Department of Computer Science & Engineering, IIT Kharagpur
CS11001 Programming & Data Structure

Endterm, Spring 2012, Time: 3 hours, Date: 25th April, 2012

Answer the questions in the spaces provided on the question sheets. You may use the **Extra Page** in this answer booklet for answers/ rough work. No other supplementary sheets will be given to you.

Roll Number		Section	
Name			

Question:	1	2	3	4	5	Total Marks
Points:	15	15	15	15	15	75
Score:						

A list of some useful library functions is given below defined in string.h and ctype.h:

- `char * strcat ( char * destination, char * source );` Appends a copy of the `source` string to the `destination` string. The terminating null character in `destination` is overwritten by the first character of `source`, and a new null-character is appended at the end of the new string formed by the concatenation of both in `destination`.
- `char * strcpy ( char * destination, const char * source );` Copies the C string pointed by `source` into the array pointed by `destination`, including the terminating null character.
- `int isalpha ( int c );` Checks if parameter `c` is an alphabetic letter (uppercase or lowercase).
- `int isupper ( int c );` / `( int islower (int c);)` Checks if parameter `c` is an uppercase / (lowercase) alphabetic letter.
- `int tolower ( int c );` / `int toupper (int c);)` Converts parameter `c` to its lowercase / (uppercase) equivalent if `c` is an uppercase / (lowercase) letter and has a lowercase / (uppercase) equivalent. If no such conversion is possible, the value returned is `c` unchanged.

1. Consider the following data type to represent a character set that represnets a set of English alphabetic characters (maximum size 26). The characters are stored in lower case form.

```
typedef struct {
   char elems[26] ;
   int size;
} CSET ;
```

A string is an array of characters terminated by he character `'\0'`

(a) Write a function `member ( . )` , which takes a character (which is an English alphabet) x and a set S of type CSET as parametrs, and returns 1 if x (or its lowercase equivalent if x is uppercase) occurs in the set S, otherwise it returns 0. [3]

(b) Write a function `adjoin ( . )` which takes a character x and a character set S as parameters, and adds the character x to the set S, and returns the resulting character set. [3]

(c) Write a function `intersect ( . )` which takes as parameters two character sets (type CSET), computes the set intersection, and returns the intersection set. [3]

(d) Write a function `makeset ( . )` which takes as parameter a string (containing English characters),and returns a character set (CSET) that contains those characters as members. [3]

(e) Write a `main ( )` which takes two strings as command line arguments. You may assume that the strings contain alphabetic characters only. It should call the above functions to get a set that contains the characters that are common to both the strings. Print the common characters. [3]

2. (a) Consider the following program segment: [4]

```
#include <stdio.h>
int foo (int a, int b, int *v1) {
      if (a>b) {
         *v1 = a; return b;
      }
      *v1 = b; return a;
}
int main ( ) {
        int n1, n2, n3 ;
        int  v1, v2, tmp;
        scanf ("%d%d%d", &n1, &n2, &n3) ;
        v2 = foo (n1, n2, &v1) ;
        v2 = foo (v2, n3, &tmp) ;
        tmp = foo (v1, n3, &v1) ;
        printf  ("v1 = %d, v2 == %d\n", v1, v2) ;
        return 0;
}
```

In the above program, state what will be the values of $v1$ and $v2$ that will be printed in terms of the values of the inputs $n1$, $n2$ and $n3$.

v1 : .....

v2: .....

(b) [5]

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main() {
    char *s = (char*)malloc(12*sizeof(char));
    char *t = (char*)malloc(15*sizeof(char));
    strcpy(s, "good");
    strcpy(t, "hi there");
    char *p = t+4;
    t = s;
    strcat(s, "bye");
    printf("s = %s\n", s);
    printf("t = %s\n", t);
    printf("p = %s\n", p);
    return 0;
}
```

i. What is the output of the above program?

ii. This program allocates memory without freeing it. Suppose you add the following code right before `return 0;`.

```
free(s);
free(t);
```

Once this code is added, does it free the dynamic storage properly? Why or why not?

(c) Consider the following recursive function which takes an array of integers a, and the length of that array n as parameters, and returns an integer. State clearly what this function computes. [2]

```
int mysteryFunction(int *a, int n) {
    if (n==0)
        return 0;
    if (a[0]%2 != 0)
        return 1 + mysteryFunction(a+1, n-1);
    else
        return mysteryFunction(a+1, n-1);
}
```

(d) Write a recursive function `is_embedded (char s1[], char s2[])` to check if the characters of `s1` are embedded in string `s2` (occur in the same order in `s2`, but not necessarily contiguous). [4]

Example: `is_embedded (''ant'', ''anarchist'')` should return 1, but `is_embedded (''ahar'', ''anarchist'')` should return 0.

```
int is_embedded (char s1[], char s2[] ) {

    // Base cases
    if (s1[0] == '\0')

    .....

    if (s2[0] == '\0')

    .....

  // Recursive cases

  .....

  .....

  .....

  .....

  .....

  .....

