Name:

B1 **Roll No:** **Section:**

ESC101: End-Semester Exam

Time: 8:30am-11:30am Max: 100 points

November 24, 2012

Instructions.

1. Write your name and Roll no in the space provided in the top right hand side of this paper. Write your Roll No on all the sheets too.
2. This question paper has 11 pages and is printed back to back. Write your answers only in ink and only in the space provided. Questions 1-4 are of general relevance and carry 75 marks (15 + 15 + 25 + 20). Questions 5 and 6 are data structure questions totalling 25 marks (13+12). Questions 7 and 8 are numerical track questions and total 25 marks (12 + 13). You may attempt questions totalling 100 or more marks. Marks scored in the best scoring six out of the eight questions will be considered and added to obtain the score. It is suggested that Questions 1 through 4 be attempted followed by either questions 5 and 6 OR questions 7 and 8.
3. The relevant parts of the operator precedence table is provided below for reference. Left-to-Right stands for left-associative and Right-to-Left stands for right associative.

(expr) [*index*] -> .	Left-to-Right
unary operators & * (*type*) ! − ++ -- (*sizeof*)	Right-to-Left
* % /	Left-to-Right
+ −	Left-to-Right
< > <= >=	Left-to-Right
== !=	Left-to-Right
&&	Left-to-Right
||	Left-to-Right
=	Right-to-Left
,	Left-to-Right

Question	Max. Marks	Marks Obtained
Q1.	15	
Q2.	15	
Q3.	25	
Q4.	20	
Q5.	13	
Q6.	12	
Q7.	12	
Q8.	13	
Total Best 6/8	100	

Q1. General (15 marks. All parts have 2 marks, except 1.3, which has 3 marks). Write the output of the following programs. If you think there is any error in the program, please explicitly mention the reason and type of error : compile error, possible memory fault, unsafe and compiler dependent (i.e., non-portable). If there is an infinite loop, please indicate so. Assume that the code includes ⟨stdio.h⟩ and ⟨stdlib.h⟩ at the start.

Q**1.1**

```
struct student {
    int rollnumber;
    char* name;
}
main () {
    struct student chinti;
    struct student *pchinti = &chinti;
    chinti.name = "chinti";
    chinti.rollnumber = 10001;
    printf("%s %s", chinti.name+3,
        pchinti->name+2);
}
```

Soln. nti inti

Q**1.2**

```
int foo(int param1, int param2) {
    if(param1==param2)
        return param1;
    else if(param1>param2)
        return foo(param1-1, param2+1);
    else
        return foo(param1+1, param2-1);
}
main () {
    printf("%d\n", foo(8,2));
    printf("%d\n", foo(4,1));
}
```

Soln.
5
No output: Infinite Loop

Q**1.3**

```
void copyn(char **new, char* old, int size) {
    *new = calloc(size, sizeof(char));
    int i;
    for(i=0; i<size; i=i+1)
        *new[i] = old[i];
}
main () {
    char* s = "aabbcc"; char *t;
    int size = 6;
    copyn(&t, s, size);
    int m = 0; int i=0;
    for(i=0; i<size; i = i+1){
        m = m + (t[i] - 'a');
    }
    printf("%d %d \n", m);
}
```

Soln. 6 a garbage integer gets printed
There is no compilation error since printf accepts a variable number of arguments and no checks are made at compile time to ensure that the number of conversions match the number of valid arguments.

Q**1.4** (3)

```
struct person {
    char *name;
    int age;
};
void init(struct person *p) {
    char N[8]="BigDon";
    p->name=s;  p->age=99;
}
void main() {
    struct person *p1 = calloc(1, sizeof(struct person));
    init(p1);
    printf("%s %d\n",p1->name,p1->age);
}
```

Soln. Illegal memory reference as char N[] is local to init()

Q**1.5** (2 marks) Same as the previous question, except that the variable char N[8] in function init() is replaced by static char N[8].

Soln. BigDon 99

Q**1.6 What is the output of f2 for a general** $a[]$ **of size** n**.**

```
void f2(int a[], int n){
    if(n<1) return;
    f2(a+1, n-1);
    printf("%d", a[0]);
}
```

Soln. f2 prints the elements of a in reverse order.

