r}{0} = 1$). But since floating-point calculations are used to compute the product, there may be some (small) error due to floating-point approximations. Use the function of Part (a) to round the product, and return the rounded value.

```
unsigned int bincoeff ( unsigned int n, unsigned int r )
{
   double prod = ____________ ;   /* Initialize to empty product */

   if (r > n) return ____________ ;

   while ( _____________________ ) /* Condition on r */ {
      /* Multiply prod by the fraction n/r (floating-point division) */

      ____________________________________________________________
      /* Prepare for the next iteration (computation of C(n-1,r-1)) */

      ____________________________________________________________
   }

   /* Return the rounded product. Use the function of Part (a). */

   return ____________________________________________ ;
}
```

4. The 8-bit 2's-complement representation of −65 is:

5. How many floating-point numbers can be represented in the denormalized form (that is, with all exponent bits equal to 0) in the 32-bit IEEE 754 format? (Treat zero as a single denormalized number, that is, +0 = −0.)

6. Consider the following declaration: int (*A)[20]. If A points to the memory location x, which memory location does A+1 point to? Assume that sizeof (int) = 4.

7. Describe what this program prints:

```
#include <stdio.h>
main ()
{
    int r, t, m;
    scanf("%d", &r); /* Enter the last four digits of your roll number as r */
    printf("r = %d\n", r);
    m = 0;
    while (r > 0) {
        t = r % 10;
        if (t > m) m = t;
        r = r / 10;
    }
    printf("m = %d\n", m);
}
```

Write your answer in the space below this line:

r =

m =

Description in one sentence:

8. What is the output of the below program?

```
#include<stdio.h>

int func()
{
      static int i = 3;
      i++;
      return i;
}

main()
{
      int i;
      for(i = 0;i<4;i++)
      {
            printf(" %d", func());
      }
}
```

9. What will the following program print?

```
int main()
{
      char str[30] = "This is LAB test";
```

```
        printf("%d", fun(str));
        return 0;
}

int fun(char *x)
{
        char *ptr = x;
        while (*ptr != '\0') ptr++;
        return (ptr - x);
}
```

10. What will be the output of the following C program?

```
int main()
{
        int a[ ] = {1,2,3,4,5,6,7};
        printf("%d\n",(a+1)[4]);
        return(0);
}
```

11. What will be the output of the following C program?

```
#define reciprocal(x) 1/x
int main(void)
{
        int y = 2, z = 3;
        float w;
        w = (y+z) * reciprocal(y+z);
        printf("%f\n",w);
        return(0);
}
```

12. What will be the output of the following C program?

```
int main()
{
          int i = 43;
          printf("%d\n",printf("%d",i));

          return(0);
}
```

13. What does the following program print?

```
void g ( int A[], int n )
{
    int i;

    for (i = 1; i < n; ++i)
        A[i] += A[i-1];
}
main ()
{
    int A[5] = {2,3,4,5,6};

    g(A,4);
    printf("%d", A[4] - A[3]);
}
```

14. What does f(240) return?

```
int f ( int n )
{
    if (n == 0) return -1;
    if (n % 2) return 0;
    return 1 + f(n+n/2);
}
```

15. Express f(n) as a function of n, where f is defined as follows. Show your calculations:

```
unsigned int f ( unsigned int n )
{
    unsigned int s = 0, i, j;

    for (i=1; i<=n; ++i)
        for (j=i; j<=n; ++j)
            s += i + j;
    return s;
}
```

16. Complete the following program that scans a positive integer n > 2 and prints the largest *proper* divisor m of n. Example: For n = 60, your program should print m = 30. For n = 61, your program should print m = 1.

```
#include <stdio.h>
main()
{
    int n, m;
    printf("Enter an integer n >= 2 :  "); scanf("%d",&n);
```

