Assignment Coversheet

Faculty of Science and Engineering

MACQUARIE University

UNIT NO:	UNIT NAME:		
COMP333	Algorithm Theory and Design		
FAMILY NAME:		**FIRST NAME:**	
Venter		Hannes	
STUDENT NO:	**CONTACT PHONE NO:**	**TUTOR'S NAME:**	
44908903	0416506844	Daniel	
ASSIGNMENT TITLE:		**TUTORIAL/PRAC DAY:**	**TUTORIAL/PRAC TIME:**
Assignment 2		Monday	1-3
DATE RECEIVED:	**SUBMITTED WORD COUNT:**	**TURNITIN NUMBER:**	**DUE DATE:**
		2431	07/11/2018
	☐ **EXTENSION GRANTED?**	☐ **DOCUMENT ATTACHED?**	

STUDENT DECLARATION:

I certify that:

- This assignment is my own work, based on my personal study and/or research
- I have acknowledged all material and sources used in the preparation of this assignment, including any material generated in the course of my employment
- If this assignment was based on collaborative preparatory work, as approved by the teachers of the unit, I have not submitted substantially the same final version of any material as another student
- Neither the assignment, nor substantial parts of it, have been previously submitted for assessment in this or any other institution
- I have not copied in part, or in whole, or otherwise plagiarised the work of other students
- I have read and I understand the criteria used for assessment.
- The use of any material in this assignment does not infringe the intellectual property / copyright of a third party
- I have kept a copy of my assignment and this coversheet
- I understand that this assignment may undergo electronic detection for plagiarism, and a copy of the assignment may be retained on the database and used to make comparisons with other assignments in future. To ensure confidentiality all personal details would be removed
- I have read and understood the information on plagiarism. For the University's policy in full, please refer to mq.edu.au/academichonesty or Student Information in the Handbook of Undergraduate Studies.

SIGNATURE: .....

DATE: .....

MARKER'S FEEDBACK:	
(Continue overleaf if required)	
MARKERS NAME:	**GRADE/RESULT:**

Question 1:

(a)

`v1.distance <= v2.distance <=…<= vr.distance <= v1.distance+1`

To begin proof that `I3` is an invariant of the main outer `while` loop of `bfs`, it must be shown that `I3` is initially true.

Initially, as the loop is entered, the queue q contains only the source vertex, s. Since the vertex s is the only vertex in q, it is both the head and the tail of the queue. `s.distance` = 0, which means that; `s.distance <= s.distance <= s.distance+1`

This proves that the invariant;

`I3: v1.distance <= v2.distance <=…<= vr.distance <= v1.distance+1`

holds true initially.

This completes the first part of the proof, known as the induction base.

Next is the induction step. Suppose that the condition `I3` is true at the beginning of some (not necessarily the first) loop iteration:

`v1.distance <= v2.distance <=…<= vr.distance <= v1.distance+1`

The actions which follow include:

- the queue, q, has its head, $v1$, removed. (This means $v2$ becomes the new head)
- each unvisited vertex, $(u_1, u_2, u_3, \dots, u_i)$ where i could possibly be 0, adjacent to $v1$, is marked as visited and added to q (u_i becomes the new tail)
- each u has a distance of `v1.distance+1` set, meaning that `u1.distance = u2.distance = u3.distance = ui.distance = v1.distance+1`

Let q' denote the new queue with $v1$ removed and all u adjacent to $v1$ added. Let vr' denote the new vertices added to q'. Then:

`q'=v2,v3,v4,…,vr,vr' and`
`v2.distance <= v3.distance <=…<= vr.distance <= vr'.distance`

Since `I3` was true at the beginning of this loop iteration, it is know that `v1.distance <= v2.distance`, and that `vr.distance <= v1.distance+1`. After this loop iteration it is shown that `vr'.distance == v1.distance`. This thus means that `vr.distance <= vr'.distance`, proving that the invariant remains true after this loop iteration.

(b)

Assume that condition `I4` is valid at the beginning of an iteration of the main `while` loop. We must show that `I4` is again true at the end of the iteration. Consider carefully the situation at the beginning of the iteration. Let d denote the value of `v.distance`, where v is the vertex at the head (front) of the queue q. To do as required, it is enough to show that the value $d + 1$ to be assigned to `u.distance`, for each unvisited vertex u adjacent to v, is equal to the length of a shortest path from s to u.