Q**1.7**Fill in the blanks so that the function int sumdig(int n) returns the sum of the decimal digits of a given non-negative integer n by computing recursively.

```
int sumdig(int n) {
    if (n==0 || n==1) return n;
    else return  n %10 + sumdig(n/10) ;
}
```

One version of the problem is

```
int sumdig(int n) {
    if (n>=0 || n<=9) return n;
    else return  n %10 + sumdig(n/10) ;
}
```

Q2. (**Simple Geometry–General. (15 marks)**) This question builds a small set of geometric functions. You have to fill in the blanks to complete the library. The structures struct point, struct triangle and struct line are defined for representing a 2-dimensional point, a triangle in a 2-d plane and a straight line in 2-dim plane, respectively. A 2-dimensional point is represented by its (x, y) coordinates, a triangle by its end points A, B and C, and a line by its slope m and y-intercept c. There is no representation for vertical lines.

```
struct point{
    double x;  double y;
};
struct line {
    double m; double c;
};
```

Q**2.1** /* create_point creates a new point with a, b as the x and y coordinates respectively and returns a pointer to the newly created point */

```
struct point * create_point(double a, double b){
    struct point pp =
        calloc(1, sizeof( struct point ));
     pp->x =a;
     pp->y =b;
    return pp;
}
```

Q**2.2** /* create_line takes two *distinct* points pointed to by pp1 and pp2 and returns a pointer to a struct line that represents the line passing through them.*/

```
struct line * create_line(struct point *pp1,
        struct point *pp2){

    struct line *pl =
    calloc(1, sizeof( struct line ));
    pl->m =
 (pp2->y-pp1->y)/(pp2->x - pp1->x)  ;

    pl->c =
 pp1->y - pl->m*pp1->x ;
    return pl;
}
```

Q**2.3** Explain in one line what happens if we do not free pl1, pl2 and pl3 in the the lies_outside() function.

Soln. If we call lies_outside a large number of times, then we may end up occupying a lot of memory without actually using most of it.

```
struct triangle{
    struct point A;
    struct point B;
    struct point C;
};
```

Q**2.4** /* is_perp returns 1 if the lines pointed to by pl1 and pl2 are perpendicular to each other, otherwise it returns 0 */

```
int is_perp(struct line *pl1, struct line *pl2){

    return  (pl1->m*pl2->m==-1) ;
}
```

/* lie_on_sameside returns 1 if points pointed to by pp1 and pp2 lie on the same side of the line pointed to by pl; otherwise it returns 0 */

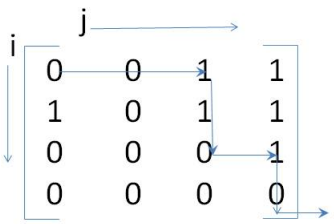
```
int lie_on_sameside(struct line *pl,
    struct point * pp1, struct point *pp2){
    double v1 = pl->m*pp1->x - pp1->y + pl->c;
    double v2 = pl->m*pp2->x - pp2->y + pl->c;

    return  v1*v2>0  ;
}
 /* return v1*v2>=0 is also marked correct. */
```

Q**2.5** /* lies_outside(pt,pp) returns 1 if the point pointed to by pp lies outside the triangle pointed to by pt; otherwise it returns 0. You must use lie_on_sameside function to fill the blank */

```
int lies_outside(struct triangle *pt, struct point *pp){
    struct line *pl1 = create_line(&pt->A,& pt->B);
    struct line *pl2 = create_line(&pt->B, &pt->C);
    struct line *pl3 = create_line(&pt->C, &pt->A);
    int result =
     !(lie_on_sameside(pl1, pp, &pt->C)
      &&lie_on_sameside(pl2, pp, &pt->A)
         && lie_on_sameside(pl3, pp, &pt->B)) ;
    free(pl1); free(pl2); free(pl3);
    return result;
}
```

