INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Stamp / Signature of the Invigilator

EXAMINATION (End Semester)	**SEMESTER (Autumn)**

Roll Number										**Section**		**Name**	

Subject Number	C	S	1	1	0	0	1	**Subject Name**	*Programming and Data Structures*

Department / Center of the Student		**Additional sheets**	

Instructions and Guidelines to Students Appearing in the Examination

1. Ensure that you have occupied the seat as per the examination schedule.
2. Ensure that you do not have a mobile phone or a similar gadget with you even in switched off mode. Note that loose papers, notes, books should not be in your possession, even if those are irrelevant to the paper you are writing.
3. Data book, codes or any other materials are allowed only under the instruction of the paper-setter.
4. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items is not permitted.
5. Additional sheets, graph papers and relevant tables will be provided on request.
6. Write on both sides of the answer script and do not tear off any page. Report to the invigilator if the answer script has torn page(s).
7. Show the admit card / identity card whenever asked for by the invigilator. It is your responsibility to ensure that your attendance is recorded by the invigilator.
8. You may leave the examination hall for wash room or for drinking water, but not before one hour after the commencement of the examination. Record your absence from the examination hall in the register provided. Smoking and consumption of any kind of beverages is not allowed inside the examination hall.
9. After the completion of the examination, do not leave the seat until the invigilator collects the answer script.
10. During the examination, either inside the examination hall or outside the examination hall, gathering information from any kind of sources or any such attempts, exchange or helping in exchange of information with others or any such attempts will be treated as adopting ‘unfair means’. Do not adopt ‘unfair means’ and do not indulge in unseemly behavior as well.

Violation of any of the instructions may lead to disciplinary action.

Signature of the Student

To be filled in by the examiner

Question Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total**
Marks Obtained											

Marks obtained (in words)	**Signature of the Examiner**	**Signature of the Scrutineer**

CS11001 Programming and Data Structures, Autumn 2014–2015

End Semester Examination

Date: 27–November–2014 Time: 09:00–12:00 hours Maximum Marks: 100

Write your answers in the question paper. Write only in the blank spaces provided. Do not use any extra variables. Do not change the interpretations of the variables. Be neat and tidy. Answer all questions. Not all blanks carry equal marks.

1. (a) Write in one or two sentences what the following program prints. **(4)**

```
int f ( char *head, char * tail )
{
    int count;

    if (head >= tail) return 0;
    count = (*head == *tail) ? 0 : 1;
    return count + f(head+1,tail-1);
}

main ( )
{
    char S[1000], *head, *tail;

    scanf("%s", S);
    head = S;
    tail = S + strlen(S) - 1;
    printf("%d\n", f(head, tail));
}
```

Let the length of S be n. The program counts and prints the number of indices i in the range $0 \leqslant i \leqslant \lfloor n/2 \rfloor - 1$ such that $S[i] \neq S[n-1-i]$. This is the minimum number of symbols in S that need to be changed in order to make S palindromic.

(b) Let n be the length of the string S. Derive a recurrence relation for the running time $T(n)$ of the function $f(S)$. Solve the recurrence to express $T(n)$ in the big-Oh notation as a *simple* function of n. **(2 + 4)**

$T(0)$ and $T(1)$ are constant. For $n \geqslant 2$, a recursive call is made on a string of size $n-2$ (after performing some constant number k of elementary operations). Thus, we have

$$T(n) = T(n-2) + k.$$

Let $n = 2m + a$ with $a \in \{0, 1\}$. It then follows that:

$$\begin{aligned} T(n) &= T(n-2) + k \\ &= T(n-4) + 2k \\ &= T(n-6) + 3k \\ &\cdots \\ &= T(a) + mk \\ &= T(a) + m\lfloor n/2 \rfloor . \end{aligned}$$

Since $T(a)$ and k are constant, we have $T(n) = O(n)$.

