CS11001 Programming and Data Structures, Autumn 2010

End-semester Test

Maximum marks: 100 November 2010 Total time: 3 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	5	6	Total
Marks							

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

(a) What is the output of the following program?

```
main()
{
   char str[30] = "This is PDS test";
   printf("%d", fun(str));
}

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

(A) 4 **(B)** 16 **(C)** 17 **(D)** 30

(b) What is the output of the following program?

```
main()
{
   int array[10] = { 20, 18, 16, 14, 12, 10, 8, 6, 4, 2 };
   int *ptr;
   ptr = array;
   printf("%d,%d", *ptr + 2, *(ptr + 2));
}
```

(A) 16,16 **(B)** 20,16 **(C)** 22,16 **(D)** 22,22

(c) 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)` $= 4$.

(A) $x + 80$ **(B)** $x + 20$ **(C)** $x + 4$ **(D)** Does not depend on x

(d) Each of the following choices declares two variables **A** and **B**. Identify the pair in which **A** and **B** do not have compatible organizations in memory.

(A) `int *A, B[MAX];` **(B)** `int *A[MAX], **B;` **(C)** `int (*A)[MAX], **B;`
(D) `int (*A)[MAX], B[MAX][MAX];`

(e) In a machine with 32-bit integers and 32-bit addresses, what is **`sizeof(node)`**, where **`node`** is defined as follows:

```
typedef struct _tag {
   int a[8];
   double b[16];
   char c[32];
   struct _tag *next;
} node;
```

(A) 384 **(B)** 196 **(C)** 132 **(D)** 60

(f) Consider the following recursive function:

```
unsigned int f ( unsigned int n )
{
   if (n <= 2) return 1;
   return f(n-3) + f(n-1);
}
```

What is the maximum height to which the recursion stack grows when the outermost call is **`f(10)`**? Assume that the stack is empty just before this outermost call.

(A) 5 **(B)** 9 **(C)** 13 **(D)** 32

(g) The total number of comparisons needed for bubble sorting an array of size n is:

(A) $O(n^2)$ **(B)** $O(n \log n)$ **(C)** $O(n)$ **(D)** None of the above

---

(h) What is the output of the following program?

```
struct node {
   int cval;
   struct node *next;
}

main()
{
   struct node N1, N2, N3;
   N1.cval = 1; N2.cval = 10; N3.cval = 100;
   N1.next = &N2; N2.next = &N3; N3.next = &N1;
   printf("%d,%d", N2.next -> cval, N2.next -> next -> cval);
}
```

(A) 1,10 **(B)** 1,100 **(C)** 10,100 **(D)** 100,1

---

2. Consider the following way to compute the maximum in an array A of size n. First, divide the array in two equal (or almost equal) halves. Then, recursively compute the maximums in the two halves. Finally, return the larger of these two recursively computed maximum values.

(a) Complete the following function which uses the above idea for finding the maximum in an array. **(6)**

```
int max ( int A[], int n )
{
   int m1, m2;

   if (n == 1) return ____A[0]____;
   /* Make two recursive calls. Do not assume that n is even. */