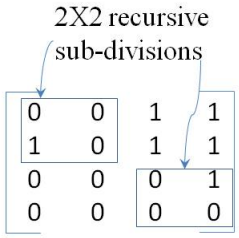
Q3. (Find Celebrity, General (12.5 + 12.5 = 25)

There are n persons numbered arbitrarily from 0 to $n-1$. The input $n \times n$ integer matrix *knows* specifies whether i knows j or not, that is, $knows[i][j]$ is 1 if and only if i knows j and is 0 otherwise. Note that the value of $knows[i][j]$ gives no information about $knows[j][i]$ (for e.g., I may know SRK but SRK does not know me). By convention, $knows[i][i]$ is 0. A person is defined to be a *celebrity* if *everyone else knows* this person and he/she *doesn't know anyone* in the group. By definition, there can be at most one (or zero) celebrity in the group. Also observe that (a) if i knows j then i cannot be a celebrity, and (b) if i does not know j, then j cannot be a celebrity. Fill in the blanks in the following two functions, (a) int IsCelebrity(int knows[][N], int n) that returns 1 if i is a celebrity index in the $n \times n$ *knows* matrix and returns 0 otherwise, and, (b) int FindCelebrity(int knows[][N], int n) that returns the index of the celebrity person if there is one, and returns -1 otherwise.



Iterative Program. In the *knows* matrix shown, person numbered 3 is a celebrity. The function findCelebrity is intended to work as follows. Set i initially to 0. The invariant is that persons 0 through $i-1$ are not celebrities. The loop scans the entries of row i starting at column position $j = i+1$ and moves forward, stopping at the first non-zero entry in the or if it has gone past the $n-1$st entry. In the latter case, person i does not know the persons numbered $i+1$ to $n-1$. Hence, it is the only possible celebrity among the last $n-i$ persons. The function then checks explicitly whether person i is a celebrity (i.e., the ith row is all 0 and the ith column is all 1's, except for the diagonal entry). In the former case, i does not know the persons between $i+1$ through $j-1$, so none among these persons can be a celebrity. But i knows j since $knows[i][j]$ is 1 and so i cannot be a celebrity. Since none of the first $i-1$ elements are celebrities by the invariant, we conclude that none among the first $j-1$ items can be a celebrity. We can now test for j to be a celebrity among the last $n-j+1$ persons, and the iteration continues with i set to j.

```
#include <stdio.h>
const int N = 100;
int IsCelebrity(int knows[][N], int n, int i) { /* is person i a celebrity? */
    int j =0; int is_celeb = 1;
    while (j < n && is_celeb) {

        if ( knows[i][j] || (j!=i &&!knows[j][i]) ) {
            is_celeb = 0;
        }
        else{ j = j+1;  }
    }
    return is_celeb;
}
int findCelebrity(int knows[][N], int n) {
    int i=0; int j = i+1;
    while (j < n) {
        if (knows[i][j]) { /* i knows j and i does not know 0..j-1. So 0..j-1 are not celebrities, */
            /* i is also not a celebrity. By invariant 0..i-1 are not celebrities. Next celebrity possibility is j */

            i =  j ;

            j =  i+1 ;
        else {  j = j+1 ; }
        }
    }
    if (  IsCelebrity(knows,n,i) ) { return i; }
    else {return -1; }
}
```



person 0 is a celebrity in top 2X2 submatrix.
person 3 is a celebrity in bottom 2X2 submatrix.
Combining, person 3 is the only celebrity in the 4X4 matrix

Recursive Program. The function int find_celeb(int knows[][N], int n) finds a celebrity, if it exists, recursively. It is based on the observation that if we divide the group of n people into any two non-empty sub-groups, say the persons $0 \ldots, k-1$ and the persons $k \ldots n-1$, then, the celebrity person in the entire group must be a celebrity in the group it lies in. So, if we recurse on the first and the second groups, we get two candidate celebrities, and we have to check and verify which is the actual celebrity (if any). In a single person group, that person is obviously a celebrity–this is the exit condition for recursion. To facililate the recursion, a function int Celeb_recurse(int knows[][N], int start, int len) is written that returns the index of the celebrity, if one exists, in the group of persons with indices start, ..., start +len-1; otherwise, it returns -1.