We shall prove this by contradiction. Suppose that there is some unvisited vertex u adjacent to v for which a shortest path from s to u has length k less than $d + 1$. Then $k \leq d$. Let $p = \langle u_0 = s, u_1, u_2, \dots, u_{k-1}, u_k = u \rangle$ be a shortest path from s to u.

First, $u_k \neq s$ since `s.distance`=0 (shown in proof of `I3`). Thus, $k \geq 1$.

Consider the shortest path from s to v. The adjacent vertex u, is thus the last vertex which is added to this path.

Note that from I3, the distance from s to v is d, meaning that the distance from s to u is $d + 1$. Since this u has a length of k which is $\leq d$, it thus means that u is either before or in conjunction with v and was not previously found. This is a contradiction to both Fact 1 and Fact 2 which disproves the fact that there can exist some unvisited vertex u adjacent to v for which a shortest path from s to u has length k less than $d + 1$. Then $k \leq d$.

Thus, the value $d + 1$ to be assigned to `u.distance`, for each unvisited vertex u adjacent to v, is equal to the length of a shortest path from s to u.

(c)

An Adjacency List would be the best graph representation for the performance of `bfs`. This representation uses an array of lists where the size of the array is equal to the number of vertices in the graph. This is a vast improvement in space complexity over an Adjacency Matrix which consumes the same amount of space even when the number of edges is sparse. An Adjacency Matrix will use $O(n^2)$ space complexity, where as an Adjacency List will only use $O(m)$ space complexity.

The time complexity of `bfs` on an Adjacency List is $O(n+m)$, and the time complexity of an Adjacency Matrix is $O(n^2)$. An Adjacency Matrix must traverse the whole 2D array with lengths n up and across. This is thus why an Adjacency Matrix has $O(n^2)$ time complexity. An Adjacency List has each vertex enqueued and dequeued at most once which is n. Then, each adjacent edge is accessed through moving to the adjacent vertex. Thus, the time complexity of `bfs` on an Adjacency List is $O(n+m)$.

http://web.eecs.utk.edu/~huangj/CS302S04/notes/graph-searching.html

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/graph-and-its-representations/

Question 2:

(a)

$\Theta\left(10^{L(a)-L(b)+1}\right)$ where $a \geq b > 0$.

The number of subtractions performed is equal to the integer quotient of a and b (a/b). The expression provided above is an estimate of the magnitude of this quotient. Having worked over tutorial questions, it is known that $a = 10^{L(a)}$ and $b = 10^{L(b)}$.

Since $\frac{a}{b} = \frac{10^{L(a)}}{10^{L(b)}} = 10^{L(a)-L(b)}$,

This is $\approx \Theta\left(10^{L(a)-L(b)+1}\right)$

(b)

$\Theta(L(a)(L(a) - L(b) + 1))$ where $a \geq b > 0$.

The number of iterations of the main (outer) while loop is equal to the length of the integer quotient of a and b. Running through this, the inner while loop really confused me. I found that the logic of finding $L(a) - L(b) + 1$ is similar to the method is in Question 2 (a). My understanding is that the outer while loop runs for the length of a, thus $L(a)$, and then the inner loop runs $L(a) - L(b) + 1$. This thus means that the algorithm would run $L(a) \times L(a) - L(b) + 1 = \Theta(L(a)(L(a) - L(b) + 1))$.

(c)

It can be obviously seen that the time complexity for executing `a.divide(b)` is far less than the time complexity for executing `a.divideBasic(b)`. This proves the that the MULTIPLY-BY-10 and DIVIDE-BY-10 methods used in `divde()`, is far superior to the ordinary division by subtraction used by `divideBasic()`.

Although the efficiency of `divideBasic()` is based on a estimate of the magnitude for the quotient, it is still far less efficient than the time complexity for `divide()`.

