CS10003 Programming and Data Structures
Spring 2024 – 2025, Class Test 2
April 03, 2025 (Thu), 6:45pm – 7:45pm
Maximum Marks: 40

Roll No: ______________________________

Name: ______________________________

[*Write your answers in the question paper itself. Answer* ***all*** *questions.* ***All programs must be written in C.*** *Fill in the blanks in the following codes to make them work as described.* ***Do not use extra variables.*** *Not all blanks carry equal marks. Evaluation will depend on overall correctness.*]

1. **Jump Search** is an algorithm for searching in sorted arrays. Although slower than binary search, jump search is much faster than linear search. Jump search uses a doubly nested loop, and employs a jump strategy at exponentially increasing indices. Like binary search, jump search maintains a search interval $I = [L, R]$. To start with, the interval I encompasses the entire array. Each iteration of the outer loop reduces the interval I to a strictly smaller sub-interval $I' = [L', R']$ determined as follows. It keeps on looking at the indices **1, 2, 4, 8, 16,** … relative to L until an index is found, at which the array element is greater than or equal to the key, or the array size is exceeded. These comparisons help to choose the sub-interval I' to the correct one from $[L, L]$, $[L+1, L+1]$, $[L+2, L+3]$, $[L+4, L+7]$, $[L+8, L+15]$, . . . , $[L+2^t, R]$. In the next iterations, the same search mechanism is continued, on gradually shrinking intervals. The loop stops when the interval length reduces to one. A final comparison of the key is made with the element at the only index of the interval, and depending on the outcome of this comparison, L (key found) or –1 (key not found) is returned. Fill in the blanks below so that the function `jumpsearch()` works as intended. Assume that a sorted array is passed as `A`, and `n` > 0. **[10]**

```
int jumpsearch ( int A[], int n, int key )
{
   int L, R, jump;
   /* Initialize the search interval I = [L, R] */

   L = ____________0____________ ; R = __________n - 1__________ ;        (1+1)
   /* Repeat until I shrinks to a single-element interval */

   while ( __________L < R__________ ) {                                  (1)
      jump = 1;       /* jump is the next search index relative to L */
      while (1) {     /* Repeat until the correct sub-interval is located */
                        L + jump > R
         if ( (L + jump >= R is also OK) ) break; /* Next search index is beyond R */   (1)

         /* Compare key with the array element indicated by jump */

         if ( ______key < A[L + jump]______ ) {      /* R is determined now */        (1)

            R = __________L + jump - 1__________ ; break;                          (1)
         } else {                                    /* Continue the search */

            L = __________L + jump__________ ;                                     (1)

            jump = __________jump * 2__________ ;                                  (1)
         }
      }
   }
   /* Make a final comparison, and return L or -1 */

   _______________return (key == A[L]) ? L : -1;_______________                   (2)

}
```