---

2. Consider two integer arrays A and B of the same size $n \geqslant 2$. We define a relation $A \prec B$ if there exists an index j in the range $0 \leqslant j \leqslant n-1$ such that $A[j] < B[j]$, and for all i in the range $0 \leqslant i < j$, we have $A[i] = B[i]$. Note that for any two arrays A and B of size n, exactly one of the three cases hold: (a) $A \prec B$, (b) $B \prec A$, or (c) $A = B$ (that is, A and B are identical for all elements). As examples, take $n = 3$, and observe that $(1,2,3) \prec (1,2,4) \prec (1,4,2) \prec (3,0,0)$.

You are given an $m \times n$ two-dimensional array M. Treat each row of M as a one-dimensional array of size n. The relation $\prec$ just defined apply to the rows of M. Your task is to bubble sort the rows of M with respect to this relation. This means that if R_1 and R_2 are two rows of M with $R_1 \prec R_2$, then R_1 should appear before R_2 in the sorted output.

(a) Complete the following function which takes two arrays `A` and `B` of size `n` as inputs, and returns $-1, 1, 0$ according as whether $A \prec B$, $B \prec A$, or $A = B$, respectively. **(4)**

```
int compare ( int A[], int B[], int n )
{
    int i;

    for ( ____i=0; i<n; ++i____ ) { /* loop on i */

        if ( ____A[i] < B[i]____ ) return -1;

        if ( ____A[i] > B[i]____ ) return 1;
    }

    return ____0____ ;
}
```

(b) Complete the following function that bubble sorts the rows of an $m \times n$ two-dimensional array `M` with respect to the relation $\prec$. Assume that `m` and `n` are not larger than a suitably defined `MAX_SIZE`. **(10)**

```
void rowsort ( int M[][MAX_SIZE], int m, int n )
{
    int i, j, k, t;

    for (i = ____m-2____ ; i >= 0; i--) {

        for ( j = ____0____ ; ____j <= i____ ; ____++j____ ) {

            if ( compare( ____M[j]____ , ____M[j+1]____ , ____n____ ) ____> 0____ ) {
                /* Swap rows */

                for ( ____k=0; k<n; ++k____ ) {

                    ____t = M[j][k]; M[j][k] = M[j+1][k]; M[j+1][k] = t;____
                }
            }
        }
    }
}
```

3. In this exercise, we deal with linked lists which are kept sorted in the increasing order of the data values. To this end, we define a node in the list in the usual way as:

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

A list is specified by a pointer to the first node in the list. We assume that all the nodes in the list store valid data items, that is, there is no dummy node at the beginning of the list.

(a) Let `H` be a pointer to the first node in a sorted linked list. `H` is `NULL` if the list is empty. We plan to insert a data value `a` in the list such that the list continues to remain sorted after the insertion. The value `a` to be inserted may or may not be already present in the list. Complete the following function to carry out this sorted insertion. The function should return a pointer to the first node in the modified list. **(12)**

```
node *sortedinsert ( node *H, int a ) {
    node *p, *q;

    /* Create a new node to store the new value */

    p = ____(node *)malloc(sizeof(node))____ ; /* Allocate memory */

    ____p -> data = a____ ; /* Store a in the new node */

    /* Handle insertion before the first node of the list. This should include
       the case of insertion in an empty list. */

    if ( ____(H == NULL) || (a <= H -> data)____ ) {

        ____p -> next = H;____

        return ____p____ ;
    }

    /* Now we are in a situation where we insert after a node. We first let q point to
       the node after which the data item a will be inserted. */

    q = ____H____ ; /* Initialize q for the search */
    /* Loop on q to locate the correct insertion position */

    while ( ____(q -> next) && (a > q -> next -> data)____ ) q = q -> next;

    /* Finally, insert p after q */

    ____p -> next = q -> next; q -> next = p;____

    return ____H____ ;
}
```

