Anna Sun, Chen Bai, Shiyu Wang

Algorithm PS 1

1. Given P always halts on any inpit.
   Consider the following program:

```
 Procedure P(x) {
    /* Ignore input and always output true */
    writeln('Yes');
}
```

   Assum Q is a program that

```
 Procedure Q(x) {
   A(x);
   writeln('Yes');
}
```

   P prints Yes for all inputs. If Q = P, then Q prints Yes for all inputs, which impiles A(x) always halts. Therefore Q = P, iff A(x) halts. We have constructed a solution to solve halting problem. Halting problem is undecidable, so there is no equality checking program can conclusively determine whether P = Q.

2. (a) $ax \equiv 1 \pmod m$

   $x \equiv 15^{-1} \pmod{11}$

   $15x \equiv 1 \pmod{11}$

   One anwer can be:

   $15(3) \equiv 45 \equiv 1 \pmod{11}$

   So. the answer is : $3 + 11 * z$ ($z \in Z$)

   (b) According to the formula :

   $a \equiv b \pmod e$ /\ $c \equiv d \pmod e$, then $a * b \equiv c * d \pmod e$

   Because there is 11 in $3 * 5 * 11 * 17 * 23 * 29 * 31 * 47 * 53$

   And $11 \equiv 0 \pmod{11}$

   So $3 * 5 * 11 * 17 * 23 * 29 * 31 * 47 * 53 \equiv 0 \pmod{11}$

   (c) According to the formula :

   $a \equiv b \pmod e$ /\ $c \equiv d \pmod e$, then $a * b \equiv c * d \pmod e$

   $19 \equiv 5 \pmod 7$

   $19^2 \equiv 5^2 \pmod 7 \equiv 4 \pmod 7$

   $19^4 \equiv 4^2 \pmod 7 \equiv 2 \pmod 7$

   $19^8 \equiv 2^2 \pmod 7 \equiv 4 \pmod 7$

   $19^{16} \equiv 2 \pmod 7$

$19^{19} \equiv 19^{16} * 19^{2} * 19 \equiv 2 * 4 * 5 \pmod 7 \equiv 5 \pmod 7$

3. a)

Assume k >= 0, $k \in Z$

Base case: n = 0, k = 0

then $2^{2n} -1 = 3K \Rightarrow 2^{0} -1 = 0$ proved.

Induction:

Assume $2^{2n} -1 = 3K$ is true for any integer n

then we need to prove $2^{2(n+1)} -1 = 3M$ (M > K)

$2^{2(n+1)} -1$

$= 2^{(2n)} * (3 + 1) -1$

$= 2^{2n} -1 + 3 * 2^{2n}$

Because $2^{2n} -1 = 3k$, we need to prove $3 \mid 3 * 2^{2n}$.

Then we need to prove $2^{2n} \in Z$

Since n is integer so $2^{2n} \in Z$, proved.

b)

Base case: n = 1,

$1 \leq 2\sqrt{1}$

Induction:

Assume case n:

$1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + ... + \frac{1}{\sqrt{n}} \leq 2\sqrt{n}$ is true for n >= 1

Then case n+1, we need prove:

$1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + ... + \frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n+1}} \leq 2\sqrt{n+1}$,

since we already know :

$1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + ... + \frac{1}{\sqrt{n}} \leq 2\sqrt{n}$,

so

$1 + \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{3}} + ... + \frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n+1}} \leq 2\sqrt{n} + \frac{1}{\sqrt{n+1}}$;

then we need to prove :

$2\sqrt{n} + \frac{1}{\sqrt{n+1}} \leq 2\sqrt{n+1}$;

equals to prove :

$\frac{1}{\sqrt{n+1}} \leq 2\sqrt{n+1} - 2\sqrt{n}$;

$\Leftarrow \quad 1 \leq (2\sqrt{n+1} - 2\sqrt{n}) \times \sqrt{n+1}$

$= 2(n + 1) - 2\sqrt{n} \times \sqrt{n+1}$

$\leq 2(n + 1) - 2\sqrt{n} \times \sqrt{n}$

$= 2$,

which is obviously true.

so we can prove case n+1 is ture, if given case n is true.