2. The following C program attempts to reverse an array of integer elements by using both call-by-value and call-by-reference swap functions. The call-by-value swap function (called `fakeSwap`) does not swap values of external variables, whereas the call-by-reference swap function (called `trueSwap`) swaps values of external variables. The function `reverseElements` still manages to reverse the array elements (magically using a trick!) by periodic swapping of elements from the two ends of the array, using both the functions `fakeSwap` and `trueSwap` (if you have a doubt, check the working of the function on arrays with odd and even numbers of elements). With this understanding, complete the following C program (by filling in the given blanks) so that it performs as indicated. **[10]**

```
#include <stdio.h>
/* erroneous swap routine which calls its arguments by value */

void fakeSwap ( ______________int a______________ , ______________int b______________ ) { (0.5+0.5)
    int temp;
    /* swap values using the variable temp (write three assignments only) */

    _____________________temp = a; a = b; b = temp;_____________________  (0.5 x 3)
}
/* correct swap routine which calls its arguments by reference/address */

void trueSwap ( ______________int *a______________ , ______________int *b______________ ) { (0.5+0.5)
    int temp;
    /* swap values using the variable temp (write three assignments only) */

    ____________________temp = *a; *a = *b; *b = temp;____________________  (0.5 x 3)
}
/* tricky function for reversing the array */

void reverseElements ( _________int A[] or int *A_________ , ______________int n______________ ) (1+0.5)
{
    int i, flag = 0;
    for ( i = 0; i < n; ++i ) {
        if ( i != n-1-i ) {
            if ( flag ) trueSwap(&A[i], &A[n-1-i]); else fakeSwap(A[i], A[n-1-i]);
            flag = !flag;
        }
    }
}
int main () {
  int *elm, n, i;
  printf("Enter number of elements: "); scanf("%d", &n);

  elm = ( ____________int *____________ )malloc(n * ____________sizeof(int)____________ );  (1+1)
  printf("Enter Array of %d integer elements:\n", n);

  for ( i = 0; i < n; ++i ) scanf("%d", ___________&A[i] or A + i___________ );  (0.5)
  reverseElements(elm, n);
  printf("Reversed array of %d integer elements:\n", n);
  for ( i = 0; i < n; ++i ) printf(" %d", *( ______________A + i______________ ));  (1)
  printf("\n");
  return 0;
}
```

3. In the partial C program given below, *A* is an array of *n* positive integers. We plan to sort *A* in place, using an algorithm called **digisort**. This algorithm repeatedly sorts the array *A*, based on a digit position starting from the least and going to the most significant ends of the numbers. As an example, take A = { 415, 73, 516, 923, 890, 318 }. In the first iteration, the array gets sorted based on the rightmost/least-significant digits, and changes to { 89**0**, 7**3**, 92**3**, 41**5**, 51**6**, 31**8** }. The second iteration sorts *A* using the second/middle digits, changing *A* to { 4**1**5, 5**1**6, 3**1**8, 9**2**3, **7**3, 8**9**0 }. The third/last iteration looks at the the leftmost/most-significant digits, and changes *A* to { **0**73, **3**18, **4**15, **5**16, **8**90, **9**23 }.

Not all elements of *A* are required to have the same number of digits (see the element 73 in the above example). In order to address this issue, we first compute the maximum in *A*, and the number of digits in that maximum element. All elements of *A* can now be considered to consist of these many digits. In this exercise, you are not asked to write codes for these operations.

Each digit-based sorting iteration uses a two-dimensional array *B* with 10 rows, one for each of the digits 0, 1, 2, . . . , 9. One by one, the corresponding digit *d* of *A*[*i*] is extracted, and *A*[*i*] is appended to the *d*-th row of *B*. A count array *C* of size 10 keeps track of how many elements of *A* are sent to the different rows of *B*. After all *A*[*i*] are copied to their correct rows, the rows of *B* are copied sequentially, back to *A*.

Fill in the blanks below so that the digisort algorithm works as explained above. **[10]**

```
void digisort ( int A[], int n )
{
   int max, maxdigitcnt, digitpos, tenpower, d, i, j;
   int B[10][MAX_SIZE], C[10];
   max = findmax(A,n);              /* Find the maximum element in A[] */
   maxdigitcnt = digitcount(max);   /* Find the number of digits in max */
   /* The following loop runs on digitpos from least to most significant positions */
   tenpower = 1; /* This will always store 10-to-the-power digitpos */

   for ( ___________digitpos = 0; digitpos < maxdigitcnt; ++digitpos___________ ) {    (0.5 x 3)

      _______for (d = 0; d < 10; ++d) C[d] = 0;_______  /* Initialize all counts to 0 */  (1)
      for (i = 0; i < n; ++i) {

         d = ___(A[i] / tenpower) % 10___ ;  /* Compute the digitpos-th digit of A[i] */  (1)
         /* Append A[i] to the d-th list B[d] */

         ____________________B[d][C[d]] = A[i];____________________                       (1.5)

         ____________________++C[d];____________________                                  (1)
      }
      /* Copy back the digit-wise sub-lists from B to A. Index j is for writing to A. */
      j = 0;
      for ( ____________d = 0; d < 10; ++d____________ ) { /* Loop on d */               (1)

         for ( ____________i = 0; i < C[d]; ++i____________ ) { /* Loop on i */          (1)

            A[j] = ____________B[d][i];____________ ; ++j;                                (1)
      }
      tenpower = ______tenpower * 10______ ;  /* Update tenpower for next iteration */  (1)
   }
}
```