   m1 = max( ____A____ , ____n/2____ ); /* left half */

   m2 = max( ____A + n/2____ , ____n - n/2____ ); /* right half */

   return ____((m1 > m2) ? m1 : m2)____ ;
}
```

(b) How is the function called from the **`main()`** function on an array A of size n? **(1)**

`max(A,n);`

(c) Deduce the running time of **`max()`** on an array of size n. For simplicity, you may assume that n is a power of 2, that is, $n = 2^t$ for some integer $t \geqslant 0$. Express the running time in the Big-Oh notation. **(7)**

Solution Let $T(n)$ denote the running time of **`max()`** on an array of size n. We then have

$$\begin{aligned} T(1) &= c, \\ T(n) &= 2T(n/2) + d \text{ for even } n \geqslant 2. \end{aligned}$$

Here, c, d are positive constant values. If $n = 2^t$, repeated unfolding of the recurrence gives

$$\begin{aligned} T(n) &= T(2^t) \\ &= 2T(2^{t-1}) + d \\ &= 2(2T(2^{t-2}) + d) + d \\ &= 2^2T(2^{t-2}) + (2+1)d \\ &= 2^2(2T(2^{t-3}) + d) + (2+1)d \\ &= 2^3T(2^{t-3}) + (2^2+2+1)d \\ &= \cdots \\ &= 2^tT(1) + (2^{t-1} + 2^{t-2} + \cdots + 2^2 + 2 + 1)d \\ &= 2^tc + (2^t - 1)d \\ &= (c+d)n - d. \end{aligned}$$

It follows that the running time of **`max(n)`** is $O(n)$.

3. A *convex polygon* is a simple polygon in which all interior angles are less that 180^0. In this exercise, you are required to compute the area of a convex polygon.

(a) Define a data type **point** consisting of double-precision floating-point fields x and y. **(2)**

```
typedef struct { double x, y; } point ;
```

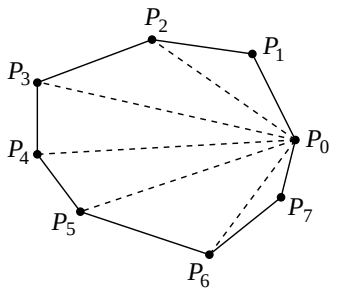
Next, define a static array of **MAX** structures of type **point** as the data type **cpolygon**. Note that a convex polygon is to be stored in an array of points as a counterclockwise sequence of the vertices of the polygon. **(2)**

```
#define MAX 1000

typedef point cpolygon [MAX] ;
```

(b) Complete the following function that accepts a convex polygon and three indices i, j, k representing vertices P_i, P_j, P_k of the polygon stored in the input array. The function computes and returns the area of the triangle $P_iP_jP_k$ using the formula $\sqrt{s(s-a)(s-b)(s-c)}$, where a, b, c are the lengths of the three sides of the triangle and $s = (a+b+c)/2$ is the semi-perimeter of the triangle. You may use math library functions. **(6)**

```
double triangleArea ( cpolygon P, int i, int j, int k )
{
   double a, b, c, s; /* Do not use any other variable */

   /* Compute the lengths of the sides */

   a = sqrt(pow(P[i].x - P[j].x, 2) + pow(P[i].y - P[j].y, 2)) ;

   b = sqrt(pow(P[j].x - P[k].x, 2) + pow(P[j].y - P[k].y, 2)) ;

   c = sqrt(pow(P[k].x - P[i].x, 2) + pow(P[k].y - P[i].y, 2)) ;
   /* Compute the semi-perimeter */

   s = (a + b + c) / 2.0 ;
   /* Return the area */

   return (sqrt(s*(s-a)*(s-b)*(s-c))) ;
}
```

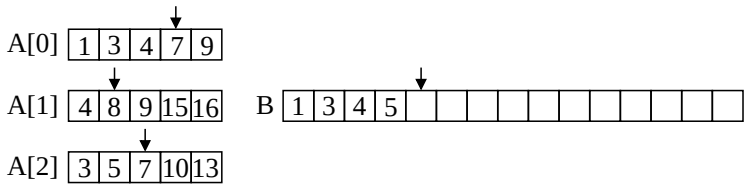
(c) In order to compute the area of a convex polygon, we first triangulate the polygon as demonstrated in the adjacent figure. The area of each triangle is computed by the function of Part (b). The sum of these areas is returned as the area of the input polygon. **(5)**




