# CS13002 Programming and Data Structures, Spring 2006

## End-semester examination

Total marks: 60 | April 26, 2006 | Total time: 3 hours

---

**Roll no:** ____________________ **Section:** __________

**Name:** ________________________________________________

- *This question paper consists of nine pages. Do not write anything on this front page except your name, roll number and section.*
- *Answer all questions.*
- *Write your answers on the question paper itself. Your final answers must fit in the respective spaces provided. Strictly avoid untidiness or cancellations on the question-cum-answer paper.*
- *Do your rough work on the rough sheets provided. The rough work must be submitted, but will not be evaluated. Only answers in the question-cum-answer paper will be evaluated.*
- *Not all blanks carry equal marks in Questions 3–6. 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, circle the letter corresponding to the correct answer. **(1×10)**

**(a)** What string is printed by the call `strFunc("PDS 2006")`?

```
void strFunc ( char A[] )
{
   int d = 'a'-'A', i = 0;

   while (A[i]) {
      if ((A[i] >= 'A') && (A[i] <= 'Z')) A[i] += d;
      ++i;
   }
   printf("%s", A);
}
```

**(A)** `pds 2006` **(B)** `PDS 2006` **(C)** `pds` **(D)** `PDS`

**(b)** What integer value is printed by the following program?

```
#include <stdio.h>

int *what ( int *p )
{
   ++p; *p = 10; ++p;
   return p;
}

int main ()
{
   int A[] = {1,2,3,4,5}, *p;

   p = what(A);
   printf("%d\n", p[0]);
}
```

**(A)** 1 **(B)** 2 **(C)** 3 **(D)** 10

**(c)** What is the listing of the elements of the matrix $\begin{pmatrix} 1 & 2 \\ 7 & 8 \\ 4 & 5 \end{pmatrix}$ in the row-major order?

**(A)** 1,2,4,5,7,8 **(B)** 1,2,7,8,4,5 **(C)** 1,2,8,7,4,5 **(D)** 1,7,4,2,8,5

**(d)** Which of the four assignments is *not* legal after the following declaration?

```
int *A, *B, C[10];
```

**(A)** `A = B;` **(B)** `B = A+10;` **(C)** `B = C+20;` **(D)** `C = B+30;`

**(e)** Assume that my machine supports 32-bit addresses. The structure `myStruct` is declared as follows. What value is returned by `sizeof(struct myStruct)` on my machine?

```
struct myStruct {
   long A;
   char B[10];
   float C[10];
   struct myStruct *D;
}
```

**(A)** 16 **(B)** 22 **(C)** 58 **(D)** 88

Question 1 is continued to the next page. . .

**(f)** How many bytes are allocated to the pointer **p** after the following call?

```
#define MAXSIZE 100
p = (long int *)malloc(MAXSIZE * sizeof(long int));
```

**(A)** 4 **(B)** 100 **(C)** 400 **(D)** 1600

**(g)** Consider the following sequence of push and pop operations on an initially empty stack $S$.

```
S = push(S,1);
S = pop(S);
S = push(S,2);
S = push(S,3);
S = pop(S);
S = push(S,4);
S = pop(S);
S = pop(S);
```

Which of the following is the correct order in which elements are popped?

**(A)** 1,2,3,4 **(B)** 1,3,2,4 **(C)** 1,3,4,2 **(D)** 4,3,2,1

**(h)** Consider the three functions $f(n) = n^3 + n + 1$, $g(n) = 56n^2 + 78$, and $h(n) = 1.23^n$. Which of the following statements is *not* true?

**(A)** $f(n) = O(h(n))$ **(B)** $g(n) = O(f(n))$ **(C)** $g(n) = O(h(n))$ **(D)** $h(n) = O(f(n))$

**(i)** The call **recFunc("programming")** is made, where **recFunc()** is defined as follows.

```
void recFunc ( char A[] )
{
   int t;

   t = strlen(A);
   if (t == 0) return;
   else if (t % 2 == 0) recFunc(&A[t/2]);
   else recFunc(&A[1]);
}
```

How many times is **recFunc()** called? Include the outermost call of **recFunc("programming")** in the count.

**(A)** 11 **(B)** 7 **(C)** 6 **(D)** 2

**(j)** Suppose that an array **A** of size $n \geqslant 1$ is passed to the following function?

```
void someFunc ( int A[] , int n )
{
   int i,j,k;

   for (i=0; i<n; ++i) {
      for (j=1; j<=i; ++j) A[j] -= A[j-1];
      for (k=j; k<n/2; ++k) A[k] += A[k+1];
   }
}
```

What is the running time of the above function?