Proved.

c)

To Prove $\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$

Base case: n = k = 0

$\sum_{i=0}^{0} \binom{0}{0} = \binom{1}{1} = 1,$

Assume case n:

$\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$ is true,

Then case n+1, we need prove:

$\sum_{i=k}^{n+1} \binom{i}{k} = \binom{n+2}{k+1},$

we know

$$\sum_{i=k}^{n+1} \binom{i}{k} = \sum_{i=k}^{n} \binom{i}{k} + \binom{n+1}{k},$$
$$= \binom{n+1}{k+1} + \binom{n+1}{k},$$
$$= \binom{n+2}{k+1}$$

Proved.

4. Ascending:

$g_{20} < g_4 < g_{10} < g_{15} = g_{16} < g_{14} < g_{13} < g_6 < g_3 < g_1 < g_{19} < g_5 < g_7 < g_8 < g_9 = g_2 < g_{18} < g_{17} < g_{11} < g_{12}$

5. a)False

Suppose $f(n) = 2 * n$ and $g(n) = n$

This holds under the assumption that $f(n)$ is $O(g(n)!)$ for all n

$2 * n < C * n$ for any constant $C > 2$

However, $f(n)! = 2^n * n!$ is not in $O(n!)$ in the case if $f(n) = 2 * n$ and $g(n) = n$

Because $2^n * n! \gg C * n! \Rightarrow 1 > C * 0$ for any $n > 0$, and any constant C

Thus, $f(n)! > O(g(n!))$

b)True

By assumption, $\exists\, n_0 \in N$ and $c \in R > 0$

s.t. for all $n \in N$ with $n \geq n_0$, we have $0 \leq f(n) \leq c * g(n)$

But then, since rooting is order-preserving (on positive values),

also, $0 = 0^{\frac{1}{2}} \leq \sqrt{f(n)} \leq \sqrt{c * g(n)} \leq \sqrt{c} * \sqrt{g(n)}$

Thus, $\sqrt{f(n)} \in O(\sqrt{g(n)}$

c)False

Suppose $f(n) = 2*n$ and $g(n) = n$

This holds under the assumption that $f(n)$ is $O(g(n))$ for all n

$2*n < C*n$ for any constant $C > 2$

However, e^{2n} is not in O(e^n). In the case, if $f(n) = 2*n$ and $g(n) = n$

Because $e^{2n} \gg c*e^n \Rightarrow 1 > C*0$ for any $n > 0$ and any constant C

Thus, $e^{f(n)} > O(e^{g(n)})$

6.

a) $T(n)$ = 4T($\frac{n}{2}$) + $\frac{n}{log(n)}$

Consider a = 4, b = 2 , which means $log_b a = 2$

c < $log_b a = 2$

$f(n) = n/log(n)$

so, $f(n) = O(n^c),\ 0 < c < 1$

Thus we have T(n) = Θ(n^2)

b) $T(n) = \sqrt{n}T(\sqrt{n}) + n \Rightarrow \frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 1$

Pick $S(n) = \frac{T(n)}{n}$. The recurrence becomes $S(n) = S(\sqrt{n}) + 1$

The solution of this recurrence is $S(n) = \Theta(loglog(n))$

Therefore, $T(n) = \Theta(nloglog(n))$

c) Assume $T(1) = 1$, then unroll the recurrence

$$
\begin{aligned}
T(n) &= T(n-1) + \tfrac{1}{n} \\
&= \tfrac{1}{n} + \tfrac{1}{n-1} + \tfrac{1}{n-2} + \cdots + \tfrac{1}{2} + \tfrac{1}{T(1)} \\
&= \textstyle\sum_{k=1}^{n} \tfrac{1}{k}
\end{aligned}
$$

We can bound this sum using integrals

$$T(n) = \sum_{k=1}^{n} \tfrac{1}{k} \geq \int_{1}^{n+1} \tfrac{1}{x}dx = ln(n+1) \geq ln(n) = \Omega(ln(n))$$

$$T(n) = \sum_{k=1}^{n} \tfrac{1}{k} = 1 + \sum_{k=2}^{n} \tfrac{1}{k} \leq 1 + \int_{1}^{n} \tfrac{1}{x}dx = 1 + ln(n) = O(ln(n))$$