```
double polygonArea ( cpolygon P, int n )
{
   int i;

   double area = 0.0 ;

   for (i= 1 ; i<= n-2 ; ++i) area += triangleArea(P,0,i,i+1) ;
   return area;
}
```

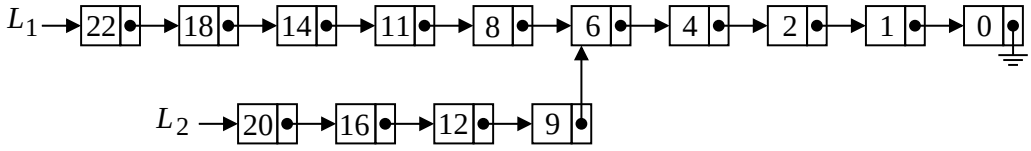
4. You are given an $m \times n$ array A of integers, each row of which is a sorted list of size n. Your task is to merge the m sorted lists and store the merged list in a one-dimensional array B. It is given that each row does not contain repetition(s) of integers, that is, the n integers in each sorted list are distinct from one another. However, integers may be repeated in different rows. During the merging step, you must remove all these repetitions.

Complete the following function to achieve this task. The function uses an array of m indices, where the i-th index is used for reading from the i-th row ($0 \leqslant i \leqslant m-1$). The function starts by initializing each of these read indices to point to the beginning of the corresponding row. Subsequently, inside a loop, it computes the minimum of the m elements pointed to by these indices. The minimum is then written to the output array. Note, however, that during the computation of this minimum, we do not need to consider those rows all of whose elements have already been written to the output array B. Finally, for all rows containing this minimum element at the current read index positions, the index values are incremented. The function is supposed to return the total number of elements written to the output array B. An example is given below. **(15)**




```
#define INFINITY 123456789

int merge ( int B[], int A[][MAX], int m, int n )
/* A is the input two-dimensional array of size m × n.
   B is the output array whose size is to be returned. */
{
   int index[MAX], i, k, min; /* Do not use other variables */

   for (i=0; i<     m     ; ++i) index[i] =     0     ;

   k =     0     ; /* k is for writing to B[] */

   while (1) { /* Let us decide to return inside this loop */
      min = INFINITY; /* Initialize min to a suitably large value */
      /* Write a loop for computing the minimum */

      for ( i=0; i<m; ++i ) {

         if ((index[i] < n) && (A[i][index[i]] < min))

            min =  A[i][index[i]]  ;
      }
      /* If all input arrays are fully processed, return the size of B */

      if (min == INFINITY) return k;
      /* Otherwise, write the computed minimum to B */

      B[k++] = min;
      /* Advance all relevant indices */

      for ( i=0; i<m; ++i ) {

         if ((index[i] < n) && (A[i][index[i]] == min))

            ++index[i] ;
      }
   }
}
```

5. In this exercise, we deal with linked lists. First, consider the list headed by the pointer L_1 (see the figure below). There is no dummy node at the beginning. The first element in the list is 22. Subsequently, if a node stores the integer n, the next node stores the integer $n - \lfloor\sqrt{n}\rfloor$. This means that we have a list of monotonically decreasing integers. The list terminates after the node containing the value 0.



Define a node in the list in the usual way:

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

(a) Write a function to create a single list like L_1 as explained above. The list starts with a supplied integer value n, and subsequently uses the above formula for the subsequent nodes. **(6)**

```
node *genSeq1 ( int n )
{
   node *L, *p;

   /* Create the fist node to store n */

   L = ____(node *)malloc(sizeof(node))____ ;

   ____L -> data = n;____
   /* Initialize the running pointer p for subsequent insertions */

   p = ____L____ ;

   while ( ____n____ ) { /* As long as n is not reduced to zero */
      /* Compute the next value of n */

