COURSEWORK PROJECT ANSWERS

Question 1.

a)

```
addN(x,n)
takes any number x and a positive intger n>0
returns the value of x*n

if n==1 then
    return x
else
    return x+addN(x,n-1)
endif
```

b)

```
power(x,n)
takes any number x and integer n≥0
returns the value of x^n

if n==0 then
    return 1
else
    return addN(power(x,n-1),x)
endif
```

c)

```
power(4,3); x=4, n=3, n!=0
addN(power(4,2),4)
power(4,2); x=4, n=2, n!=0
addN(power(4,1),4)
power(4,1); x=4, n=1, n!=0
addN(power(4,0),4)
power(4,0); x=4, n=0, base case!
```

```
STOP!
addN(16,4)=16+16+16+16=64

addN(4,4)=4+4+4+4=16

addN(1,4)=4
power(4,0)=1
```

Question 2.

a)

```
concat(L1,L2)
takes two lists
returns concatenation L1+L2
------------------------------------------------
1. if isEmpty(L1) then
2.     return L2
3. else
4.     return cons(head(L1),concat(tail(L1),L2))
5. endif
```

b)

```
subset(L1,L2)
takes two lists
returns TRUE if all the elements of L1 are in L2
------------------------------------------------
1. if length(L1)>length(L2) then
2.     return FALSE
3. else
4.     if isEmpty(L1) then
5.         return TRUE
6.     elseif !(linSearch(head(L1),L2)) //can also use binSearch()
7.         return FALSE
8.     else
9.         return subset(tail(L1),L2)
10.    endif
11. endif
```

c)

```
L1=[1,5,2]; L2=[4,5,1,0,2,9]
Line1:FALSE; Line4:FALSE; Line6:FALSE; GOTO Line 9
L1=[5,2]; L2=[4,5,1,0,2,9]
Line1:FALSE; Line4:FALSE; Line6:FALSE; GOTO Line 9
L1=[2]; L2=[4,5,1,0,2,9]
Line1:FALSE; Line4:FALSE; Line6:FALSE; GOTO Line 9
L1=[]; L2=[4,5,1,0,2,9]
Line1:FALSE; Line4:TRUE → return TRUE
```

Question 3.

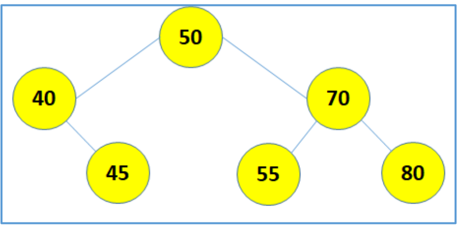
a)

```
level(x,binT) [main]
takes a node value and binary tree
returns a positive integer; the corresponding level of node x
------------------------------------------------------------------
return LHelper(x,binT,0) //call Helper to start at level 0
```

```
LHelper(x,binT,n) [helper]
takes a node value a binary tree and a positive integer n
returns a positive integer (n)
------------------------------------------------------------------
if isLeaf(binT) then
    return 0 //empty BT (base case 1)
elseif x==root(binT) then
    return n //when the node is found (base case 2)
else
    return LHelper(x,left(binT),n+1)+LHelper(x,right(binT),n+1)
endif
```

b)

T:



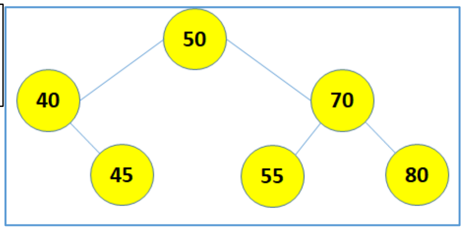
```
                                Level(45,T)
                              LHelper(45,T,0)
               LHelper(45,left(T),1)+LHelper(45,right(T),1)
    0        +LHelper(45,right(T),2)+LHelper(45,left(T),2)+LHelper(45,right(T),2)
---------------------------------------------------------------------------------
   0       +       2       +       0       +       0
                           2
                  Node 45 is in level 2
```

c)

```
level(x,bsT) [main]
takes a node value and BST
returns a positive integer; the corresponding level of node x
-------------------------------------------------------------
return LHelper(x,bsT,0) //call Helper to start at level 0
```

```
LHelper(x,bsT,n) [helper]
takes a node value a BST and a positive integer n
returns a positive integer (n)
-------------------------------------------------------------
if isLeaf(bsT) then
    return 0 //empty BST (base case 1)
elseif x==root(bsT) then
    return n //when the node is found (base case 2)
elseif x<root(bsT)
    return LHelper(x,right(bsT),n+1)
else  //x>root(bsT)
    return LHelper(x,left(bsT),n+1)
endif
```

T:



```
Level(45,T)
LHelper(45,T,0)
LHelper(45,left(T),1)
LHelper(45,right(T),2)
-------------------------------------------------------------
2
Node 45 is in level 2
```

We see that the algorithm for BST requires only 3 recursive calls, i.e. ***O(h).*** But the algorithm for BT makes 6 recursive calls, i.e. ***O(n).***

Question 4.

a)

```
partition(L) [main]
takes a list of numbers
returns three lists

if isEmpty(L) then
    return nil
else
    let PP=cons(head(L),nil) // makes the pivot list
    return pHelper(tail(L),[],PP,[]) //calling helper function
endif
```

```
pHelper(L,LP,PP,RP) [helper]
takes four lists: L,LP,PP,RP
returns the partition of L into LP-PP-RP

if isEmpty(L) then
    return (LP,PP,RP) //base case
elseif head(L)<head(PP) //elements less than the pivot
    let LP=cons(head(L),LP) //constructs the left partition
    return pHelper(tail(L),LP,PP,RP) //recursive call
else                          //elements larger than the pivot
    let RP=cons(head(L),RP) //constructs the right partition
    return pHelper(tail(L),LP,PP,RP) //recursive call
endif
```

b)

```
quickSort(list) [main]
takes a list of numbers
returns a sorted list (ascending)

if isEmpty(list) || isEmpty(tail(list) then
    return list
else
    let [LP,PP,RP]=partition(list) //partitioning the input list
    let Q1=quicksort[LP] //sorting (recursive call)
    let Q2=quicksort[RP] //sorting (recursive call)
    return merge(LP,merge(PP,RP)) //merging the partitions
endif
```

c)

```
L1=[10,20,40,50,45]
[][10][45,50,40,20]
[][10][20,40][45][50]
[][10][][20][40][45][50] base case
merge([],merge([10],merge([],merge([20],merge([40],merge([45],[50]))))))
=[10,20,40,45,50]
You can also use concat() from 2(a) instead of merge()
```

```
L2=[10,9,12,8,15]
[8,9][10][15,12]
[][8][9][10][12][15][] base case
merge([],merge([8],merge([9],merge([10],merge([12],merge([15],[]))))))
=[8,9,10,12,15]
You can also use concat() from 2(a) instead of merge()
```

d) **`L1`** is nearly sorted which is close to the worst case scenario of quicksort. The partitioning order is *linear* rather than *logarithmic*.

Example of a worst case scenario for quicksort is: **`L=[1,2,3,4,5,6,7,8,9].`** Try and see how many times you need to partition the list?