**CELEN086 Final Exam Solutions**

**Semester 1, 2020-21**

1. (a)

Algorithm: myCal(n)
Requires: an integer n (1<= n <= 365), a day number in a year.
Returns: a list of numbers [day, month], representing what day of what month is n

```
1. if n>=1 && n<=186 then //the first 6 months
2.    let day = n mod 31
3.    let month = n/31 //integer-value division
4.    if day == 0 then //end of a month
5.          let day = 31
6.    else
7.          let month = month + 1
8.    endif
9. elseif n>186 && n<=336 then //months 7 to 11
10.   let day = (n-186) mod 30
11.   let month = (n-186)/30
12.   if day == 0 then //end of a month
13.         let day = 30
14.   else
15.         let month = month + 1
16.   endif
17. else // if n>336, n<=365
18.   let day = n - 336
19.   let month = 12 //last month
20. endif
21. return cons(day,cons(month,Nil)) //makes the output list
```

**Marks:** **Algorithm Header- 2marks**
**Body- 4marks (1mark for each if-block and 1mark for line 21)**
**Comments are not necessary.**

(b)

Algorithm: isLeap(yr)
Requires: a positive integer (yr) as year
Returns: a Boolean value, True if year is leap and False otherwise.

```
1. if (yr mod 4 ==0) && (yr mod 100 !=0) || (yr mod 400  ==0) then
2.    return True
3. else
4.    return False
5. endif
```

The result of `isLeap(1964)=True,` since 1964 is divisible by 4.

**Marks:** **Algorithm Header- 1mark**
**Body- 2marks (line 1 in the body is crucial)**
**Result of isLeap(1964)- 1mark**

2. (a)

(i)

Algorithm: collatz(n)
Requires: a positive integer n
Returns: 1 regardless of what the initial n is

```
1. if n == 1 then
2.    return 1 //base case
3. elseif n mod 2 == 0 then // n is even
4.    return collatz(n/2)
5. else // n is odd
6.    return collatz(3*n+1)
7. endif
```

(ii)

```
collatz(10)=collatz(5)=collatz(16)=collatz(8)=collatz(4)=
=collatz(2)=collatz(1)=1 (algorithm stops)
```

**Marks:** **Algorithm Header- 1mark (the 1st two lines are sufficient)**
**Body- 3marks (1mark for lines 2, 4 and 6)**
**Collatz(10) sequence generation- 1mark**

(b)

(i)

Algorithm: listMax(L) [this is our main algorithm]
Requires: a list L of nonnegative integers
Returns: the maximum value of the list L

```
1. return listMaxHelper(0,L)
```

Algorithm: listMaxHelper(mxVal,List)
Requires: takes a number mxVal and a list of integers List
Returns: the maximum value in the List. Returns 0 if List is empty.

```
1. if isEmpty(List) then
2.    return mxVal //returns 0 since mxVal is initialized to 0 in Main.
3. elseif head(List) > mxVal then
4.    return listMaxHelper(head(List),tail(List))//swap mxVal with head
5. else
6.    return listMaxHelper(mxVal,tail(List))//mxVal remains largest
7. endif
```

(ii)

```
L=[10,8,12]
In main call listMaxHelper(0,[10,8,12])
In Helper function we have:
     mxVal=0; head(L)=10; 10>0; so calls listMaxHelper(10,[8,12])
     mxVal=10; head(L)=8; 8<10; so calls listMaxHelper(10,[12])
     mxVal=10; head(L)=12; 12>10; so calls ListMaxHelper(12,[])
Base-case: mxVal=12 and L=[]; so return mxVal=12.
In main returns 12, i.e. max([10,8,12]).
```

**Marks:** **Helper Algorithm Header- 1mark**
**Helper Body- 3marks**
**Trace for List- 1mark (slightest error in tracing will result in nil marks)**

3. (a)

Algorithm: mySqrt(n) [main algorithm]
Requires: a positive integer n>1
Returns: a positive integer m such that m*m <= n

```
1. return mySqrtHelper(n-1,n) //m in helper is set to start at n-1.
```

Algorithm: mySqrtHelper(m,n)
Requires: two positive integers m and n
Returns: a positive integer m such that m*m <= n

```
1. if (m*m) <= n then
2.    return m //base case
3. else
4.    return mySqrtHelper(m-1,n) //recursive statement
5. endif
```

**Marks:** **Line1 in Main Algorithm- 1mark (or correct equivalent)**
**Helper Header- 1mark**
**Helper Body- 2marks**

(b)