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

**2.** Consider the following recursive function in two non-negative integer arguments $m, n$.

```
unsigned int A ( unsigned int m , unsigned int n )
{
   if (m == 0) return n+1;
   if (n == 0) return A(m-1,1);
   return A(m-1,A(m,n-1));
}
```

Denote by $A(m, n)$ the value returned by this function. $A(m, n)$ is called the *Ackermann function*. We have $A(0, n) = n + 1$ for all $n \geqslant 0$. Moreover,

$$\begin{aligned} A(1,n) &= A(0, A(1, n-1)) = 1 + A(1, n-1) = 1 + [1 + A(1, n-2)] = 2 + A(1, n-2) \\ &= \cdots = n + A(1,0) = n + A(0,1) = n + 2 \text{ for all } n \geqslant 0. \end{aligned}$$

Derive closed-form mathematical formulas (as functions of $n$) for the following special cases. Show the details of your derivations.

**(a)** $A(2, n) =$ **(5)**

**(b)** $A(3, n) =$ **(5)**

**3.** You are given an $r \times c$ matrix **A**. Your task is to compute the transpose of **A** and store the transpose in **A** itself, without using an additional matrix.

The computation of the transpose is easy if **A** is a square matrix, i.e., if $r = c$. So assume that $r \neq c$. Let $D = \max(r, c)$. We may treat **A** as a $D \times D$ matrix with garbage (possibly uninitialized) values stored in some entries. Transposing this $D \times D$ matrix solves the problem, but involves some unwanted work. For example, if $r = 10$ and $c = 100$, then the matrix contains 1000 elements. Here $D = 100$. Transposing a $D \times D$ matrix means consideration of $D^2 = 10,000$ elements, 90% of which are garbage values.

We avoid this problem as follows. Let $d = \min(r, c)$. We first transpose the initial $d \times d$ part of the matrix. Next we look at the remaining elements and relocate them to appropriate places. Complete the following function that implements this algorithm. **(10)**

```
#define MAXSIZE 100

void transpose ( int A[MAXSIZE][MAXSIZE] , int r , int c )
/* In-place transposition of the r x c matrix A */
{
   int i,j; /* i runs over row indices, j over column indices */
   int d;   /* d is the minimum dimension */
   int t;   /* temporary variable used for swapping */

   /* Assign to d the minimum of the row and column dimensions of A */

   ________________________________________________________________

   /* First transpose the initial d x d part */

   for (i = 0; i < ________________; ++i)

      for (j = 0; j < ________________; ++j) {
         /* Swap the i,j-th and the j,i-th elements */

         ________________________________________________________

      }

   if (r < c) {

      for (i = ________________; i < ________________; ++i)

         for (j = ________________; j < ________________; ++j)
            /* Relocate the i,j-th element to the j,i-th location */

            ____________________________________________________

   } else if (r > c) {

      for (i = ________________; i < ________________; ++i)

         for (j = ________________; j < ________________; ++j)
            /* Relocate the i,j-th element to the j,i-th location */

            ____________________________________________________

   }
}
```

**4.** For a pair of real numbers $a, b$ with $a < b$, the interval $[a, b]$ is defined as the set of all real numbers between $a$ and $b$, i.e., $[a, b] = \{x \mid x \geqslant a \text{ and } x \leqslant b\}$. Represent an interval by the following structure:

```
typedef struct { float a, b; } interval;
```

We are given an array $A$ of $n$ intervals $[a_i, b_i]$ for $i = 0, 1, \ldots, n-1$. We look at the union of all these intervals, namely, at the set $S = \bigcup_{i=0}^{n-1} [a_i, b_i]$. The given intervals may be overlapping. Our task is to write $S$ as the union of non-overlapping intervals, i.e., as an array $B$ of $m$ non-overlapping intervals $[c_j, d_j]$ for $j = 0, 1, \ldots, m-1$ (for some $m \leqslant n$). As an example, consider the ten intervals:

$$[3, 5], [8, 12], [10, 19], [11, 18], [19, 28], [23, 29], [27, 31], [33, 42], [33, 40], [35, 38].$$

The corresponding array of non-overlapping intervals is as follows. We have $m = 3$ in this case.

