CS11001 Programming and Data Structures, Autumn 2010

Mid-semester Test

Maximum marks: 60 | September 2010 | Total time: 2 hours

Roll no: 10FB1331 **Name:** Foolan Barik **Section:** @

Write your answers in the question paper itself. Be brief and precise. Answer all questions. Do your rough work on supplementary sheets. Write your final answers in the spaces provided. Not all blanks carry equal marks. Evaluation will depend on the overall correctness.

(To be filled in by the examiners)

Question No	1	2	3	4	Total
Marks					

1. For each of the following parts, mark the correct answer. Mark like this: **(B)** **(16)**

(a) The program counter in the CPU is used

(A) to store an operand,
(B) to store the address of an operand,
(C) to store an instruction,
(D) to store the address of an instruction.

(b) The 8-bit 2's-complement representation of -65 is:

(A) 10111110
(B) 10111111
(C) 11000001
(D) 11000010

(c) 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, $+0 = -0$.)

(A) $2^{23} - 1$ **(B)** 2^{23} **(C)** $2^{23} + 1$ **(D)** $2^{24} - 1$

(d) Which of the following can be a valid name of a C variable?

(A) `default`
(B) `_default`
(C) `-default`
(D) `123default`

(e) What is the value of `x` after the following statements are executed?

```
int m = 5, n = 5, x;
char p = 'p', q = 'q';
x = !((m>=n)||(m<=n)&&(p>q));
```

(A) 0 **(B)** 1 **(C)** -1 **(D)** Any non-zero value

(f) What is the value of `s` after the termination of the following loop?

```
int n = 100, s;
for (s = 2; n > 2; --n) {
    s += 2; n -= 2;
}
```

(A) 64 **(B)** 66 **(C)** 68 **(D)** 98

(g) What does `f(240)` return, if `f()` is defined as follows?

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

(A) 4 **(B)** 5 **(C)** 16 **(D)** 361

(h) 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]);
}
```

(A) 1 **(B)** 5 **(C)** 6 **(D)** -8

2. 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. **(18)**

```
#include <stdio.h>

#include ________<math.h>________

#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 = ___sqrt(2.0)___ ; pi = ___4.0 / d___ ; error = ___pi - 2.0___ ;

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

   while ( ________error >= ERROR_LIMIT________ ) {   /* Condition on error */

      d = ____sqrt(2.0 + d)____ ;   /* Compute next value of denominator */

      nextpi = ____pi * 2.0 / d____ ;     /* Compute next approximation */

      error = ____nextpi - pi____ ;    /* Difference of approximations */

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

Roll no: 10FB1331 **Name:** Foolan Barik **Section:** @

3. You are given an array A of n integers. It is given that the elements of A satisfy the following inequalities

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

for some (unknown) index m in the range $1 \leqslant m \leqslant n-2$. Let us call such an array a *hill-valued* array. The sequence $A[0], A[1], \ldots, A[m-1], A[m]$ is called the ascending part of the hill, and the remaining part $A[m], A[m+1], \ldots, A[n-1]$ is called the descending part of the hill. The element $A[m]$ is the peak of the hill and is the largest element in the array.

Your task is to locate the peak (that is, $A[m]$) in the hill-valued array A. Initially, start searching in the entire array. Subsequently, in each iteration of the loop, reduce the search space to a subarray of half the size of the subarray in the previous iteration. In order to achieve that, compute the middle index in the current search space. Compare the element at this index with its two neighbors. If you are at the peak, break the loop, else adjust the search space appropriately. (This procedure is similar to binary search in a sorted array.) Complete the following C program that implements this idea. Do not use new variables. **(16)**

```
#include <stdio.h>

#define MAX 1000

main () {
   int A[MAX], i, n, L, R, M, found;

   printf("Enter a hill-valued array.\n");
   printf("Number of elements = "); scanf("%d", &n);
   for (i=0; i<n; ++i) { printf("A[%d] = ", i); scanf("%d", &A[i]); }

   /* Initialize the left and right boundaries L and R of the search space
      so as to encompass the entire array A */

   L = ____0____ ; R = ____n - 1____ ;
   found = 0; /* Initialize flag to false */
   while (!found) { /* 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])____ ) {
         /* if the top of the hill is located, */
         /* set the flag to break the loop before the next iteration */
         found = 1;

      } else if ( ____(A[M-1]<A[M]) && (A[M]<A[M+1])____ ) {
         /* if the middle index is in the ascending part of the hill, */
         /* discard a suitable half of the search space */

         ____L = M____ ;
      } else {
         /* if the middle index is in the descending part of the hill, */
         /* discard a suitable half of the search space */

         ____R = M____ ;
      }
   }
   printf("Maximum = %d\n", ____A[M]____ );
}
```

4. 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*. **(10)**

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] = ____f[0] * g[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] = ___f[m+i]___ ; g1[i] = ___g[m+i]___ ;
   }

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

      f1f0[i] = ____f1[i] + f0[i]____ ; g1g0[i] = ____g1[i] + g0[i]____ ;
   }

   /* Three recursive calls */

   polyMul(f0g0, ___f0___ , ___g0___ , ___m___ );                 /* f0g0 */

   polyMul(f1g1, ___f1___ , ___g1___ , ___m___ );                 /* f1g1 */

   polyMul(f1f0g1g0, ___f1f0___ , ___g1g0___ , ___m___ );  /* (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]     += ____________________f0g0[i]____________________ ;

      h[m+i]   += ____f1f0g1g0[i] - f0g0[i] - f1g1[i]____ ;

      h[2*m+i] += ____________________f1g1[i]____________________ ;
   }
}
```