Thus, $T(n) = \Theta(ln(n)) = \Theta(log(n))$

7. a)The time complexity of multiply two n-digit numbers using long multiplication is $\Theta(n^2)$

Code in C:

```
#include<stdio.h>
```

```
#include<string.h>

/* function declaration */
void calc1(char* str1,int len1,int* tmp,int m);
void accumulate(int cnt,int* res,int res_len,int* tmp,int tmp_len);
char* bignum_multi(char* str1,int len1,char* str2,int len2,char* result,int len);

int main()
{
        int i,j,m1,m2;
        int len,len1,len2;
        char *str1,*str2;
        char* result;
   /* get the calculation data */
        printf("input multiplicand: ");
        gets(str1);
        printf("input multiplier: ");
        gets(str2);

        /* calculate the length of two strings and the storage space for the result */
        len1=strlen(str1);
        len2=strlen(str2);
   len=len1+len2;

   /* initialize the string array */
   result=(char*)malloc(len*sizeof(char));
   for(i=0;i<len;i++)
                *(result+i)='0';

   /* calculate and output results */
   printf("The result is: %s\n",bignum_multi(str1,len1,str2,len2,result,len));
   free(result);
   system("pause");
   return 0;
}

/*==========================================================================
call functions calc1&accumulate to complete long multiplication
==========================================================================*/
char* bignum_multi(char* str1,int len1,char* str2,int len2,char* result,int len)
{
        int i,j,m=0,cnt=0,*tmp,*res;
```

```
        /* allocate storage space for temporary result */
    tmp=(int*)malloc((len1+1)*sizeof(int));
    res=(int*)malloc(len*sizeof(int));

    /* initialize two arrays */
    for(i=0;i<len1;i++)
    tmp[i]=0;
    for(j=0;j<len;j++)
    res[j]=0;

    for(i=len2-1;i>=0;i--)
        {
            /* get the i-th digit of multiplier */
            m=str2[i]-'0';
            /* store the product of multiplicand and m into tmp */
            calc1(str1,len1,tmp,m);
            /* store the values of tmp into res */
            cnt++;
            accumulate(cnt,res,len,tmp,len1+1);
        }

        /* convert the values in res to string and store the string into result */
    i=0;j=0;
    /* remove those zeros before the first non-zero digit in res */
    while(res[i++]==0);

    for(m=i-1;m<len;m++,j++)
            result[j]=res[m]+0x30;
        result[j]='\0';

    free(tmp);
    free(res);
    return result;
}

/*==========================================================================
Calculate the product of the multipland and one digit of the multiplier
==========================================================================*/
void calc1(char* str1,int len1,int* tmp,int m)
{
        /* d: the product of two digits */
    int i,d=0,remainder=0,carry=0;
    /* count from the character before the string of multiplicand '\0'  */
```

```
    for(i=len1-1;i>=0;i--)
        {
                d=str1[i]-'0';
                d*=m;
                remainder=(d+carry)%10;
                carry=(d+carry)/10;
                tmp[i+1]=remainder;
        }

        if(carry)
                tmp[0]=carry;
        else
                tmp[0]=0;
}
/*==============================================================================
Put the product of the multiplicand and a digit of the multiplier into the res array
==============================================================================*/
void accumulate(int cnt,int* res,int len,int* tmp,int len1)
{
        int m=0,n=0,i,k,remainder=0;
        static int carry=0;
        for(k=len1-1,i=0;k>=0;k--,i++)
        {
                m=tmp[k];
                n=res[len-cnt-i];
                if(m+n+carry>=10)
                {
                        remainder=(m+n+carry)%10;
                        carry=1;
                }
                else
                {
                        remainder=m+n+carry;
                        carry=0;
                }
                res[len-cnt-i]=remainder;
        }
}
```

b)The time complexity of multiply two n-digit numbers using karatsuba algorithm is $\Theta(n^{\log_2 3})$