      n -= ____(int)sqrt((double)n)____ ;
      /* Allocate memory */

      ____p -> next = (node *)malloc(sizeof(node));____
      /* Store n in the new node, and advance */

      ____p = p -> next; p -> data = n;____
   }

   ____p -> next = NULL;____ /* Terminate the list */
   return L; /* Return the header */
}
```

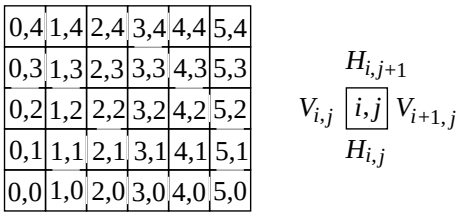
(b) Now, we create another list to be headed by L_2 (see the above figure). This second list starts with another value (like 20), and contains nodes storing integer values satisfying the same formula used in the first list. After some iterations, two lists must encounter a common value (6 in the example of the above figure). From this node onwards, the second list follows the same nodes and links as the first list. Complete the following C function to create the second list. The header L_1 to the first list is passed to this function. Also, the starting value n for the second list is passed. **(8)**

```
node *genSeq2 ( node *L1, int n )
{
   node *L2, *p1, *p2;
   /* Skip values in the first list larger than n */

   p1 = L1; while ( ____p1 -> data > n____ ) p1 = p1 -> next;
   /* If n is already present in the first list */

   if ( ____p1 -> data == n____) return ____p1____ ;
   /* Create the fist node in the second list to store n */

   L2 = ____(node *)malloc(sizeof(node))____ ;

   ____L2 -> data = n____;
   /* Initialize the running pointer p for subsequent insertions */

   p2 = ____L2____ ;
   while (1) { /* Let us decide to return inside the loop */

      n -= ____(int)sqrt((double)n)____ ; /* Next value of n */
      /* p1 skips all values in the first list, larger than the current n */

      while ____((p1 != NULL) && (p1 -> data > n))____ p1 = p1 -> next;

      if ( ____p1 -> data == n____ ) { /* n found in first list */

         ____p2 -> next = p1____ ; /* Adjust the second list */

         return ____L2____ ; /* Return header to second list */
      }
      /* n not found in first list, so create a new node in second list */

      ____p2 -> next = (node *)malloc(sizeof(node));____

      ____p2 = p2 -> next; p2 -> data = n;____
   }
}
```

(c) Complete the following function that, given the headers L_1 and L_2 as input, returns a pointer to the first common node in these two lists. **(6)**

```
node *getIntersection ( node *L1, node *L2 )
{
   node *p1, *p2;

   p1 = ____L1____ ; p2 = ____L2____ ; /* Initialize pointers */
   while (1) { /* Return inside the loop */
      /* If the common node is located, return an appropriate pointer */

      if ( ____p1 -> data == p2 -> data____ ) return ____p1____ ;
      /* else if p1 points to a larger integer than p2 */

      else if ( ____p1 -> data > p2 -> data____ ) ____p1 = p1 -> next____ ;

      else ____p2 = p2 -> next____ ;
   }
}
```

6. You have an $m \times n$ maze of rooms. Each adjacent pair of rooms has a door that allows passage between the rooms. At some point of time some of the doors are locked, the rest are open. A mouse sits at room number (s, t), and there is fabulous food for the mouse at room number (u, v). Your task is to determine whether there exists a route for the mouse from room (s, t) to room (u, v) through the open doors. The idea is to start a search at room no (s, t), then investigate adjacent rooms $(s_1, t_1), \ldots, (s_k, t_k)$ that can be reached from (s, t), and then those adjacent rooms that can be reached from each (s_i, t_i), and so on.



In order to set up the notations about indices, look at the above figure. The rooms are numbered (i, j), where i grows horizontally (along the x direction), and j grows in the vertical direction (along the y axis). The four walls of the (i, j)-th room are numbered as shown to the right of the maze. If a horizontal or vertical wall has an open door, we indicate this by the value 1; otherwise, we use the value 0. That is, $H_{i,j} = 1$ if the horizontal door connecting the rooms $(i, j-1)$ and (i, j) is open; $H_{i,j} = 0$ otherwise. Similarly, $V_{i,j}$ is 1 or 0 depending upon whether the vertical door between the rooms $(i-1, j)$ and (i, j) is open or not. This numbering scheme also applies to the walls of the boundary of the maze. However, we assume that the mouse cannot go out of the house, that is, all the walls on the boundary are closed. H is available as an $m \times (n+1)$ array, whereas V is available as an $(m+1) \times n$ array. In the example shown above, Room $(3, 1)$ is reachable from Room $(2, 4)$, but Room $(1, 2)$ is not reachable from Room $(2, 4)$.

We use a stack to implement the search (from (s, t) to (u, v) given the arrays H and V). Since there is no need to revisit a room during the search, we maintain an $m \times n$ array of flags in order to keep track of the rooms that are visited—the value 1 means "visited", and 0 means "not visited so far". The stack contains a list of rooms, that is, it is capable of storing pairs of indices (i, j). The stack ADT is supplied as follows.

