1.

```
Algorithm: isEven(n)
Requires: one positive integer number n
Returns: whether the number is even (True/False)
1:  if n==0
2:      return True
3:  elseif n==1
4:      return False
5:  else
6:      return isEven(n-2)
9:  endif
```

Trace isEven(9):

1. isEven(9): return isEven(7)
2. isEven(7): return isEven(5)
3. isEven(5): return isEven(3)
4. isEven(3): return isEven(1)
5. isEven(1): return False.

2.

```
Algorithm: sumDigits(n)
Requires: one positive integer number n
Returns: the sum of all digits of n
1:  if n/10==0
2:      return n%10
3:  else
4:      return n%10+sumDigits(n/10)
5:  endif
```

Trace sumDigits(1942):

1. 1942/10==0, False, return 2+sumDigits(194):	2+14=16
2. 194/10==0, False, return 4+sumDigits(19):	4+10=14
3. 19/10==0, False, return 9+sumDigits(1):	9+1=10
4. 1/10==0, True, return 1	(backtrack)

3.

```
Algorithm: numDigits(n)
Requires: one positive integer number n
Returns: number of digits of n
1:  if n/10==0
2:      return 1
3:  else
4:      return 1+numDigits(n/10)
5:  endif
```

Trace numDigits(1942):

1. 1942/10==0, False, return 1+numDigits(194):	1+3=4

2. 194/10==0, False, return 1+numDigits(19): 1+2=3
3. 19/10==0, False, return 1+numDigits(1): 1+1=2
4. 1/10==0, True, return 1 (backtrack)

4. gcd(2022,12345) = 3; gcd(924,198) = 6; gcd(234,385) = 1

   Examples of tracing
   gcd(234,385):
   1. Line 1: 234==385, False; Line 3: 234<385, True; return gcd(234, 151)
   2. Line 1: 234==151, False; Line 3: 234<151, False; return gcd(83, 151)
   3. Line 1: 83==151, False; Line 3: 83<151, True; return gcd(83, 68)
   4. Line 1: 83==68, False; Line 3: 83<68, False; return gcd(15, 68)
   5. Line 1: 15==68, False; Line 3: 15<68, True; return gcd(15, 53)
   6. Line 1: 15==53, False; Line 3: 15<53, True; return gcd(15, 38)
   7. Line 1: 15==38, False; Line 3: 15<38, True; return gcd(15, 23)
   8. Line 1: 15==23, False; Line 3: 15<23, True; return gcd(15, 8)
   9. Line 1: 15==8, False; Line 3: 15<8, False; return gcd(7, 8)
   10. Line 1: 7==8, False; Line 3: 7<8, True; return gcd(7, 1)
   11. Line 1: 7==1, False; Line 3: 7<1, False; return gcd(6, 1)
   12. Line 1: 6==1, False; Line 3: 6<1, False; return gcd(5, 1)
   13. Line 1: 5==1, False; Line 3: 5<1, False; return gcd(4, 1)
   14. Line 1: 4==1, False; Line 3: 4<1, False; return gcd(3, 1)
   15. Line 1: 3==1, False; Line 3: 3<1, False; return gcd(2, 1)
   16. Line 1: 2==1, False; Line 3: 2<1, False; return gcd(1, 1)
   17. Line 1: 1==1, True; return 1

   Euclid(234,385)
   1. Line 1: 385==0, False; return Euclid(385, 234)
   2. Line 1: 234==0, False; return Euclid(234, 151)
   3. Line 1: 151==0, False; return Euclid(151, 83)
   4. Line 1: 83==0, False; return Euclid(83, 68)
   5. Line 1: 68==0, False; return Euclid(68, 15)
   6. Line 1: 15==0, False; return Euclid(15, 8)
   7. Line 1: 8==0, False; return Euclid(8, 7)
   8. Line 1: 7==0, False; return Euclid(7, 1)
   9. Line 1: 1==0, False; return Euclid(1, 0)
   10. Line 1: 0==0, False; return 1

   Note: no need for backtracking, as both are tail recursion. We can also see the improved Euclid algorithm is more efficient (there are less steps) compared to the other.