(b) In this part, assume that you are given a sorted linked list with possible duplicate data items stored in consecutive nodes. Complete the following function that removes all duplicate values (that is, if a data value is present multiple times, the function will retain only one instance of the data). The function returns a pointer to the updated list having no duplicate items. It uses three pointers `p`, `q` and `r`. Their roles are explained as comments in the function. Notice that in the freeing loop, `q` is no longer needed to point to the last node with the same data whose possible repetitions are deleted, and can be reused as a temporary variable. The body of the outer `while` loop should work even if there is only one instance of a data item. **(10)**

```
node *rmdup ( node *H )
{
   node *p, *q, *r;

   p = ______H______ ; /* Initialize p. It will run over the nodes in the list. */

   while ( ____________p____________ ) { /* so long as p points to a node with data */
      q = p ;
      /* When the following loop breaks, q will point to the last node with the same
         data as is pointed to by p */

      while ( ___(q -> next) && (q -> next -> data == p -> data)___ ) q = q -> next;
      r = p -> next; /* r will be used in the freeing loop */
      /* Adjust pointer(s) to remove all nodes after p and before q -> next */

      ________________p -> next = q -> next;________________
      /* Free all the nodes deleted from the list in this iteration */

      while ( ____________r != p -> next____________ ) { /* loop on r */

         ____________________q = r; r = r -> next; free(q);____________________
      }

      ____________p = p -> next;____________ /* Advance p for the next iteration */
   }
   return H;
}
```

4. Consider a polynomial with real (floating-point) coefficients: $f(X) = c_0X^{d_0} + c_1X^{d_1} + c_2X^{d_2} + \cdots + c_{t-1}X^{d_{t-1}}$ with integer degrees $0 \leqslant d_0 < d_1 < d_2 < \cdots < d_{t-1}$ and with coefficients $c_i \neq 0$. We call each $c_iX^{d_i}$ a non-zero term in $f(X)$. We store f as the sequence $(c_0,d_0),(c_1,d_1),(c_2,d_2),\ldots,(c_{t-1},d_{t-1})$ which is sorted with respect to the degrees (the second components in the pairs). We first define a term as follows:

```
typedef struct {
   double coeff; /* The coefficient in a non-zero term */
   int degree;   /* The degree of X in the term */
} term;
```

A polynomial is then stored as a structure instance of the following type:

```
typedef struct {
   int nterms;     /* The number of non-zero terms */
   term *termlist; /* The list of terms, that is, (coefficient,degree) pairs */
} poly;
```

We assume that the term-list contains a sequence of terms sorted in the increasing order of the degrees, and that no two terms have the same degree. All coefficients are assumed to be non-zero. Moreover, the term-list should be allocated memory just sufficient to store all the non-zero terms in the polynomial.

Let us have two polynomials f and g in the above representation. We plan to compute their sum $h = f + g$ again in the same representation. Before the sum is computed, the number of non-zero terms in h is not known, so we prepare a local array `sum[]` to store the maximum possible number of terms that can appear in the sum. The intermediate addition result is stored in `sum[]`. Finally, the term-list of h is allocated the exact amount of memory as needed, and the local array `sum[]` is copied to the term-list of h.

As an example, let $f(X) = 1 - 2X^3 - 3X^7 + 4X^9 - 5X^{15}$ and $g(X) = 4 + 3X + 2X^3 + X^9$. Their sum can have a maximum of nine non-zero terms. The sum $h(X) = f(X) + g(X) = 5 + 3X - 3X^7 + 5X^9 - 5X^{15}$ actually contains only five non-zero terms. This happens because each of the sums $1 + 4$ and $4X^9 + X^9$ introduces only one new term. Moreover, the sum $-2X^3 + 2X^3$ does not add to the sum any term involving X^3.