$$[3, 5], [8, 31], [33, 42].$$

In order to solve this problem, we sort the input intervals with respect to their left end-points. Then we enter a loop. We store the next unprocessed interval $[a_i, b_i]$ in a temporary structure variable **`t`**. Then we look at the interval $[a_{i+1}, b_{i+1}]$. If $a_{i+1} \leqslant$ **`t.b`**, the interval $[a_{i+1}, b_{i+1}]$ can be merged with **`t`**. This merging step alters the right end-point of **`t`** if and only if the right end-point of $[a_{i+1}, b_{i+1}]$ is strictly bigger than that of **`b`**. Then consider the intervals $[a_{i+2}, b_{i+2}], [a_{i+3}, b_{i+3}], \ldots, [a_{i+l}, b_{i+l}]$ as long as merging can be done. When merging is no longer possible, we write **`t`** in the output array $B$. We then go to the top of the loop, store $[a_{i+l+1}, b_{i+l+1}]$ in **`t`**, and repeat the same procedure, until all of the $n$ input intervals are taken care of.

**(a)** Complete the following function that implements this algorithm. The function should return the number of intervals in the output array. You do not have to write the sorting function **`sortIntervals()`**. **(8)**

```
int mergeIntervals ( interval A[] , int n , interval B[] )
/* A is the input array of size n and B is the output array */
{
   int i = 0, j = 0; /* i is for reading from A, j is for writing to B */
   interval t; /* temporary variable */

   sortIntervals(A,n); /* Sort A with respect to the left end-points */
   while (1) {
      /* Store in t the next unprocessed input interval */

      t = ____________________________; ++i;
      /* As long as merging is possible */

      while ((i < n) && ( ____________________________________________ )) {
         /* Look at the possibility of extending the right end-point of t */

         if ( ____________________________________ ) t.b = ______________________;
         ++i;
      }
      /* Write the accumulated interval t to the output array B */

      ____________________________________________________________________

      /* Check whether all input intervals are processed */

      if ( ______________________ ) return ____________; /* Return the size of B */
   }
}
```

**(b)** Suppose that **`sortIntervals()`** runs in $O(n \log n)$ time. What is the running time of the function **`mergeIntervals()`** (Circle the best answer)? **(2)**

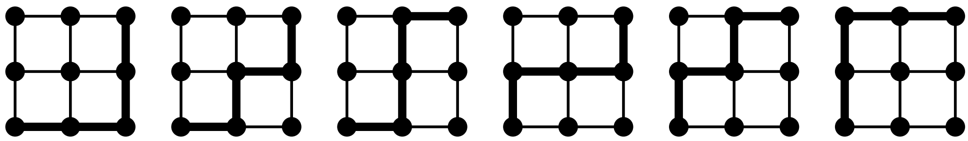
**(A)** $O(n)$ **(B)** $O(n \log n)$ **(C)** $O(n^2)$ **(D)** $O(2^n)$

**5.** [*Josephus problem*] Imagine a situation where $n$ convicts numbered $1, 2, \ldots, n$ are sitting (in that order) at a round table in a prison. The prosecutor keeps on circling around the table in the order $1, 2, \ldots, n$ and then back to 1. He starts his rotation at convict 1 and kills (and removes) every $m$-th convict he encounters. After $n - 1$ iterations, only one convict survives and is set free. As an example, take $n = 10$ and $m = 3$. The convicts are killed in the order $3, 6, 9, 2, 7, 1, 8, 5, 10$, and convict $4$ survives.

In this exercise, you write a program that, given $n$ and $m$, determines the survivor. You are asked to use a *circular linked list* which is like an ordinary linked list with the only exception that the pointer of the last node points to the first node (instead of being **NULL**). The following program first creates a circular list on $n$ persons and then simulates the prosecutor by traversing around the circular list. The program finally prints the survivor. No dummy node is maintained at the head. However, in order to make deletion easy, the running pointer points to the node immediately before the node to be deleted. **(10)**

```
#include <stdio.h>

typedef struct _node {
   int number;
   struct _node *next;
} node;

int main ()
{
   node *head,  /* the header */
        *p;     /* the running pointer (prosecutor) */
   int n, m, i;

   scanf("%d%d",&n,&m);

   /* Allocate memory for the first node */
   p = head = (node *)malloc(sizeof(node)); p -> number = 1;
   /* Allocate memory for the remaining nodes */
   for (i=2; i<=n; ++i) {

      ____________________________________________________________

      ____________________________________________________________

      ____________________________________________________________
   }
   /* Let the last link point to the head */

   _______________________________________________________________

   /* As long as there are two or more living convicts */

   while ( __________________________________________ ) {
      /* Skip m-1 convicts */

      for (i=1; __________; ++i) ___________________________________

      printf("Convict %d killed\n", ________________________________);
      /* Delete the node from the list */

      ____________________________________________________________
   }

   printf("Convict %d survives\n", __________________________________);
}
```