(i)
Algorithm: isPrime(p) [main]
Requires: a positive integer p>1
Returns: a Boolean value True if p is prime and False otherwise.

```
1. let m=mySqrt(p)
2. return isPrimeHelper(m,p)
```

Algorithm: isPrimeHelper(m,n)
Requires: two positive integers m>0 and n>1 and n>m
Returns: a Boolean value True if n is prime, False otherwise

```
1. if m == 1 then
2.    return True   //base case when n is prime
3. elseif n mod m == 0 then
4.    return False  //base case when n isn't prime
5. else
6.    return isPrimeHelper(m-1,n) //recursive call
7. endif
```

(ii)

```
isPrime(19); in main m=mySqrt(19)=4
calls isPrimeHelper(4,19)
in Helper Function the recursive call will run 3 times:
isPrimeHelper(3,19); isPrimeHelper(2,19); isPrimeHelper(1,19);
i.e. m==1 hence returns True and stops. So 19 is prime.
```

**Students must be aware that they need a helper function, i.e. isPrimeHelper(m,n).**

**Marks:** **Initializing lines in Main- 1mark**
**Helper Function Header- 1mark**
**Helper Body- 3marks**
**Tracing for 19- 1mark**

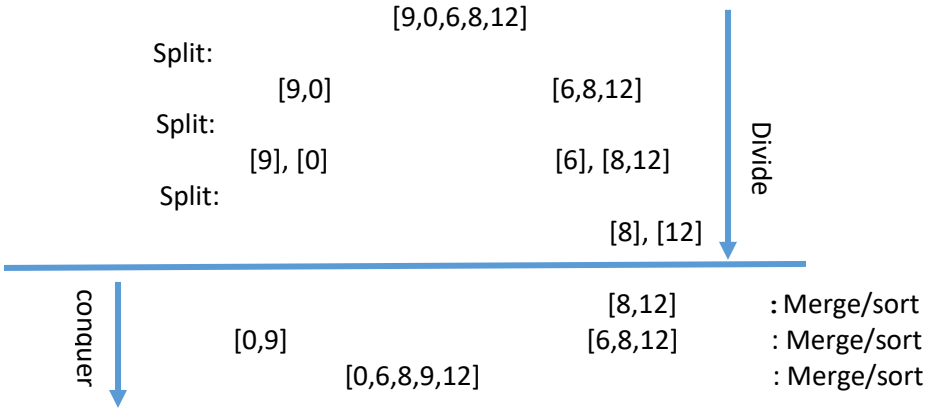
4. (a)

Algorithm: product(List1,List2)
Requires: two nonempty lists of the same length, List1 and List2
Returns: a nonempty list whose elements are the product of similar elements of List1 and List2.

```
1. if isEmpty(tail(List1)) && isEmpty(tail(List2)) then
2.    let K=head(List1)*head(List2) //product of single element lists
3.    return cons(K,Nil) //builds a list
4. else
5.    let K=head(List1)*head(List2)
6.    return cons(K,product(tail(List1),tail(List2))) //recursive call
7. endif
```

**Marks:** **Header- 1mark**
**Body- 3marks**

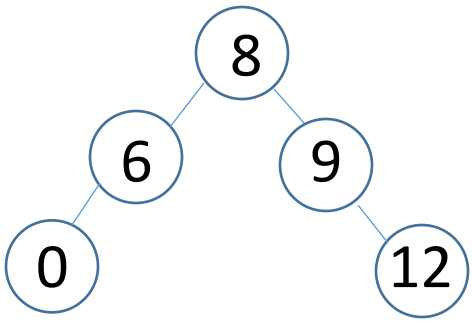
(b)




**Marks:** **Divide- 2marks**
**Conquer- 2marks**

(c)

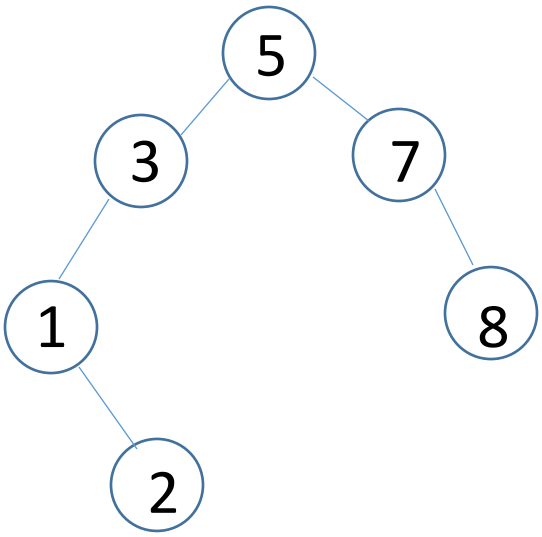
Sorted list is [0,6,8,9,12] which gives the BST below:




**Marks:** **Identifying Root- 1mark**

**Correct Balancing- 1mark**

5. (a)




(b)

It is BST since all the node values of the left subtree are less than the root and all the values of the right subtree are greater than the root.