Code in C++:

```
#include "stdafx.h"
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

//string -> int
int string_to_num(string k)
{
    int back;
    stringstream instr(k);

    instr>>back;
    return back;
}

//int -> string
string num_to_string(int intValue)
{
        string result;
        stringstream stream;
        stream << intValue;
        stream >> result;
        return result;
}

//add zeros before the string str
string stringBeforeZero(string str,int s)
{
        for(int i=0;i<s;i++)
        {
                str.insert(0,"0");
        }
        return str;
}

//string addition
string stringAddstring(string str1,string str2)
{
        //add zeros to the short string, so str1&str2 have the same length
        if (str1.size() > str2.size())
```

```
        {
                str2 = stringBeforeZero(str2,str1.size() - str2.size());
        }
        else
                if (str1.size() < str2.size())
                {
                        str1 = stringBeforeZero(str1,str2.size() - str1.size());
                }

    string result;
        int flag=0;//0 means no-carry; 1 means carry
        for(int i=str1.size()-1;i>=0;i--)
        {
                int c = (str1[i] - '0') + (str2[i] - '0') + flag;
      flag = c/10;//if c >10, set flag to 1, else 0;
      c %= 10;

                result.insert(0,num_to_string(c));//insert the new character to the front of result
    }
    if (0 != flag) //if the highest significant digit need carry, add one more character
    {
                result.insert(0,num_to_string(flag));
        }

    return result;
}


//string substraction
string stringSubtractstring(string str1,string str2)
{
        //remove invalid zeros
        while ('0' == str1[0]&&str1.size()>1)
        {
                str1=str1.substr(1,str1.size()-1);
        }

        while ('0' == str2[0]&&str2.size()>1)
        {
                str2=str2.substr(1,str2.size()-1);
        }

        //str1&str2 have the same length by adding zeros
```

```
	if (str1.size() > str2.size())
	{
		str2 = stringBeforeZero(str2,str1.size() - str2.size());
	}

	string result;
	for(int i=str1.size()-1;i>=0;i--)
	{
		int c = (str1[i] - '0') - (str2[i] - '0');
       if (c < 0) //borrowing
       {
			c +=10;
			int prePos = i-1;
			char preChar = str1[prePos];
			while ('0' == preChar)
			{
				str1[prePos]='9';
				prePos -= 1;
				preChar = str1[prePos];
			}
			str1[prePos]-=1;
		}

		result.insert(0,num_to_string(c));//insert the new character to the front of result
	}
	return result;
}


//add zeros at the end of str
string stringFollowZero(string str,int s)
{
	for(int i=0;i<s;i++)
	{
		str.insert(str.size(),"0");
	}
	return str;
}

//divide and conquer
string IntMult(string x,string y)
{
	//remove invalid zeros
```