Question 3:

(a)

```
egcd(a,b) {
      //these will be the columns which get multiplied by the
      //quotient as the algorithm progresses
      colA = {1, 0}
      colB = {0, 1}

      while(a and b >= 0) {
            //the quotient to be multiplied into the columns
            q = a / b

            //to find the remainder after the quotient's division
            a = a mod b

            //multiply the quotient by the top item in the columns
            //subtract the multiplied number from the bottom item in
            //the columns
            colA[0] = colA[0] - q*colA[1];
```

```
        colB[0] = colB[0] - q*colB[1];

        //if a has reached 0 after the a mod b, leave the loop
        if(a == 0) {
             break;
        }

        //instead of dividing a by b and then transfering into a
        //temp, i just kept going by dividing b with a after.
        //the quotient to be multiplied into the columns
        q = b / a

        //to find the remainder after the quotient's division
        b = b mod a

        //multiply the quotient by the bottom item in the columns
        //subtract the multiplied number from the top item in the
        //columns. this should now swap the number into the
        //correct order again. this avoids the need to place
        //items into temp variables to swap them
        colA[1] = colA[1] - q*colA[0];
        colB[1] = colB[1] - q*colB[0];

        //if b has reached 0 after the b mod a, leave the loop
        if (b == 0) {
             break;
        }
}

        //if the loop broke because a == 0
        if (a == 0) {
             //b has been reduced down to be the gcd
             result[0] = b;
             //x is in the top item, the corresponding bottom
             //item will be a
             result[1] = colA[1];
             //y is in the top item, the corresponding bottom
             //item will be b
             result[2] = colB[1];
             return result;
        }

        else {//the loop broke because b == 0
             //a has been reduced down to be the gcd
             result[0] = a;
             //x is in the bottom item, the corresponding top
             //item will be a
             result[1] = colA[0];
             //y is in the bottom item, the corresponding top
             //item will be b
             result[2] = colB[0];
             return result;
        }
```

I: a and b > 0. colA and colB will always represent the multiplication and subtraction of the quotient q.

The time complexity will be $\Theta(L(a)(L(a) - L(b) + 1))$ where $d = \gcd(a, b)$. This is because the `divide()` method is the only method used in the BigInt class. This is used to find both the quotient and the remainder. Thus, this `egcd()` has the same time complexity as `divide()`.

(b)

```
minv(a,n) {
      //check that n>1, otherwise inverse is 0
      if (n == 1)
            return 0;

      //call the egcd function to find d,x,y.
      int [] gcd = egcd(a, n);
      //x is the only number needed since its the inverse of a mod n
      int x = gcd[1];

     // Make x positive
      if (x < 0)
         x += n;

     return x;
}
```

The time complexity will be $\Theta(L(a)(L(a) - L(b) + 1))$. This is because this method calls `egcd()` to do all the work. The `divide()` method is the only one used in the BigInt class for `egcd()`. This is used to find both the quotient and the remainder. Thus, this `minv()` has the same time complexity as `egcd()` which has the same time complexity as `divide()`.

(c)

```
cra(p,q,a,b) {
      //initialise c
      int c = 0;

      //see note below***
      int k = (b - a) * (minv(p, q) % q);
      c = a + k * p;

      return c;
}
```

***I used Scott's hand notes on the Chinese Remainder algorithm to find this algorithm. https://ilearn.mq.edu.au/pluginfile.php/5399634/mod_resource/content/1/HandNotesCRA.pdf

The time complexity will be $\Theta(L(a)(L(a) - L(b) + 1))$. This is because this method calls `minv()` to do all the work. Thus, this `cra()` has the same time complexity as `minv()` which has the same time complexity as `egcd()` which has the same time complexity as `divide()`.

Question 4:

(a)

```
modexp(a,b,n) { //a,b,n are all BigInt
      //y is BigInt
      y = 1

      //this is the remainder from mod
      z = a.divide(n)[1];

      //i is BigInt
      i = b;

      //while i is not lessOrEqual (ie. greater than) 0
      while(not i.lessOrEqual(0)) {
            //find i mod 2, to check if its even
            //if the remained not equal to 0, its odd
            if(!i.divide(2)[1].isEqual(0)) {
                  y = z.multiply(y);
                  //y mod n
                  y = y.divide(n)[1];
            }

            //square z
            z = z.multiply(z);

            //z mod n
            z = z.divide(n)[1];

            //find quotient when dividing by 2
            i = i.divide(2)[0];
      }
      return y;
}
```

I used Scott's solutions to tutorial week 10.

https://ilearn.mq.edu.au/pluginfile.php/5397646/mod_resource/content/1/tutorialWeek10Solution.txt

I: b is always > 0.

The time complexity of multiply is $O(L(a)L(b))$, and the time complexity of divide is $O(L(a)(L(a) - L(b) + 1))$. Thus, in Big-Oh terms, this algorithm's complexity is the same as divide; $O(L(a)(L(a) - L(b) + 1))$.