17. Complete the following program:

A house has n rooms numbered $0, 1, \ldots, n-1$. Consider the $n \times n$ matrix M. The i, j-th entry of M is 1 if there is a door between room i and room j; it is 0 otherwise. Given two rooms u, v, the following function finds out whether there exists a way to go from room u to room v using the doors. The function works as follows. It maintains two arrays `visited[]` and `toExplore[]`. To start with, room u is marked as visited and is also inserted in the array `toExplore[]`. Then we enter a loop. We first check if there are some more rooms to explore. Let i be one such room. We find out all unvisited rooms j sharing doors with room i. We mark all these rooms j as visited and also insert them in the array `toExplore[]`. A good way to handle elements of the array `toExplore[]` is to insert elements at the end (of the current list of rooms to explore) and consider the rooms from beginning to end for further exploration. We maintain two indices `start` and `end`. The rooms yet to be explored lie in the indices `start`, ..., `end` in `toExplore[]`. Complete the following function to solve this connectivity problem.

```
int connected ( int M[MAXDIM][MAXDIM], int n, int u, int v )
{
   int *visited, *toExplore, i, j, start, end;

   /* Allocate memory for n integers to each of visited and toExplore */

   visited   = _____;

   toExplore = _____;
   for (i=0; i<n; ++i) visited[i] = 0; /* Initialize the array visited*/

   visited[u] = _____ ; /* Mark room u as visited*/
   /* Insert room u in the array toExplore*/
   toExplore[0] = u; start = end = 0;
   /* As long as there are more rooms to explore*/

   while ( _____ ) {
      i = toExplore[start]; ++start;
      /* if i is the destination room v, return true*/
      if ( i == v ) return 1;
      /* Check all rooms j sharing doors with room i*/
      for (j=0; j<n; ++j) {
         /* if there is a door between i and j, and j is not visited*/

         if ((_____) && (_____)) {
            /* Mark j as visited */

            _____
            /* Insert j in toExplore[] and adjust the insertion index*/

            _____
         }
      }
   }
   /* Loop ends. Room v could not be reached from room u.*/
   /* Free allocated memory*/

   free(_____); free(_____);
   /* Return failure */

   _____

}
```

18. Complete the following program:

Let $\omega = \sqrt[3]{2}$ be the real cube root of 2. Consider the set

$$A = \{a + b\omega + c\omega^2 \mid a, b, c \text{ are integers}\}$$

of real numbers. It turns out that the set A is closed under addition, subtraction and multiplication, i.e., the sum, difference and product of $a_1 + b_1\omega + c_1\omega^2, a_2 + b_2\omega + c_2\omega^2 \in A$ can be expressed in the form $a + b\omega + c\omega^2$. Clearly, $(a_1 + b_1\omega + c_1\omega^2) \pm (a_2 + b_2\omega + c_2\omega^2) = (a_1 \pm a_2) + (b_1 \pm b_2)\omega + (c_1 \pm c_2)\omega^2$.

For computing the product, first multiply $a_1 + b_1\omega + c_1\omega^2$ and $a_2 + b_2\omega + c_2\omega^2$ and obtain a polynomial in ω of degree 4. Then use the facts $\omega^3 = 2$ and $\omega^4 = 2\omega$ in order to reduce this polynomial of degree 4 back to a polynomial of degree 2. That is, we have:

$$\begin{aligned}
& (a_1 + b_1\omega + c_1\omega^2)(a_2 + b_2\omega + c_2\omega^2) \\
= \; & (a_1a_2) + (a_1b_2 + a_2b_1)\omega + (a_1c_2 + b_1b_2 + a_2c_1)\omega^2 + (b_1c_2 + b_2c_1)\omega^3 + (c_1c_2)\omega^4 \\
= \; & (a_1a_2) + (a_1b_2 + a_2b_1)\omega + (a_1c_2 + b_1b_2 + a_2c_1)\omega^2 + (b_1c_2 + b_2c_1) \times 2 + (c_1c_2) \times (2\omega) \\
= \; & (a_1a_2 + 2b_1c_2 + 2b_2c_1) + (a_1b_2 + a_2b_1 + 2c_1c_2)\omega + (a_1c_2 + b_1b_2 + a_2c_1)\omega^2.
\end{aligned}$$

Represent an element of A by a structure of three integers:

```
typedef struct {
   int a,b,c; /* Represents a + bω + cω² ∈ A */
} cubicNumber;
```

Complete the following function that takes two cubic numbers **x1**, **x2** as arguments and returns their product.

```
cubicNumber cubicProd ( cubicNumber x1, cubicNumber x2 )
{
   _____________________________________ /* local variable */

   _______ = _____________________________________________________;

   _______ = _____________________________________________________;

   _______ = _____________________________________________________;

   ___________________________________
}
```

19. The following is the skeleton of a C program that computes the number of each numeral in a string. Fill in the blanks with appropriate C constructs.

```
#include <stdio.h>

#define base 10

main ()  /* This program outputs the numbers of 0's,1's, ..., 9's in an
            input string ending in $ */
{
   char b;
   int i, a[base];

   /* Initialize array elements to zero */

   for ( _____ ; _____ ; _____ )
      a[i] = 0;

   printf("Input numeric characters ending with $\n" );

   scanf("%c", &b); /* Scan next character */

   /* Execute the loop as long as $ is not scanned */

   while ( _____ ) {
      printf("Processing the digit %c\n", b);

      /* Increment the count for the new digit */

      a[_____
      scanf("%c", &b); /* Scan next character */
   }

   for ( i=0 ; i<=9 ; i=i+1 )

      printf("There are %___ %___'s\n", a[i], i);
}
```

Note: You don't have to know the exact ASCII values for the characters $0, 1, 2, \ldots$. It is sufficient to know only that the ASCII representations of $0, 1, 2, \ldots$ are consecutive. Not all blanks carry equal marks. Evaluation will depend on the overall correctness.