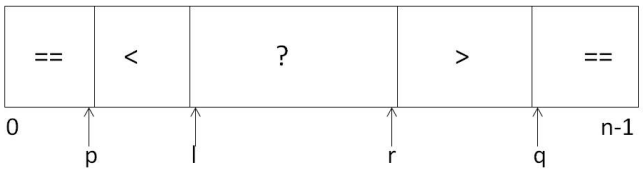
```
#include <stdio.h>
const int N = 100;
int check_celeb(int knows[][N], int i, int start, int len) { /*Checks and returns 1 if i is a celebrity
in the group consisting of start, ..., start + len -1. Returns 0 otherwise. */
    int j = start;
    int is_celeb = 1;
    while (is_celeb &&  j < start +len ) {

        if ( (!knows[i][j] && (knows[j][i] || j==i)) ) {
            j = j+1;
        }
        else { is_celeb = 0; }
    }
    return is_celeb; }
int celeb_recurse(int knows[][N], int startrow, int len) {
/* returns the index of person in the group startrow, startrow +1, ..., startrow + len-1 who is a celebrity,
if there is one, otherwise, returns -1. */
    if (len ==1 ) return startrow;
    else {
        int celeb1, celeb2;
        celeb1 = celeb_recurse(knows,  startrow, len/2 );
        celeb2 = celeb_recurse(knows,  startrow+len/2, len-len/2 );
        /* check which of the celeb1 or celeb2 is the actual celebrity */
        if ( celeb1 >=0 && check_celeb(knows, celeb1,  startrow+len/2, len - len/2 ))
            return celeb1;
        else if (celeb2 >=0 && check_celeb(knows, celeb2,  startrow, len/2 ))
            return celeb2;
        else return -1;
    }
}
int find_celeb(int knows[][N], int n) { /* top level call */
    return celeb_recurse(knows,0,n); }
```

Q4. Partition. General (20 marks) This is now reduced to 10 marks, only the blanks for quicksort() have scores of $2+2+3+3$ respectively. Fill in the blanks so that (1) part_3way() terminates with the properties as described, and, (2) quicksort() works correctly. (20 marks)

A 3-way partition function part_3way(int a[], int n, int *part1, int *part2) is a variation of the int partition(a[], int n) function used for *quicksort*. It permutes the array $a[0], \ldots, a[n-1]$ and returns 2 indices *part1 equaling u (say) and *part2 equaling v such that all array elements in $a[0, \ldots, u-1]$ are less than the pivot, all elements in $a[u, \ldots, v]$ are equal to the pivot, and all elements in $a[v+1, \ldots, n-1]$ are larger than the pivot. As with standard partition(), the pivot is $a[0]$. For example, corresponding to the input array $\boxed{2\ 1\ 3\ 1\ 5\ 2}$ the output of part_3way is $\boxed{1\ \ 1}\boxed{2\ \ 2}\boxed{5\ \ 3}$, with u and v as 2 and 3 respectively.