```
S = init();                                   /* Create an empty stack */
S = push(S,i,j);              /* Push the pair (i,j) to the top of the stack S */
S = pop(S);              /* Pop an element (a pair) from the top of the stack */
(i,j) = top(S);                        /* Return the top (i,j) of the stack S */
isEmpty(S);              /* Returns true if and only if the stack S is empty */
```

(a) Fill out the following **main()** function that pushes the source room (s, t) to an initially empty stack, and subsequently calls the search function with appropriate arguments. **(5)**

```
main ()
{
   stack S;
   int m, n, s, t, u, v, H[MAX][MAX], V[MAX][MAX], visited[MAX][MAX], status;

   /* Assume that m, n, s, t, u, v, H[][], V[][] and visited[][] are
      appropriately initialized here. You do not have to write these. */
   . . .

   S = _____init()_____ ; /* Initialize the stack S */

   S = push( S, __s__ , __t__ ); /* Push source room to stack */

   visited[ __s__ ][ __t__ ] = 1; /* Mark source room as visited */
   status = search(m,n,s,t,u,v,H,V,visited,S); /* Call the search function */
   printf("Search %s\n", (status == 1) ? "successful" : "unsuccessful");
}
```

(b) Complete the search function whose skeleton is provided below. The function returns 1 if the destination node is ever reached. Otherwise, it returns 0. **(15)**

```
int search ( int m, int n, int s, int t, int u, int v,
             int H[][MAX], int V[][MAX], int visited[][MAX], stack S )
   /* m × n is the size of the maze, (s,t) is the source node,
      (u,v) is the destination node, H[][] and V[][] are door arrays,
      visited[][] is the array to store which nodes are visited so far,
      and S is the stack to be used in the search. */
{
   pair room; /* pair is a structure of two integer values x and y */
   int i, j;

   /* So long as the stack is not empty */

   while ( ____!isEmpty(S)____ ) {

      room = ____top(S)____ ; /* Read the top element from the stack */
      i = room.x; j = room.y ; /* Retrieve x and y coordinates of room */
      /* Delete the top from the stack */

      ____S = pop(S);____
      /* If (i,j) is the destination node, return success */

      if ( ____(i == u) && (j == v)____ ) return ____1____ ;
      /* Otherwise, look at the four adjacent rooms one by one */
      /* Left room: If left door is open and left room is not yet visited */

      if ( ____(V[i][j]) && (!visited[i-1][j])____ ) {
         /* Push left room to the stack and mark left room as visited */

         S = push(S, ____i-1____ , ____j____ );

         visited[ ____i-1____ ][ ____j____ ] = ____1____ ;
      }
      /* Analogously process the adjacent room to the right */


                if ((V[i+1][j]) && (!visited[i+1][j])) {
                   S = push(S,i+1,j); visited[i+1][j] = 1;
                }
      /* Process the adjacent room at the bottom */


                if ((H[i][j]) && (!visited[i][j-1])) {
                   S = push(S,i,j-1); visited[i][j-1] = 1;
                }
      /* Process the adjacent room at the top */


                if ((H[i][j+1]) && (!visited[i][j+1])) {
                   S = push(S,i,j+1); visited[i][j+1] = 1;
                }
   }

   return ____0____ ;
}
```