(a) We first write a recursive helper function to generate the intermediate array `sum[]`. This function uses a merging procedure as in merge sort (notice that the term-lists of `f` and `g` are sorted with respect to the degrees of the terms). The term-lists of `f` and `g` are indexed by `i` and `j`, respectively. The result is stored in

the intermediate array `sum[]`, and the index `k` is used for writing to this array. Thus `k` stores the number of non-zero terms. In the recursive calls, the indices `i`, `j` and `k` are changed appropriately. The outermost call in Part (b) gets the *exact* number of non-zero terms in the final sum. Complete the helper function. **(15)**

```
int addhelper ( poly f, poly g, int i, int j, term *sum, int k )
{
   /* If both f and g are completely read, return the number of non-zero terms */
   if ((i == f.nterms) && (j == g.nterms)) return k;
   /* If g is completely read (but not f), or the current term in f has lower degree
      than that in g, then copy the current term in f to sum */
   if ( ( j == g.nterms ) ||

        ( (i < f.nterms) && (f.termlist[i].degree < g.termlist[j].degree) ) ) {

      sum[k] = f.termlist[i] ;

      return addhelper( f, g, i+1, j, sum, k+1 );
   }
   /* If f is completely read (but not g), or the current term in g has lower degree
      than that in f, then copy the current term in g to sum */

   if ( (i == f.nterms) || (f.termlist[i].degree > g.termlist[j].degree) ) {

      sum[k] = g.termlist[j] ;

      return addhelper( f, g, i, j+1, sum, k+1 );
   }
   /* Here the current terms in both f and g have the same degree */

   sum[k].degree = f.termlist[i].degree ;

   sum[k].coeff = f.termlist[i].coeff + g.termlist[j].coeff ;

   if (sum[k].coeff) ++k; /* Update k if needed */

   return addhelper( f, g, i+1, j+1, sum , k );
}
```

(b) Complete the following function that adds `f` and `g` by invoking the helper function of Part (a). **(4)**

```
poly add ( poly f, poly g )
{
   poly h;
   term *sum;
   int i;

   /* Allocate the maximum possible amount of memory that may be needed for sum */

   sum = (term *)malloc((f.nterms + g.nterms) * sizeof(term)) ;
   h.nterms = addhelper(f,g,0,0,sum,0); /* Call the helper function */
   /* Allocate the exact amount of memory to the term-list of h */

   h.termlist = (term *)malloc(h.nterms * sizeof(term)) ;
   /* Copy the intermediate array sum[] to the term-list of h */

   for (i=0; i<h.nterms; ++i) h.termlist[i] = sum[i];

   free(sum); /* Clean locally used dynamic memory */
   return h;
}
```

5. In a doubly linked list, each node has a link in the forward direction and another link in the backward direction. The forward link of the last node and the backward link in the first node of the list are `NULL`.

```
typedef struct _node {
    int data;
    struct _node *flink; /* forward link pointing to the next node */
    struct _node *blink; /* backward link pointing to the previous node */
} node;
```

A *double-ended queue* is an ordered list in which insertion and deletion can occur at both the ends. Let us represent a double-ended queue by a doubly linked list as follows. This is an array of two pointers, the first pointer (at index zero) pointing to the first node of the linked list and the second (at index one) the last node of the list. In an empty queue `Q`, both the pointers `Q[0]` and `Q[1]` are `NULL`.

```
typedef node *d_e_queue[2];
```

(a) Complete the following insert function which takes three arguments: a double-ended queue `Q`, the integer `a` to be inserted, and the `end` (zero or one) where insertion would take place. **(13)**

```
void insert ( d_e_queue Q, int a, int end )
{
    node *p;
    p = ____(node *)malloc(sizeof(node))____ ; /* Allocate memory */
    p -> data = a;

    if ( ____Q[0] == NULL____ ) { /* Insertion in an empty queue */
        ____p -> flink = p -> blink = NULL;____ /* Set the links of p */

        ____Q[0] = Q[1] = p;____ /* Set the pointers in Q */
        return;
    }
    if (end == 0) { /* Insertion at the beginning of the list */

        ____p -> flink = Q[0]; p -> blink = NULL;____ /* Set the links of p */

        Q[0] -> blink = p; ____Q[0] = p;____ /* Set Q[0] */
    } else { /* Insertion at the end of the list */

        ____p -> blink = Q[1]; p -> flink = NULL;____ /* Set the links of p */

        ____Q[1] -> flink = p; Q[1] = p;____ /* Adjust Q[1] */
    }
}
```