An outline of the 3-way partition is shown in the function part_3way, that uses the function swap() for exchanging integers. The program maintains the following invariant: at the beginning of each iteration (i.e., just before or after the test of the outer while loop, line number 5), the indices p and q satisfy the property: (1) $a[0, \ldots, p]$ are all equal to pivot and contain all the occurrences of the pivot in $a[0, \ldots, l-1]$ (i.e., there are no pivots in $a[p+1, \ldots, l-1]$, and, (2) $a[q, \ldots, n-1]$ are all equal to pivot and are all the occurrences of the pivot in $a[r+1, \ldots, n-1]$. In addition, the standard loop invariants of the binary partition also hold, namely, (3) $a[0, \ldots, l-1]$ are each at most pivot, (4) $a[r+1, \ldots, n-1]$ are each at least pivot. Further, after the outer while loop terminates, $0 \le r \le n-2$ and $0 \le l \le n-1$.

```
#include ⟨ stdio.h⟩
void swap(int *x, int *y) {
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
    }

void part_3way(int a[], int n, int *part1, int *part2) {
    1.  int l =0, r = n-1; /* as usual */
    2.  int p=-1; /* a[0..p] are all equal to pivot */
    3.  int q=n; /* a[q..n-1] are all equal to pivot */
    4.  pivot = a[0]; /* as usual */
    5.  while ( l < r) {
    6.      while (a[l] < pivot) { l = l+1; }
    7.      while (a[r] > pivot) { r = r-1; }
    8.      if (l >= r) break;
    9.      swap( &a[l] , &a[r] );
```




```
   10.      if ( a[l] == pivot ) {
   11.          p = p+1;
   12.          swap(&a[p], &a[l]); }
   13.      if ( a[r] == pivot ) {
   14.          q=q-1;
   15.          swap(&a[q], & a[r]);}
   16.      l= l+1 ;
   17.      r= r-1 ;
   18.  }
   19.  int i,j;
   20.  if (l==r) { j = r-1; }
   21.  else { j = r ; }
   22.  for (i= 0 ; i <=  p ; i=i+1) {
   23.      swap(&a[i],&a[j]);
   24.      j=j-1; }

   25.  *part1 =  j+1 ;
   26.  j =  r+1 ;

   27.  for (i= n-1 ; i >=  q ; i=i-1) {
   28.      swap(&a[i],&a[j]);
   29.      j=j+1; }

   30.      *part2 =  r+ n-q ;
}
void quicksort(int a[], int n) {
        int p1, p2;
        if (n < 0 )  return;
    /* Mistake: exit condition should be if (n <= 1)  return;
        else {

            part_3way(a,n,  &p1, &p2  );

            quicksort(  a, p1 );

            quicksort( a+p2+1, n-p2-1  ); } }
```

Q5. (**Data Structures. (13 marks)**)
We are given a line consisting of only the following six parentheses (or brackets) characters '(' ')' '{' '}' '[' ']' in arbitrary order and ending with the newline character '\n'. The program reads the line and prints whether the sequence of parentheses is balanced or not. The empty parenthesis expression is balanced, and if B is any balanced parenthesis expression, then, (B), $[B]$ and $\{B\}$ (and **BB**) are balanced. For example, (a) ({[]}) is balanced, but (b) ([)] is not balanced. Fill the blanks in the function main() so that it correctly prints whether the parentheses are balanced or not. It uses the stack data structure of characters to check the validity of such parenthesis expression. The functions in the file stack.h are as follows :

```
struct stack_struct {
        char *stk; /* array that stores the stack elements */
        int top; /* the index of stk pointing to the next available position to insert */
        int size; /* the allocation for the stk array */
};*/
typedef struct stack_struct * Stack; /* Defining a new type Stack
int IsEmpty_Stack(Stack ); /* Function returns 1 if the Stack is empty, 0 otherwise*/
Stack Create_Stack(int size); /* Function returns a Stack of size passed*/
char pop(Stack stack); /* Function returns a char after popping from the Stack*/
void push(Stack stack, char val ); /* Function pushes a char on the Stack*/
```

```
#include<stdio.h>
#include "stack.h"
void main() {
    char c; int isvalid = 1;
    int is_newline = 0;
    Stack stack = Create_Stack(100);
    while (isvalid && !is_newline) {
        c = getchar();
        if (c=='\n') {
            isvalid = IsEmpty_Stack(stack);
            is_newline = 1;
        }
        if(c=='(' || c=='{' || c=='[')  {  push(stack, c) ;  }
        else {
    /* If the stack is not empty then there should be matching open parenthesis on top of the stack */
            if ( IsEmpty_Stack(stack) ) { isvalid = 0; }
            else { char t =  pop(stack) ;

                if(!( (t == '[' && c==']')  ||  t == '{' && c == '}'  ||  t == '(' && c == ')'  )) {
                    isvalid = 0;
                }
            }
        }
    }
    if (isvalid) printf("parentheses is balanced\n");
    else printf("parantheses is NOT balanced\n");
}
```