(b)

```
modexpv2(a,b,n)      {
      // reduce b using Fermat's little theorem (Fact 1)
      //subtract 1 from the mod (ie. n-1)
      BigInt nReduced = n.subtract(new BigInt(1));

      //multiply the smaller n by the quotient of b and the
      //smaller n
      BigInt fermat = nReduced.multiply(b.divide(nReduced)[0]);

      //subtract the number achieved from Fermat
      b = b.subtract(fermat);

      //run the algorithm with the reduced b
      return modexp(a, b, n);
}
```

This method reduces the size of b. The largest complexity that gets used is the divide method from BigInt. Thus, in Big-Oh terms, this algorithm's complexity is the same as divide; $O(L(a)(L(a) - L(b) + 1))$. However, in this situation, b is the new reduced b.

(c)

I understand here that where are trying to find x, where $x = a^b \ mod \ pq$. Since p and q are prime, this means that:

$x = a^b \ mod \ p$

$x = a^b \ mod \ q$

Understanding this and trying to implement it are two different things. I tried to compute $a^b \ mod \ p$ and $a^b \ mod \ q$ separately, and then use `cra()` to solve for x. I understand that something is wrong, however I am struggling to implement it.

```
modexpv3(a,b,p,q) {
      //find a^b mod p, same as in modexp, only with ints
      p1 = 1; z = a mod n; i = b;
      while (i > 0) {
            if (i is odd)
                  p1 = z * p1 mod n;
            z = z * z mod n;
            i = i / 2;
      }

      //find a^b mod q, same as in modexp, only with ints
      q1 = 1; z = a mod n; i = b;
      while (i > 0) {
            if (i is odd)
                  p1 = z * p1 mod n;
            z = z * z mod n;
            i = i / 2;
      }

      int c = cra(a,b,p1,q1);

      return c;
}
```

This method hypothetically calls the modexp() function twice. It then calls the cra() method. As such, it would use the time complexity from modexp() + modexp() + cra().

As proven above, the complexity for both functions is $O(L(a)(L(a) - L(b) + 1))$, since they primarily use the divide() function. This method would simply use:

$3 \times (L(a)(L(a) - L(b) + 1) = O(L(a)(L(a) - L(b) + 1))$.

Question 5:

I understand this theoretically and have read over the lecture slides and various other online resources to try and figure this out. Similar to the previous question, I feel that I am close, however after lots of time spent on this, I still have not been able to have it sufficiently working.

```
karatsuba(a,b) {
      //initialise c as a BigInt
      c=0

      //stopping case.
      //n is the number of digits in a||b (length)
      if(n == 1) {
            c = a.multiply(b); //compute double prec. product
            return c;
      }

      //turn a and b into strings
      String aString = a.toString();
```

```
        String bString = b.toString();

        //find the middle of a||b
        int mid = (aString.length())/2;

        //find a substring of the high order part for a
        a1 = aString.substring(0, mid);
        //find a substring of the low order part for a
        a0 = aString.substring(mid);
        //find a substring of the high order part for b
        b1 bString.substring(0, mid);
        //find a substring of the low order part for b
        b0 = bString.substring(mid);

        //add the high and low order parts
        as = a1.add(a0);
        bs = b1.add(b0);

        //recurse over karatsuba 3 times
        BigInt c2 = karatsuba(a1, b1);
        BigInt c0 = karatsuba(a0, b0);
        BigInt cs = karatsuba(as, bs);

        //combine the products
        s = c0.add(c2);
        c1 = cs.subtract(s);

        //find 10^a.length, multiply c2 by it
        aL = power(10, aString.length());
        BigInt cP2 = c2.multiply(new BigInt(aL.toString()));

        //find 10^a.length/2, multiply c1 by it
        aL = power(10, aString.length()/2);
        BigInt cP1 = c1.multiply(new BigInt(aL.toString()));

        //add all the products together
        c = cP2.add(cP1.add(c0));

        return c;
}
```

Finding the time complexity of Karatsuba is very difficult. In this representation, n denotes the length of a in digits.

From the Week 10 lecture slides, it is given that Karatsuba has a time complexity of $\Theta(n^{\log_2 3})$.

When the values of n are small, the time complexity of Karatsuba becomes larger than classical multiplication. This reason is due to the constant of proportionality hidden in the notation $\Theta(\dots)$ which is much larger for Karatsuba than for Classical. Thus, Karatsuba would be more efficient when n is sufficiently large.

Question 6: