INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
Department of Computer Science & Engineering

Programming and Data Structures (CS11001)
Endsem (Autumn, 1st Year)

Date: Tue, Nov 22, 2011 *Time:* 09:00-12:00pm
Students: 660 *Marks:* 80

Answer ALL the questions. Write you name and roll number on ALL sheets.
Do all rough work on separate rough sheets which you should NOT submit.
Answer neatly, without overwriting on the question paper itself in the spaces provided.

Roll no: __________ **Section:** _____ **Name:** ________________

1. Complete the following function which takes as input a given positive integer (representing a sum of money) and prints the number of currency notes of denominations 1, 10 and 50 required to pay that amount, so that the number of currency notes is minimised. Eg: 320→6×50+2×10; 475→9×50+2×10+5×1

```
void printPayment (____int N____) {                                  1
  int n1, n10, n50;

  n50 = N/50__________________________________________;              1

  N = N - 50 * n50____________________________________;              1

  n10 = N/10__________________________________________;              1

   n1 = N - 10 * n10__________________________________;              1
  printf("Number of notes of 1, 10 and 50 denominations are:  ");

  printf("%d, %d and %d\n", n1, n10, n50______________________);     1
}
```

2. A rabbit has found itself in a maze of tunnels connecting underground pits. Initially, each tunnel has one carrot in it and each pit has two or four tunnels connecting it to other pits. The rabbit intends to start at a pit, eat some of the carrots in the tunnels and definitely return to the starting pit. It can do so if: (*i*) it enters a tunnel only if there is a carrot there (assume there is some light to see), (*ii*) eat that carrot and (*iii*) go to the pit at the other end of the tunnel. Eventually, it will be at a pit from where there is no tunnel with a carrot in there. *That pit is bound to be the starting pit!* Let there be N pits numbered $0..(N-1)$and M tunnels numbered $0..(M-1)$. Information on the pits and tunnels are represented as described below; fill in the missing parts.

Information on the M tunnels is stored in an array `tunTyp tunnel[M]`, where `tunTyp` is defined as:

```
typedef {
  int p1, p2; // index values of pits it connects

  int carrotCount; // initialised to __1__ (1:  carrot present, 0:  absent)    1
} tunTyp;
```

Information on the N pits is stored in an array `pitTyp pit[N]`, where `pitTyp` is defined as:

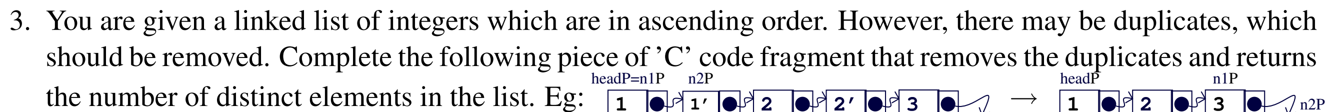
```
typedef {

  int tunnelCount; /* set to __2__ or __4__, depending on connectivity */     1

  int tunVec[__4__]; /* vector of index values of                              1
                          maximum number tunnels to which this pit is connected */
} pitTyp;
```

1	2	3	4	5	6	T

Now, complete the function that prints a possible round tour (returning to the starting point) of some of the pits and tunnels by the rabbit, starting at the pit with index of 0.

```
void printPitTunTour (int M, tunTyp tunnel[M], int N, pitTyp pit[N]) {
  int nTun_forPit, tunVecIdx, tunIdx, otherPitIdx, carrotsEaten=0, pitIdx=0;
  repeat { // keep looping, until the tour ends
    // check if there is a tunnel with a carrot...
    //   first find out how many tunnels are there with this pit

    nTun_forPit = pit[pitIdx].tunnelCount ______________________;          1
    //   next, search for a tunnel with a carrot among tunnels in tunVec

    for (tunVecIdx=__0__; tunVecIdx<____nTun_forPit____; tunVecIdx++) {   2
      // ...get the index of the tunnel to check for a carrot in tunIdx

      tunIdx = pit[pitIdx].tunVec[tunVecIdx] __________________;          1
      // ...check if this tunnel has a carrot

      if (_____tunnel[tunIdx].carrotCount_____) { // ...carrot present    1

        tunnel[tunIdx].carrotCount = 0 _______________; // eat carrot      1
        // identify other end of tunnel in otherPitIdx

        otherPitIdx = tunnel[tunIdx].p1 + tunnel[tunIdx].p2 - pitIdx

        _______________________________________________________;          2
        // carrot eaten, ...tour must continue from pit with index of otherPitIdx
        //   first, display the move...
        printf("eating carrot and going from pit %d to pit %d\n",

          pitIdx, otherPitIdx ____________________________________);       2
        //   next, prepare to carry on from the new pit...

        pitIdx = otherPitIdx ____________________________________;         1
        //   don't forget to keep count of carrots eaten...

        carrotsEaten++ __________________________________________;         1
        //   finally, ...carry on!

        break; __________________________________________________;         1
      } // end-if
    } // end-for
  } // check if the search done above for a tunnel with a carrot did fail...

  until (________tunVecIdx >= nTun_forPit________); // tour over!          1

  printf("%d of %d carrots were eaten\n", ___carrotsEaten___, __M__);      2
} // end-printPitTunTour
```

The running time of the above function in the big-O notation is: $O(M)$ OR $O(N^2)$ 1