(c)

The depth=4 since there are 4 generations (levels) that can be seen on this tree.

(d)

The node with value **0** goes to the left-side of 1 and the node with value **10** goes to the right side of 8, maintaining the tree's original depth of 4.

(e)

Inorder traversal or LNR (Left-Node-Right).

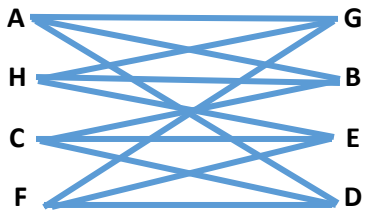
1. [2,4,6]
2. [2,4,6,8]
3. [2,4,6,8,10,12,13]
4. [2,4,6,8,10,12,13,14]
5. [2,4,6,8,10,12,13,14,18]

6. (a)

(i)

The graph is bipartite since we don't see any triangular cycles.

(ii)



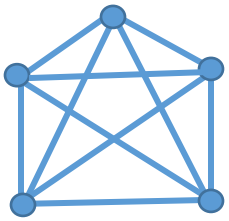

(b)

(i)

For a graph with n vertices to be complete there must be n*(n-1)/2 edges and at each vertex must have n-1 edges coinciding.

(ii)

Each vertex has 4=5-1 edges coincident and the total number of edges are 10 = 5*4/2.



(c)

Graph (a) **has Eulerian path since it has two odd-degree vertices (both deg=3)** as we can start a path in one of the odd-degree vertices and end in the other one visiting all other edges and nodes. Other nodes have even degree.

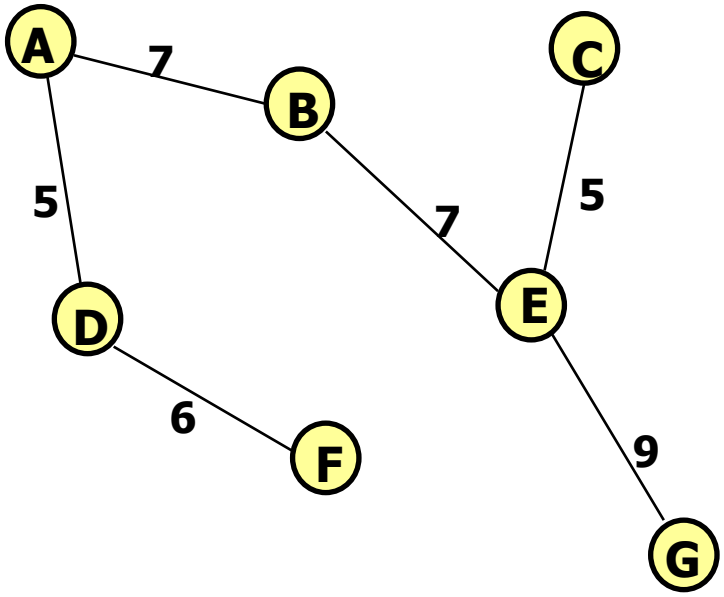
Graph (b) **has Eulerian path since it has two odd-degree vertices (deg=3, deg=5)** as we can start a path in one of the odd-degree vertices and end in the other one visiting all other edges and nodes. Other nodes have even degree.

Graph (c) **has an Eulerian tour since all the nodes have even degrees.**

(d)

(i)

We start by selecting the least costing edge. Either AD or CE with cost=5. Let's select AD. The next least costing is CE with cost=5 so we then select CE. The next least costing is DF with cost=6 so we select DF. The next least coting edges are AB and BE with cost=7. We can select them both since they don't form a cycle. The next least costing edges are CB and EF but selecting either one will result in a closed cycle. So we move to the next least costing path, which are DB and EG each costing 9. We can only choose EG since choosing DB will result in a closed cycle. We have covered all vertices. So we get the following minimal spanning tree:




(ii)

The minimum cost = 5+5+6+7+7+9 = 39