}
```

[Rough Work]

3. In the following problem we conside the representation of a black-and-white rectangular image. The image has height h and width w pixels. The image is stored as a 2d-array, where the 2d-array index (i, j) denotes the coordinate of the pixel in 2d-space. Each pixel has the value: 0 or 1 (0 indicates background, and 1 indicates object point).

Implement the following functions:

(a) `int ** alloc( int h, int w)` : Dynamically allocate space for storing an array of pixels of dimension $h \times w$. Return the address of the dynamically allocated memory. [4]

(b) `void read(int **mat, int h, int w)` : Read the values of the $h \times w$ array of pixels. Assume each pixel value could be given as either 0 or 1, and the input is read row-wise, and at each row column-wise. [3]

(c) `int distance(int x1, int y1, int x2, int y2)` : Compute the Manhattan distance between two points $(x1, y1)$ and $(x2, y2)$ in the $x - y$ plane. The Manhattan distance is defined as $|x1 - x2| + |y1 - y2|$. [3]

(d) `int neighbours (int **mat,int h, int w, int x, int y, int k)` : Find the number of object points in the array *mat* of size $h \times w$ that are within a Manhattan distance of k from point (x, y). [5]

4. (a) Consider the implementation of a linked list for which every insertion takes place at the end of the list. The starting element of the list is pointed by a variable head and the list is terminated by a NULL pointer.

```
struct node {
   int key;
   struct node *next;
}; // Structure of a node of the list.
```

Implement the following functions:

i. `struct node * insert(struct node * head, int val)`: Insert a node with the key value as val at the end of the linked list. Returns the head of the list. [5]

ii. `int printsecond (struct node * head )` : Print the second element of the linked list. Return -1 if the second element does not exist, and 1 if it exists. [2]

iii. `int count (struct node * head)` : Return the number of elements in the list. [2]

iv. `void printreverse ( struct node * head)` : Take as input a linked-list of integers and prints the elements of the list in reverse order (i.e. last element first) using recursion. (The list is not to be changed.) [3]

v. Write a `main( )` function, that [3]

- creates/ initializes a linked list,
- calls the function `insert ( )` 3 times to insert nodes containing the following values to the linked list: 20, -10, and 8.

5. (a) Consider the following functions that search for an element x in an array A containing n elements. The functions are called as `search1/ search2 / search3 (A, 0, n, x)`. For each of these functions [6]

   i. State whether the function returns the correct result when called on any array of integers.

   ii. State whether the function works correctly when called on an array of integers sorted in ascending order.

   iii. What is the minimum number of '*'s that will be printed when the function is called on an array of n integers (which is the same as the number of times the function is called in the best case)?

   iv. What is the maximum number of '*'s that will be printed when the function is called on an array of n integers (worst case number of function calls)?

   The answers may be given by filling up the table below.

```
int search1 (int A[], int left, int right, int x) {
      printf (''*'');
      if (left == right) return 0;
      if (A[right-1] == x) return 1;
      return search1 (A, left, right-1) ;
}
int search2 (int A[], int left, int right, int x) {
       int m;
       printf (''*'');
       if (left == right) return 0;
       m = (left+right)/2;
       if (A[m] == x) return 1;
       if (x < A[m]) return search2  (A, left, mid, x) ;
       return search2 (A, mid+1, right, x) ;
}
int search3 (int A[], int left, int right, int x) {
       int m, v1, v2, v3;
       printf (''*'');
       if (left == right) return 0;
       m = (left+right)/2;
       if (A[m] == x) v3= 1;
       v1 = search2  (A, left, mid, x) ;
       v2 = search2 (A, mid+1, right, x) ;
      return ((v1 || v2) || v3) ;
}
```

	i. correctly works for any array	ii. correctly works for sorted array	iii. Minimum no of '*'s printed	iv. Maximum no of '*'s printed
search1				
search2				
search3				

(b) Suppose that you are performing the *selection sort* algorithm as described below: [3]
After the entire sort, elements will be in order from smallest to largest Each pass through the array "selects" the *largest* element from the smaller array, and swaps it into place. With that in mind, indicate the values of each element of the array after the 1st, 2nd and 3rd pass.

	a[0]	a[1]	a[2]	a[3]	a[4]
initial values	42	88	13	9	27
after 1st swap					
after 2nd swap					
after 3rd swap					
after 4th swap (final array)	9	13	27	42	88

[Rough Work]

(c) What will be printed when the following program executes? [2]

```
int partition( int  a[], int low, int high ) {
  int left, right, pivot, temp, pivot_item;
  pivot_item = a[low];
  pivot = left = low;
  right = high;
  while ( left < right ) { // Move left while item < pivot
    while( a[left] <= pivot_item ) left++;
    while( a[right] > pivot_item ) right--;
    if ( left < right )   { // swap  a[left] with a[right];
        temp = a[left];
        a[left] = a[right];
        a[right] = temp;
    }
  }   // right is final position for the pivot
  a[low] = a[right];
  a[right] = pivot_item;
  return right;
}
int main ( ) {
   int pos, i;
   int A[10] = {22, 17, 46, 12, 82, 5, 19, 80, 61, 20} ;
   pos = partition (A, 0, 9) ;
   printf ("pos = %d\n", pos) ;
   for (i=0; i<10; i++) printf ("%d, ", A[i]) ;   printf ("\n") ;
   return 0;
}
```

(d) Complete the function `qsort ( )` that calls the `partition ( )` function from the previous question, and uses the *quicksort* algorithm to sort an array of integers [4]

```
void qsort (int a[], int low int high) {
     int pos;
     if (low >= high) return;
     pos = partition (a, low, high) ;

.....

.....

.....

.....
}
```