3. You are given a linked list of integers which are in ascending order. However, there may be duplicates, which should be removed. Complete the following piece of 'C' code fragment that removes the duplicates and returns the number of distinct elements in the list. Eg: 


```
typedef     struct llNodeTag      { // ...via struct for linked list of ints
  int val; // value stored in node

  struct llNodeTag          *nextP; // pointer to next node

} llNodeTyp, *llNodePtr                    ;
// type names for struct and pointer to struct                                 3

int removeDuplicates (llNodePtr headP) {
  int elemCount=0; // needs to be incremented when a new value is seen
  llNodePtr n1P, n2P;
  n1P = headP;
  if (n1P != NULL) { // list is not empty
  // n2P is initialised to point to the successor of n1P

  n2P = n1P->nextP                                  ;                          1

  elemCount++                                       ;                          1

  } else return 0;                                                             1
  // keep looping to find duplicates

  while (             n2P ! = NULL              ) {                            1
    // test for a duplicate

    if (n1P->val == n2P->val              ) { // duplicate found               1
      // steps to remove the duplicate at n2P

      n1P->nextP = n2P->nextP                      ;                           1

      free(n2P)                                    ;                           1

      n2P = n1P->nextP                             ;                           1
    } else { // advance in the list

      n1P = n2P                                    ;                           1

      n2P n1P->nextP                               ;                           1

      elemCount++                                  ;                           1
    }
  }

  return elemCount                                 ; // final step             1
}
```

4. Write a function `charCount()` to count the characters, corresponding *only* to the letters of the English alphabet, occurring in a `NUL` terminated string `char s[]` ignoring the case (upper or lower) of the letters. The count is to be maintained in an array `counts[]` of 26 integers corresponding to each letter of the alphabet.

```
// the supplied string of characters is s[]
// letter counts are maintained and returned via counts[]
void charCount(char s[], int counts[]) {
int i; // next, initialisations of locations in count[]

  for (i=0; i<26; i++) counts[i] = 0;                         2

  for (i=0;i<n;i++) {                                         2

    if ('a'<=s[i] && s[i]<='z') count[s[i]-'a'] += 1;         2

    if ('A'<=s[i] && s[i]<='Z') count[s[i]-'A'] += 1;         2

    if (s[i] == '\0') return;                                 2
  }
}
```

5. A test for divisibility by 7 is stated as follows: *Take the two left-most digits, multiply the left digit by 3 and add it to the second digit. Replace these two digits with the result. Then we can keep repeating, always dealing with only the two left-most digits, until we end up with a single digit number which is either divisible by 7 (if digit is 0 or 7) or not.*

   Eg: *249* → *109* → *39* → *18* → *11* → **4** (not divisible by 7); *49* → *21* → **7** (divisible by 7).

   For the above divisibility test by 7, assume that a k digit number N (most significant digit, N_{k-1} is non-zero) is available in the array `int N[K]={`N_{k-1}`, ...,`N_1`, `N_0`}`, suitably initialised with the digits of the number.

   Consider a function `int isDiv7(int k, int N[])` to test divisibility by 7 of the k digit number whose digits are stored in the array `N[k]`. Giving single lines of 'C' code (comparisons, assignments, etc), describe each of the following tests and actions for the **B**ase and **I**nductive/recursive cases, *in terms of k and entries in the array* `N[]`,

   'C' code fragments to support the case analysis of `isDiv7(k, N)`:

   Base case N is a single digit number not divisible by 7

   Condition code (CB1): `k==1 && N[0]!=0 && N[0]!=7` 2
   Action code (AB1): `return 0; // number is not divisible`

   Base case N is a single digit number divisible by 7

   Condition (CB2): `k==1 && (N[0]==0 || N[0]==7)` 2
   Action (AB2): `return 1; // number is divisible`

   Inductive/recursive case N is multi-digit and needs to be reduced (including the action if N_{k-1} becomes 0)

   Condition code (CI1): `k>1` 1

   Action code (AI1a): `t = 3*N[k-1] + N[k-2]` 1

   Action code (AI1b): `N[k-2] = t % 10` 1

   Action code (AI1c): `N[k-1] = (t-N[k-2])/10;` 1

   Action code (AI1d): `if ( N[k-1]==0 ) k=k-1 ;` 1

   Action code (AI1e): `return isDiv7(k, N)` 1

Based on the above condition tests (CB1, CB2, CI1) and actions (AB1, AB2, AI1a, ... , AI1e) develop **isDiv7(k, N)** as a recursive routine. Instead of rewriting code, develop the routine in terms of the code aliases (short names for the code fragments): CB1, CB2, CI1, AB1, AB2, etc.

```
isDiv7(int k, int N[]) {
  int t; // declarations

    if (          CB1          ) AB1;                                    2

    if (          CB2          ) AB2;                                    2

    if (          CI1          ) {        AI1a; ...  ; AI1e        }    2
  }
```

Now develop **isDiv7(k, N)** as an iterative routine. Instead of rewriting code, develop the routine in terms of the code aliases (short names for the code fragments): CB1, CB2, CI1, AB1, AB2, etc.

```
isDiv7(int k, int N[]) {
  int t; // declarations

  while(1)                              { // start the loop            1

    if (CI1) { AI1a; ...  ; AI1d; }                                     2

    else break;                                                         1
  } // end of the loop

  if (CB1) AB1;                                                         2

  if (CB2) AB2;                                                         2
```

6. (a) The function call to open a text file **data.txt** for reading is:

   fopen(data.txt, "r") 1

   (b) The function call to open a text file **data.txt** for writing is:

   fopen(data.txt, "w") 1

   (c) The function call to open a text file **data.txt** in the directory **dataDir** under the current working directory, for appending is:

   fopen(dataDir/data.txt, "a") 1

   (d) When the function call to open a file fails, the return value of the function is: **NULL** 1

   (e) The function call to check whether the end of an open file **FILE *fileP** has been reached is:

   feof(fileP) 1