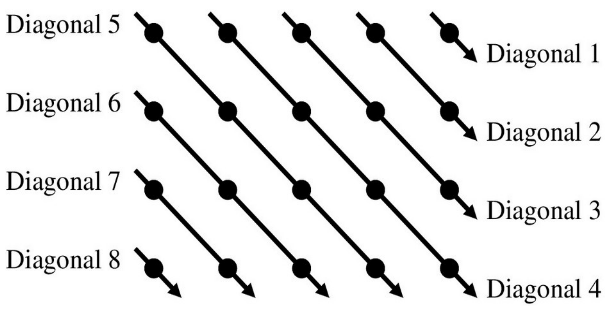
4. A two-dimensional array is defined in the `main()` function of a C program as: `int *A[MAX_SIZE]`. In the `main()` function, the user supplies the number *r* of rows and the number *c* of columns in *A*. The following function is then called to initialize the array with user inputs. Each row is allocated memory to store exactly *c* `int` variables. Fill in the blanks to complete the function. **[3]**

```
                              *A[]
void initMatrix ( int ____________________ , int r, int c )                    (1)
{
   int i, j;
   for (i = 0; i < r; ++i) {  /* i is the row index */
      /* Allocate the exact amount of memory for allocating each row */

                        (int *)malloc(c * sizeof(int));
      A[i] = _____________________________________________________________ ;   (1)
      for (j=0; j<c; ++j)   /* j is the column index */

                      &A[i][j]
         scanf("%d, _or A[i] + j or *(A + i) + j_ );   /* Read the (i,j)-th element of A */  (1)
   }
}
```

The `main()` function then calls the following function to print the diagonals of *A*. A diagonal of *A* starts from the left or the top boundary, proceeds toward south-east, and ends at the right or the bottom boundary. Let us number the diagonals in the south-west direction starting from top right. As an example, the adjacent figure shows a 4×5 matrix. The elements are represented by black dots. Note that not all the diagonals contain the same number of elements. In the function below, `d` stands for the number of the diagonal (under the numbering scheme mentioned above), the pair (`i, j`) runs over the indices on a diagonal, and the printing of a diagonal starts at indices (`row_start, col_start`). Fill in the details of the function so that it correctly prints the diagonals sequentially. **[7]**



```
                                  *A[]
void printDiagonals ( int ____________________ , int r, int c )                 (1)
{
   int d, i, j, row_start, col_start;

    /* Initialize the start indices for printing Diagonal 1 */

                      0                                  c - 1
   row_start = ____________________ ; col_start = ____________________ ;     (0.5+0.5)

                       r + c - 1
   for ( d = 1; d <= ____________________ ; ++d ) {   /* d is the diagonal number */    (1)

      printf("Diagonal %d:", d);

      i = row_start; j = col_start;  /* Indices of the first position on the diagonal */

                          (i < r) && (j < c)
      while ( ____________________________________________________________ ) {   (1)

         printf(" %d", A[i][j]);

         /* Update both i and j to the next position on the diagonal */

                               ++i; ++j;
         ____________________________________________________________________      (1)
      }
      printf("\n");  /* Printing the d-th diagonal is complete */

      /* Prepare for printing the next diagonal */

                  if (col_start > 0) --col_start; else ++row_start;
      _________________________________________________________________________    (2)
   }
}
```