Q6. (Data Structures) (12 marks)
Consider the singly linked list structure defined in the class. Complete the following function which reverses the sequences nodes of a given linked list **by modifying only the next pointers** of the nodes.

```
#include ⟨stdio.h⟩
#include ⟨stdlib.h⟩

struct node {
        int data;
        struct node *next;
};
typedef struct node *Listnode;

/* function which returns the head of reversed linked list */
Listnode reverse (Listnode head) {

        /* we have to process the remaining list from this Listnode onwards */
        Listnode curr = head;

        /* store the head of linked list reversed (processed) till current node */
        Listnode proc_head = NULL;

        Listnode temp; /* to help us keep track of pointers */
        while(  curr != NULL  ) {

                temp =  curr→next ;

                /* append the head of partial reversed list as curr→next */
                curr→next =  proc_head ;

                /* make current Listnode the head of linked list processed till now */
                proc_head  = curr;

                curr = temp; /* continue */

        }

        return proc_head;
}
```

Q7. (**Numerical Track. Floating Point (12 marks)**). The IEEE double precision format stores a floating point number using 64 bits, as follows.

sgn	e_{10}	e_9	$\ldots$	e_0	d_1	d_2	$\ldots$	d_{52}

The low order 52 bits, denoted by $d_1, \ldots, d_{52}$ represent the mantissa, the next 11 bits denoted by $e_0, \ldots, e_{10}$ form the 11 bit exponent and bit 64 is the sign bit. The exponent is an integer between 0 and 2047 represented by $e_{10}e_9 \ldots e_0$ in binary as $e = e_0 + 2e_1 + 2^2e_2 + \ldots + 2^{10}e_{10}$. The mantissa bits represent mantissa as the number $M = (1.0 + d_1 \cdot 2^{-1} + d_2 \cdot 2^{-2} + \ldots + d_{52} \cdot 2^{-52})$. The floating point number is given by

$$(-1)^{sgn} M \times 2^{e-1023}$$

Q7.1 In this representation, what is the smallest number by absolute value? (2)
The smallest number by absolute value has representation

$sgn\ = 0/1$	$e_{10} = 0$	0	$\ldots$	$e_0 = 0$	$d_1 = 0$		$\ldots$	$d_{52} = 0$

This number is 1.0×2^{-1023}.

Q7.2 What is the smallest number larger than 1.5 that has an exact representation in this format? (2 marks)
The number $1.5 = 1 + 1/2$ and has the representation

$sgn\ = 0/1$	$e_{10} = 0$	$e_9 = 1$	$\ldots$	$e_0 = 1$	$d_1 = 1$		$\ldots$	$d_{52} = 0$

The next number has the representation

$sgn\ = 0/1$	$e_{10} = 0$	$e_9 = 1$	$\ldots$	$e_0 = 1$	$d_1 = 1$	$d_2 = 0$	$\ldots$	$d_{51} = 0$	$d_{52} = 1$

and equals

$$(1 + 1/2 + 1/2^{52}) \times 2^{1023-1023} = 1.5 + 2^{-52} \ .$$

Q7.3 Suppose numbers are represented in this format by chopping (or truncating) the number to fit within the mantissa bits. Let a be a real number and a^* be its representation. Let $\rho = (a - a^*)/a$. Give the smallest upper bound on ρ that holds for all numbers a within the range of this representation. (2).

Let $a^* = M \times 2^e$ where M is the mantissa in the representation. Then, $a - a^* \leq (M + 2^{-52}) \times 2^e$ and so it follows that $(a - a^*)/a \leq 2^{-52}/M \leq 2^{-52}$.

Q7.4 Let $\rho = \max_{a \text{ is in representation range}} (a - a^*)/a$. *Derive* a tight bound in terms of ρ of the relative error of computing $a + b$ as $(a^* + b^*)^*$, that is, $1 - (a^* + b^*)^*/(a + b)$. Now comment on the relative error of computing $a - b$ as $(a^* - b^*)^*$. (2+1 = 3).