**6.** You are given an $m \times n$ grid of points, where $m$ is counted along the $x$-direction and $n$ along the $y$-direction. The grid points are numbered as $(i, j)$ for $i = 1, 2, \ldots, m$, $j = 1, 2, \ldots, n$. The lower left corner has the number $(1, 1)$ and the top right corner has the number $(m, n)$. We plan to locate all the paths from $(1, 1)$ to $(m, n)$ such that each step in each path is either a forward or an upward movement from one grid point to a neighboring point (i.e., backward and downward movements are not allowed). Since there is a total of $m-1$ forward and $n-1$ upward movements in each path, the total number of paths is $\binom{m+n-2}{m-1}$ which corresponds to the choice of the forward steps in a sequence of $m+n-2$ steps. The following figure describes a $3 \times 3$ mesh and the $\binom{4}{2} = 6$ admissible paths through it (consider the bold edges).



Path 1 Path 2 Path 3 Path 4 Path 5 Path 6

```
Path 1 : (1,1) (2,1) (3,1) (3,2) (3,3)
Path 2 : (1,1) (2,1) (2,2) (3,2) (3,3)
Path 3 : (1,1) (2,1) (2,2) (2,3) (3,3)
Path 4 : (1,1) (1,2) (2,2) (3,2) (3,3)
Path 5 : (1,1) (1,2) (2,2) (2,3) (3,3)
Path 6 : (1,1) (1,2) (1,3) (2,3) (3,3)
```

We print these paths using a stack. We initially push the point $(1, 1)$ to the top of the stack. Against every point in the stack, we maintain a direction indicator which implies the next direction to explore from this point: 0 means along $x$ direction, 1 means along $y$ direction and 2 means that both directions are explored from this point. Whenever we push a point, we set its direction indicator to 0. When the indicator becomes 2, we pop it from the stack.

We then enter a loop which repeats as long as the stack is not empty. If the top stores the point $(m, n)$, then the stack contains a path from $(1, 1)$ to $(m, n)$. We print this path. We then determine whether further movement from this point is possible. If not, we pop the point from the stack. Otherwise we first check whether a movement in the $y$ direction is possible. If so, we push the point vertically above the current point, otherwise we push the point to the right of the current point. Before this push operation, the direction indicator of the current top is modified appropriately.

Complete the following program that implements this algorithm. Here we do not use the stack ADT calls, but write in-line codes for the push and pop operations. The top of the stack is copied to the variable **`top`**, and the index of the top of the stack is maintained in the variable **`topidx`**. The direction indicator is stored in the field **`dir`** of the structure **`element`**. **(10)**

```
#include <stdio.h>

#define MAXSIZE 100

typedef struct { int x, y, dir; } element;

void printPath ( element stack[] , int m , int n )
/* This function prints the current path stored in the stack */
{
   int i;

   for (i=0; i<=m+n-2; ++i) printf("(%d,%d) ", stack[i].x, stack[i].y);
   printf("\n");
}
```

Question 6 is continued to the next page. . .

```
int main ()
{
    int m, n, count, topidx;
    element stack[MAXSIZE], top;

    scanf("%d%d",&m,&n);
    /* Initialize the stack */
    stack[0].x = 1; stack[0].y = 1; stack[0].dir = 0; topidx = 0; count = 0;
    /* while the stack is not empty */

    while ( ________________________ ) {
        /* store the current top of the stack */
        top = stack[topidx];
        if ((top.x == m) && (top.y == n)) {
            /* Reached end of path. Print the path. */
            ++count; printf("Path %d : ", count); printPath(stack,m,n);
            top.dir = 2; /* no further exploration */
        }
        if (top.dir == 2) {
            /* Both directions explored. Pop the current top. */

            ________________________
        } else if ( (top.dir == 1) || ((top.dir == 0) && (top.x == m)) ) {
            /* Explore next step in y direction */

            if (top.y == n) ________________________ /* pop */
            else { /* push */

                stack[topidx].dir = ____________;

                topidx = ________________________;

                stack[topidx].x = ____________________________________________;

                stack[topidx].y = ____________________________________________;

                stack[topidx].dir = ____________;
            }
        } else {
            /* Explore next step in x direction */

            __________________________________________________________________

            __________________________________________________________________

            __________________________________________________________________

            __________________________________________________________________

            __________________________________________________________________
        }
    }
}
```