5. Refer to Seminar 3 example.

6.

```
Algorithm: fakeLog(x, y)
Requires: two positive integers x, y
Returns: largest integer k such that $x^k \le y$
1:  return fakeLogHelper(x, y, 1)
```

```
Algorithm: fakeLogHelper(x, y, k)
Requires: three positive integers x, y, k
Returns: largest integer k such that $x^k \le y$
1:  if x^k>y
2:      return k-1
3:  else
4:      return fakeLogHelper(x, y, k+1)
5:  endif
```

Example of tracing

fakeLog(3, 28):

1. fakeLogHelper(3, 28, 1): Line 1: 3^1>28, False; Line 4: return fakeLogHelper(3, 28, 2)
2. fakeLogHelper(3, 28, 2): Line 1: 3^2>28, False; Line 4: return fakeLogHelper(3, 28, 3)
3. fakeLogHelper(3, 28, 3): Line 1: 3^3>28, False; Line 4: return fakeLogHelper(3, 28, 4)
4. fakeLogHelper(3, 28, 4): Line 1: 3^4>28, True; Line 2: return 3

Alternative solution without calling helper function:
Header will be the same as the main algorithm above.

```
1.  if x>y
2.      return 0
3.  else
4.      return 1+fakeLog(x, y/x)    // here y/x is the integer division
5.  endif
```

Trace this algorithm to see how it works.

7. (234, 385) are co-prime, because gcd(234,385)=1.

8.

```
Algorithm: twinPrime(p, q)
Requires: two positive integers p, q
Returns: Ture if p, q are twin prime; False if p, q are not
1:  if |p-q|!=2
2:      return False
3:  elseif isPrime(p) && isPrime(q)
4:      return True
5:  endif
```

Note: isPrime(n) algorithm can be referred to as the examples in Lecture 3 and Seminar 3.

i: (59, 61): True.
ii: (127, 129): False. Because 129=3*43, not a prime number.

9. Refer to Lecture 4 example. Change the name F(n) to fib(n) for this question.

10.

```
Algorithm: fibSum(n)
Requires: one positive integer n
Returns: sum of first n terms in Fibonacci sequence

1:  if n==1
2:      return fib(n)
3:  else
4:      return fib(n)+fibSum(n-1)
5:  endif
```

Trace it yourself. Just focus on procedures of this algorithm.
You can call fib(n) for computing the Fibonacci number directly without showing the procedures there.

11.

```
Algorithm: fakeSqrt (n)
Requires: a positive integer n
Returns: returns the largest positive integer $k$ such that $k \times k \leq n$.

1:  return fakeSqrtHelper(n, 1)
```

```
Algorithm: fakeSqrtHelper(n, k)
Requires: two positive integer n, k and n>=k
Returns: returns the largest positive integer $k$ such that $k \times k \leq n$.

1:  if k*k>n
2:      return k-1
3:  else
4:      return fakeSqrtHelper(n, k+1)
5:  endif
```

Trace fakeSqrt(10):
1. Line 1: return fakeSqrtHelper(10, 1);
2. Line 1: 1*1>10, False; Line 4: return fakeSqrtHelper(10, 2);
3. Line 1: 2*2>10, False; Line 4: return fakeSqrtHelper(10, 3);
4. Line 1: 3*3>10, False; Line 4: return fakeSqrtHelper(10, 4);
5. Line 1: 4*4>10, True; Line 2: return 3

Note: a tail recursion in the helper function, so no need for backtracking after obtaining 3 as answer.

12. Idea of designing this algorithm:
    Given a and b, taking multiple of b (b, 2b, 3b, 4b …) until we find a multiple of b that is also divisible by a. This process does have an end, because we know at least a*b is divisible by a.

```
Algorithm: LCM (a, b)      [main]
Requires: two positive integers a and b
Returns: the least common multiple of a and b
1:   return LCMHelper(a, b, b)
```

```
Algorithm: LCMHelper (a, b, k)    [helper]
Requires: three positive integers a, b and k
Returns: returns the least common multiple
1:   if k%a==0
2:       return k
3:   else
4:       return LCMHelper(a, b, k+b)
5:   endif
```

Trace LCM(6, 8)
1. Line 1: return LCFHelper(6, 8, 8).
2. Line 1: 8%6==0, False; Line 4: return LCMHelper(6, 8, 16).
3. Line 1: 16%6==0, False; Line 4: return LCMHelper(6, 8, 24).
4. Line 1: 24%6==0, True; Line 2: return 24

Alternative solution (not a recursive algorithm):
Header will be the same as the main algorithm above.

```
1.  let m = gcd(a,b)
2.  return a*b/m
```

This method requires the mathematical fact that a*b = gcd(a,b)*lcm(a,b).
And of course we are calling the gcd() function as a sub-algorithm here.

Comments:

I. So far, you should be familiar with tracing algorithms for understanding the ideas. In the future, we may exclude the tracing process in the solution sheet and leave it to you for checking through.
II. In particular, if there is a call of helper function, think about why we need it and how it handles those extra variables in recursive calls.