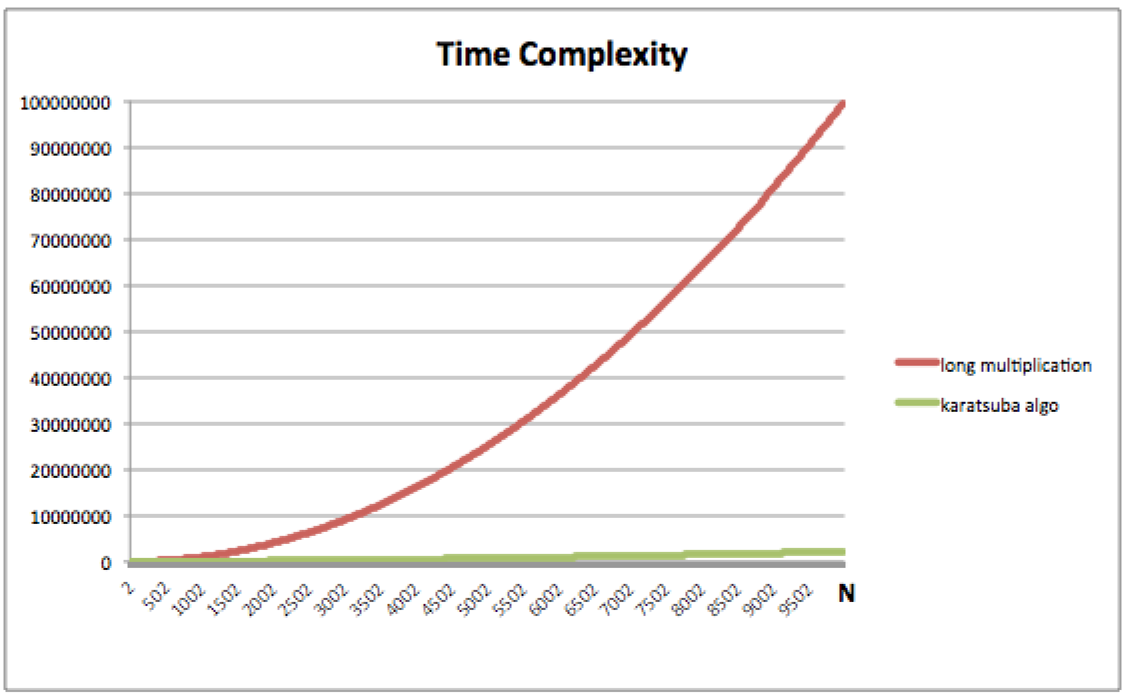
```
while ('0' == x[0]&&x.size()>1)
	{
		x=x.substr(1,x.size()-1);
	}

	//remove invalid zeros
	while ('0' == y[0]&&y.size()>1)
	{
		y=y.substr(1,y.size()-1);
	}
	/*f is used to deal with two conditions:
	  the length of two strings are diffirent;
	  the length of strings is not the exponential times of 2*/
	int f=4;

	if (x.size()>2 || y.size()>2)
	{
		if (x.size() >= y.size())
		{
			while (x.size()>f)
			{
				f*=2;
			}
			if (x.size() != f)
			{
				x = stringBeforeZero(x,f-x.size());
				y = stringBeforeZero(y,f-y.size());
			}
		}
		else
		{
			while (y.size()>f)
			{
				f*=2;
			}
			if (y.size() != f)
			{
				x = stringBeforeZero(x,f-x.size());
				y = stringBeforeZero(y,f-y.size());
			}
		}
	}
```

```
	if (1 == x.size()) //add 1 zero when the x.size is 1
	{
		x=stringBeforeZero(x,1);
	}
	if (1 == y.size()) //add 1 zero when the y.size is 1
	{
		y=stringBeforeZero(y,1);
	}

	//make sure the two strings have the same length
	if (x.size() > y.size())
	{
		y = stringBeforeZero(y,x.size()-y.size());
	}
	if (x.size() < y.size())
	{
		x = stringBeforeZero(x,y.size()-x.size());
	}

int s = x.size();
string a1,a0,b1,b0;

if( s > 1)
{
   a1 = x.substr(0,s/2);

   a0 = x.substr(s/2,s-1);

   b1 = y.substr(0,s/2);

   b0 = y.substr(s/2,s-1);
}

	string result;

if( s == 2) //the end condition of recursion
{
		int na = string_to_num(a1);
   int nb = string_to_num(a0);
   int nc = string_to_num(b1);
   int nd = string_to_num(b0);
   result = num_to_string((na*10+nb) * (nc*10+nd));
}
```

```
    else{
/*==========================================================================
tips:
c = a*b = c2*(10^n) + c1*(10^(n/2)) + c0;
a = a1a0 = a1*(10^(n/2)) + a0;
b = b1b0 = b1*(10^(n/2)) + b0;
c2 = a1*b1;  c0 = a0*b0;
c1 = (a1 + a0)*(b1 + b0)-(c2 + c0);
==========================================================================*/
                string c2 = IntMult(a1,b1);// (a1 * b1)

                string c0 = IntMult(a0,b0);// (a0 * b0)

                string c1_1 = stringAddstring(a1,a0);// (a1 + a0)

                string c1_2 = stringAddstring(b1,b0);// (b1 + b0)

                string c1_3 = IntMult(c1_1,c1_2);// (a1 + a0)*(b1 + b0)

                string c1_4 = stringAddstring(c2,c0);// (c2 + c0)

                string c1=stringSubtractstring(c1_3,c1_4);// (a1 + a0)*(b1 + b0)-(c2 + c0)

                string s1=stringFollowZero(c1,s/2);// c1*(10^(n/2))

                string s2=stringFollowZero(c2,s);// c2*(10^n)

                result = stringAddstring(stringAddstring(s2,s1),c0);// c2*(10^n) + c1*(10^(n/2)) +
c0
        }

        return result;
}

void main()
{
        string A,B,C,D;
        string num1,num2;
        string r;

        cout<<"input multiplicand: ";
    cin>>num1;
```

```
        cout<<"input multiplier:";
        cin>>num2;

        r=IntMult(num1,num2);
    //remove invalid zeros
    while ('0' == r[0]&&r.size()>1)
        {
            r=r.substr(1,r.size()-1);
        }

        cout<<"the result: "<<endl;
        cout<<num1<<" "<<"*"<<" "<<num2<<" "<<"="<<" "<<r<<endl<<endl;
}
```

c)