(b) Complete the following deletion function that takes two arguments: a double-ended queue `Q` and an `end` (zero/one) at which deletion occurs. Free the node being deleted. **(9)**

```
void delete ( d_e_queue Q, int end )
{
    if (Q[0] == NULL) return; /* Deletion from an empty queue */
    if (Q[0] == Q[1]) { /* Deletion from a queue with one element */

        ____free(Q[0]); Q[0] = Q[1] = NULL;____
        return;
    }
    if (end == 0) { /* Deletion at the beginning of the list */

        ____Q[0] = Q[0] -> flink; free(Q[0] -> blink); Q[0] -> blink = NULL;____
    } else { /* Deletion at the end of the list */

        ____Q[1] = Q[1] -> blink; free(Q[1] -> flink); Q[1] -> flink = NULL;____
    }
}
```

6. You start with an empty stack S, and then perform n push and n pop operations on S such that:

- You push the integers $1, 2, 3, \ldots, n$ in that order. Assume that the stack has enough memory to store n elements, that is, no push operation fails.
- You never pop from an empty stack, that is, no pop operation fails, so after the n push and the n pop operations S becomes empty again.

Immediately before each pop operation, you print the top of the stack, that is, the element which is going to be popped. The above rules indicate that the integers $1, 2, 3, \ldots, n$ are printed in some order. The printed sequence is said to be *realized* by a stack.

For example, let $n = 5$. The permutation $2, 4, 3, 5, 1$ can be realized by the following sequence of push and pop operations (the print statements are not shown): push(1), push(2), pop(), push(3), push(4), pop(), pop(), push(5), pop(), pop(). Observe that this example sequence satisfies the specified rules. We push the integers in the order $1, 2, 3, 4, 5$. We perform the same number of pop operations and never pop from an empty stack. Not every permutation of $1, 2, 3, \ldots, n$ can be realized by a stack. For example, convince yourself that for $n = 5$ the permutation $2, 4, 1, 3, 5$ cannot be realized by a stack.

The following function takes as input n and an array `seq[]` storing a permutation of $1, 2, 3, \ldots, n$, and checks whether the input sequence can be realized by a stack. There are no external `push()` and `pop()` functions. We instead use a local array `S[]` as the stack, and write the codes for push and pop explicitly in the function body. The variable `top` stores the index of the stack top in `S[]`. The variable `a` stores what element is to be inserted in the stack in the next push operation. Finally, the variable `i` stands for how many integers are printed (that is, how many pop operations are carried out). The function also prints the maximum-length prefix of the input sequence that can be realized using the stack. Complete the function. **(13)**

```
void check ( int *seq, int n )
{
    int a, top, i, *S;

    S = ____(int *)malloc(n * sizeof(int))____ ; /* Allocate memory */
    a = 1; top = -1; i = 0; /* Initialize */
    /* Repeat the following loop until explicitly broken in the body */
    while (1) {
        /* First check whether it is necessary to print and pop now */

        if ( ____(top >= 0) && (seq[i] == S[top])____ ) {

            printf("%d ", ____S[top]____ );

            ____--top____ ; /* Pop from S */

            ____++i____ ; /* Another integer printed */
        } else
        /* Then check whether it is allowed to push another element to S */

        if ( ____a <= n____ ) {

            ____S[++top] = a____ ; /* Push a to S */

            ____++a____ ; /* Prepare for the next push */

        } else ____break____ ;
    }

    if ( ____i == n____ )
        printf("+++ The input sequence can be realized by a stack...\n");
    else
        printf("+++ The input sequence cannot be realized by a stack...\n");

    ____free(S)____ ; /* Clean locally allocated dynamic memory */
}
```