Assume that a and b have the same sign, otherwise the discussion for $a + b$ below applies to $a - b$ and vice-versa. If a and b have the same sign, without loss of generality assume that they are both positive. For any number within range, we have, $x(1 - \rho) \leq x^* \leq x$. So

$$1 - \frac{(a^* + b^*)^*}{a + b} = 1 - \frac{(a^* + b^*)^*}{a^* + b^*} \cdot \frac{a^* + b^*}{a + b} \leq 1 - (1 - \rho)(1 - \rho) \leq 2\rho \ .$$

Unfortunately, there can be no bound on the relative error of $a-b$, since, $(a-b)-\rho a \leq a^*-b^* \leq a-b(1-\rho) = (a-b)+\rho b$. The relative error is then

$$\left|1 - \frac{(a^* - b^*)^*}{a - b}\right| = \left|1 - \frac{(a^* - b^*)^*}{a^* - b^*} \cdot \frac{a^* - b^*}{a - b}\right| \leq 1 - \left(1 + \frac{(1 - \rho)\rho a}{a - b}\right) \approx \frac{\rho a}{a - b}$$

which has no bound, since a could be significantly larger than $a - b$, there cannot be an obvious bound. This is the phenomenon of (relative) error propagation due to subtraction of almost equal numbers. Precision has been lost.

Q7.5 (**Newton's method**).Give the iterative equation (relating p_n to p_{n-1}) obtained when we apply Newton's method to find the cube-root of a number n. (3 marks) Newton's method for finding the root of $f(x) = 0$ gives the iterate as $p_{k+1} = p_k - \frac{f(p_k)}{f'(p_k)}$, where, for finding the cube-root, $f(x)$ is $x^3 - n$. This gives,

$$p_{k+1} = p_k - \frac{p_k^3 - n}{3p_k^2} = \frac{2p_k}{3} + \frac{n}{3p_k^2}$$

Q8. (**Numerical Track. 13 marks** (4+3+3+3 = 13)) Consider the following second order Newton's method for finding roots of a function $f(x) = 0$, where, f is twice differentiable and the initial approximation x_0 is close to the actual root to ensure convergence.

Q**8.1** Expanding $f(x)$ around x_0 by Taylor's series upto the quadratic term and approximating x by $x_1 = x_0 + h$, show that we obtain the relation

$$h = -\frac{f'(x_0)}{f''(x_0)}\left(1 - \sqrt{1 - \frac{2f(x_0)f''(x_0)}{(f'(x_0))^2}}\right)$$

By Taylor's series, expanding $f(x)$ around x_0 and letting $x = x_0 + h$, we get

$$f(x) = f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(\xi), \quad \text{for some } \xi \in (x_0, x).$$

Now $f(x)$ is 0, and we make an approximation that $f''(\xi)$ is close to $f''(x_0)$, to yield,

$$0 \approx f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) \ .$$

Solving the above equation by viewing it as a quadratic for h, we get

$$h = \frac{-f'(x_0) \pm \sqrt{(f'(x_0))^2 - 2f(x_0)f''(x_0)}}{f''(x_0)} = -\frac{f'(x_0)}{f''(x_0)}\left(1 \pm \sqrt{1 - \frac{2f(x_0)f''(x_0)}{(f'(x_0))^2}}\right) \ .$$

Choose the $-$ sign of the $\pm$ to choose the smaller change for h in absolute value so that our approximation holds better.

Q**8.2** Using $1 - \sqrt{1-\alpha} \approx \frac{\alpha}{2} + \frac{\alpha^2}{8} + O(\alpha^3)$, for small values of α, show that

$$h = \frac{f(x_0)}{f'(x_0)}\left(1 + \frac{f(x_0)f''(x_0)}{2(f'(x_0))^2}\right)$$

Here we are assuming that $\alpha = \frac{2f(x_0)f''(x_0)}{(f'(x_0))^2}$ is small, and using the previous part and ignoring $O(\alpha^3)$, we get,

$$h \approx -\frac{f'(x_0)}{f''(x_0)}\left(1 - \sqrt{1-\alpha}\right) \approx -\frac{f'(x_0)}{f''(x_0)}\left(\frac{f(x_0)f''(x_0)}{(f'(x_0))^2} + \frac{(f(x_0))^2(f''(x_0))^2}{2(f'(x_0))^4}\right) = -\frac{f(x_0)}{f'(x_0)}\left(1 + \frac{f(x_0)f''(x_0)}{2(f'(x_0))^2}\right)$$

Q**8.3** Write the iterative equation expressing x_{n+1} in terms of x_n. This is Halley's method. Express the iterative equation x_{n+1} in terms of x_n for finding the cube-root of a positive number t.