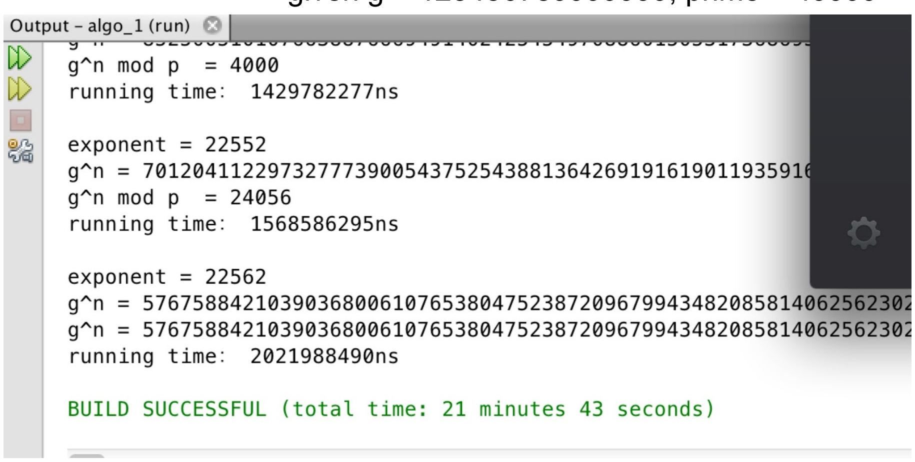
8.

a. Code in Java:

```
public class Q8_1_NEW {
```

```java
    public static void main(String[] args) {
        BigInteger prime= new BigInteger("49999");
        BigInteger generator= new BigInteger("123456789999999");
         int exponent = 2;
        long startTime = 0;
        long endTime = 0;

        while ((endTime - startTime) < 2000000000){
            exponent = exponent + 10;
            System.out.println("exponent = " + exponent + "  ");
            // start time
            startTime=System.nanoTime();

             // g ^ n
             BigInteger g_n = generator.pow(exponent);
             System.out.println("g^n = "+ g_n);
            // g ^ n mod p
            BigInteger g_n_p = g_n.mod(prime);
            System.out.println("g^n mod p  = "+ g_n_p);

            // end time
             endTime=System.nanoTime();
             System.out.println("running time：  "+(endTime-startTime)+"ns\n");
        }
    }
}
```

Running result: largest n to be 22562 within 2.0s
given g = 123456789999999, prime = 49999