The iterative equation is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\left(1 + \frac{f(x_n)f''(x_n)}{2(f'(x_n))^2}\right)$$

For finding the cube-root, $f(x)$ is $x^3 - t$. So, $f'(x_n) = 3x_n^2$ and $f''(x_n) = 6x_n$. Thus,

$$x_{n+1} = x_n - \frac{(x_n^3 - t)}{3x_n^2}\left(1 + \frac{(x_n^3 - t)}{3x_n^3}\right) = x_n\left(1 - \frac{1}{9}\left(1 - \frac{t}{x_n^3}\right)\left(4 - \frac{t}{x_n^3}\right)\right)$$

Q**8.4** Householder's method gives the iterative equation as

$$x_{n+1} = x_n + (p+1)\left(\frac{(1/f)^{(p)}}{(1/f)^{(p+1)}}\right)_{x_n}$$

where p is a non-negative integer and $(1/f)^{(p)}$ is the pth order derivative of the function $1/(f(x))$. Express the iterative equation x_{n+1} in terms of x_n for finding the fourth root of a positive number t.

For $p = 0$, Householder's method is the same as Newton's method, and iterative method is identical. For finding the fourth root, $f(x) = x^4 - t$, so that, $f'(x_n) = 4x_n^3$, $f''(x_n) = 12x_n^2$, $f'''(x_n) = 24x_n$ and $f^{(4)}(x_n) = 24x_n$.

For $p = 0$, $(1/f)^{(1)} = -f'/f^2$ and

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^4 - t}{4x_n^3} = \frac{3x_n}{4} + \frac{t}{4x_n^3} \ .$$

For $p = 1$,

$$(1/f)^{(2)} = -(f^{(1)}/f^2)^{(1)} = -(f^{(2)}/f^2) + 2(f^{(1)})^2/f^3 = \frac{2(f')^2 - ff''}{f^3}$$

and so

$$x_{n+1} = x_n - \left.\frac{2(1/f)^{(1)}}{(1/f)^{(2)}}\right|_{x_n} = x_n + \frac{2}{2f'(x_n)/f(x_n) - f''(x_n)/f'(x_n)} = x_n + \frac{2}{\dfrac{8x_n^3}{(x^4 - t)} - 3/x_n}$$

For $p = 2$,

$$(1/f)^{(3)} = \left(\frac{2(f')^2 - ff''}{f^3}\right)^{(1)} = \frac{(3f'f'' - ff''')}{f^3} + \frac{(-3)(2(f')^2 - ff'')f'}{f^4} = \frac{6ff'f'' - f^2f''' - 6(f')^3}{f^4}$$

So,

$$\begin{aligned}\frac{(1/f)^{(2)}}{(1/f)^{(3)}} &= \left(\frac{2(f')^2 - ff''}{f^3}\right)\left(\frac{f^4}{6ff'f'' - f^2f''' - 6(f')^3}\right) = \frac{2f'^2/f - f''}{6f'f''/f - f''' - 6(f')^3/f^2} \\ &= \frac{2(4x^3)^2/(x^4 - t) - (12x^2)}{6(4x^3)(12x^2)/(x^4 - t) - (24x) - 6(4x^3)^3/(x^4 - t)^2}\end{aligned}$$

So the iterate is

$$x_{n+1} = x_n + \frac{3\big(2(4x_n^3)^2/(x_n^4 - t) - (12x_n^2)\big)}{6(4x_n^3)(12x_n^2)/(x_n^4 - t) - (24x_n) - 6(4x_n^3)^3/(x_n^4 - t)^2}$$