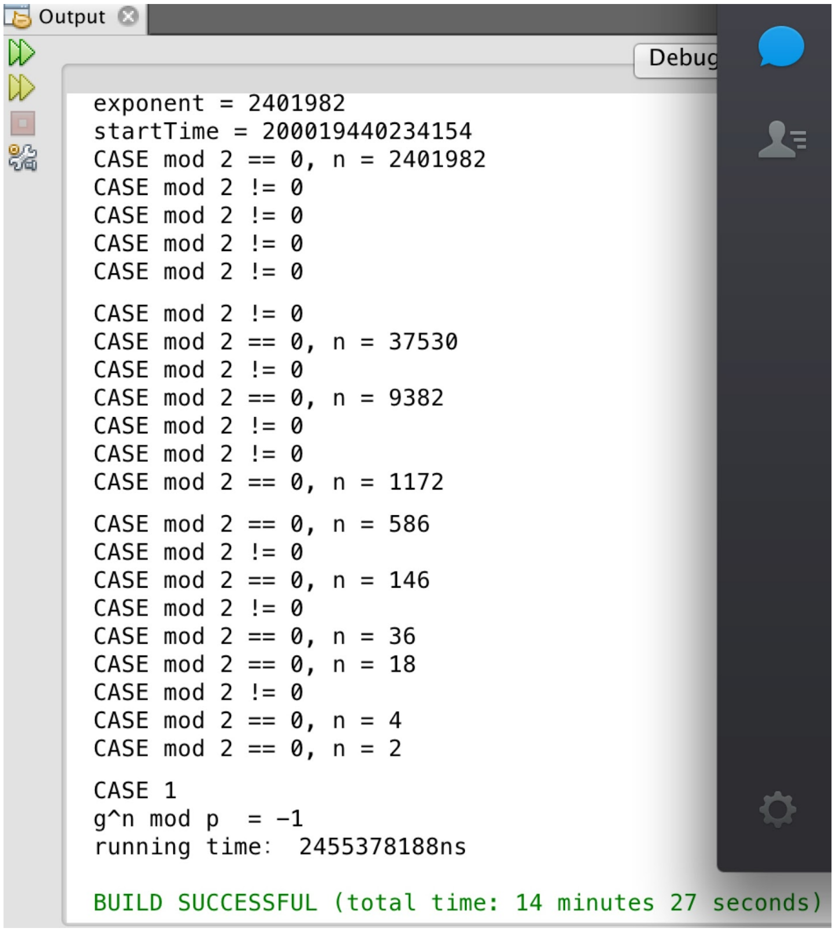


(b)
Code in Java:

```
public class Q8_2 {
    static BigInteger bi2 = new BigInteger("2");
    static BigInteger bi1 = new BigInteger("1");
    static BigInteger bi0 = new BigInteger("0");

public static void main(String[] args) {
        BigInteger prime= new BigInteger("49999");
        BigInteger generator= new BigInteger("123456789999999");
        BigInteger exponent= new BigInteger("2");

        long startTime = 0;
        long endTime = 0;

        while ((endTime - startTime) < 2000000000){
            exponent = exponent.add(new BigInteger("10"));
            System.out.println("exponent = " + exponent + "  ");

            // start time
            startTime=System.nanoTime();
            System.out.println("startTime = " + startTime + "  ");

            // g ^ n mod p
            BigInteger g_n_p = exponent(generator, exponent, prime);
            System.out.println("g^n mod p  = "+ g_n_p);

            // end time
             endTime=System.nanoTime();
             System.out.println("running time：  "+(endTime-startTime)+"ns\n");
        }
    }
    private static BigInteger exponent(BigInteger g, BigInteger n, BigInteger p){
        return exponentHelper(p, new BigInteger("1"), n, p);
    }

    private static BigInteger exponentHelper(BigInteger m, BigInteger x, BigInteger n,
BigInteger p){
        if (n.equals(bi0)) {
            System.out.println("CASE 0");
            return bi0;
        }
```

```
        else if (n.equals(bi1)){
            System.out.println("CASE 1");
            return x.multiply(m).mod(p);
        }
        else {
            if (n.mod(bi2).equals(bi0)){
                System.out.println("CASE mod 2 == 0, n = "+ n );
                exponentHelper(m.multiply(m).mod(p), x, n.divide(bi2), p);
            } else {
                System.out.println("CASE mod 2 != 0, n = " + n);
                exponentHelper(m.multiply(m).mod(p), x.multiply(m).mod(p), n.divide(bi2), p);
            }
        }
        return new BigInteger("-1");
    }
}
```

Running result: largest n to be 2401982 within 2.0s
given g = 123456789999999, prime = 49999


Output
Debug
exponent = 2401982
startTime = 200019440234154
CASE mod 2 == 0, n = 2401982
CASE mod 2 != 0
CASE mod 2 != 0
CASE mod 2 != 0
CASE mod 2 != 0
CASE mod 2 != 0
CASE mod 2 == 0, n = 37530
CASE mod 2 != 0
CASE mod 2 == 0, n = 9382
CASE mod 2 != 0
CASE mod 2 != 0
CASE mod 2 == 0, n = 1172
CASE mod 2 == 0, n = 586
CASE mod 2 != 0
CASE mod 2 == 0, n = 146
CASE mod 2 != 0
CASE mod 2 == 0, n = 36
CASE mod 2 == 0, n = 18
CASE mod 2 != 0
CASE mod 2 == 0, n = 4
CASE mod 2 == 0, n = 2
CASE 1
g^n mod p = -1
running time: 2455378188ns
BUILD SUCCESSFUL (total time: 14 minutes 27 seconds)