Il calcolo di (250.000.000 cifre di) *e*

1 Introduzione

Consideriamo la funzione esponenziale:

$$a^x$$

e la sua derivata:

$$\frac{d}{dt}a^x = \lim_{h\to 0}\frac{a^{x+h}-a^x}{h}$$

vale:

$$\frac{d}{dt}a^x = a^x \lim_{h\to 0}\frac{a^h-1}{h} = C(a)a^x$$

Siamo interessati al valore di a per cui $C(a) = 1$, ovvero a risolvere l'equazione differenziale del primo ordine

$$y' = y.$$

Manipolando opportunamente l'equazione:

$$\lim_{h\to 0}\frac{a^h-1}{h} = 1$$

si ottiene il valore cercato:

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n \qquad (1)$$

espressione che comunemente viene usata come definizione del numero *e.*

Sviluppando la potenza del binomio e andando al limite (con accuratezza) si ottiene lo sviluppo in serie:

$$e = \lim_{n\to\infty} s_n, \quad s_n = 1+1+\frac{1}{2}+\frac{1}{2\,3}+\ldots+\frac{1}{n!} = \sum_{i=0}^{n}\frac{1}{i!} \qquad (2)$$

La relazione (2) converge molto velocemente, si dimostra

$$s_n < e < s_n + \frac{1}{n\,n!}$$

Confrontiamo due algoritmi, uno per il calcolo della(1) e uno per la (2).

ALGORITMO 1: Formula (1) con $n=2^m$

$x_0 = 1+2^{-m}$

$x_i = x_{i-1}\, x_{i-1}, \;\; i=1,\ldots,m$

NB. i calcoli vanno effettuati in precisione fissa

Esempio di convergenza (con Mathematica)

m	valore approssimato
10	2,71695572946643554499315528362682079339408387911632010086272857835178
50	2,71828182845904402820065812119279527097993467566856504982774090089721
100	2,71828182845904523536028747135159032462532507892279563366245802233809
200	2,71828182845904523536028747135266249775724709369995957496696678192860
300	2,71828182845904523536028747135266249775724709369995957496696762772407

ALGORITMO 2: Regola di Ruffini-Horner applicata alla (2)

$x_0 = 1+1/n$

$x_i = (1+ x_{i-1}/(n-i)),\ i=1,\ldots,n-1$

$x_n = 1+ x_{n-1}$

NB. i calcoli possono essere effettuati con precisione crescente

Esempio di convergenza (con Mathematica)

n	valore approssimato
10	2,7182818011463844797180578093087797964154384 7
50	2,7182818284590452353602874713526624977572470936999595749669676277 2341
100	2,7182818284590452353602874713526624977572470936999595749669676277 2407

2 Digressione: Uso della FFT

Il prodotto di due numeri interi di n cifre con l'algoritmo classico richiede n somme di $O(n)$ cifre per un totale di $O(n^2)$ operazioni elementari.

Un algoritmo molto più efficiente $O(n \log^2 n)$ è basato sull'uso della FFT.

Consideriamo dapprima il caso del prodotto di due polinomi (un intero rappresentato in base b è un polinomio in b). Si ha

$$p(x)q(x) = \sum_{i=0}^{n} \sum_{j=0}^{i} a_j b_{i-j} x^i$$

operazione che richiede $O(n^2)$ moltiplicazioni tra i coefficienti.

Un metodo alternativo sarebbe valutare i due polinomi in $2n$ punti

$$\{x_0, x_1, \ldots x_{2n-1}\},$$

calcolare i prodotti dei valori dei due polinomi in quei punti e trovare l'unico polinomio che assume quei valori in quei punti, questo polinomio è il prodotto dei polinomi di partenza.

In genere questo non sembra aiutare molto ma, per mezzo della *Fast Fourier Transform,* si può valutare un polinomio nelle radici dell'unità e tornare indietro con una complessità $O(n \log n)$ e una buona stabilità numerica.

L'algoritmo della moltiplicazione di interi con la FFT si può dividere nei seguenti passi.

1. Si sceglie una base adeguata (100 permette di arrivare a 300 milioni di cifre senza rischi di errori numerici).
2. Si rappresentano i coefficienti dei polinomi con due array di reali di lunghezza $2n$.
3. Si calcolano le trasformate dirette dei due array (costo $O(n \log n)$).
4. Si fa una moltiplicazione elemento per elemento (costo $O(n)$)
5. Si calcola la trasformata inversa del risultato (costo $O(n \log n)$).
6. Si riporta il risultato a numeri interi (l'errore assoluto deve essere minore di 0.5).
7. Si propagano i riporti in modo da tornare alla base (100).

La complessità totale e $O(n \log n)$ in termini di *operazioni aritmetiche a precisione finita* e $O(n \log^2 n)$ in termini di *operazioni su bit*. Se i numeri sono piccoli è preferibile moltiplicarli in modo tradizionale.

Se si temono gli arrotondamenti numerici si può operare in una struttura finita (campi di finiti: interi modulo un numero primo) invece che nel campo dei complessi, ma in tal caso l'algoritmo diviene più lento e macchinoso e i problemi di arrotondamento si trasformano in problemi di overflow.

Esiste un algoritmo più veloce ma molto più complicato dovuto a Schönage e Strassen che richiede O(n log n log log n) *bit-operations.*

NB: tutti questi algoritmi valgono anche per i numero reali, trattando separatamente, in modo corretto, le mantisse e gli esponenti.

3 Digressione: il calcolo del Fattoriale

Il fattoriale di n (ovvero il prodotto dei numeri interi fino a n) si indica con $n!$ ed è una funzione che cresce molto rapidamente. Abbiamo

10! = 3 628 800,
14! = 87 178 291 200

e già i numeri a 32 bit non sono sufficienti per la rappresentazione.
Una buona approssimazione si ottiene troncando al primo termine lo sviluppo asintotico di Stirling:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + \cdots\right)$$

In generale vale $\ln n! \geq n \ln n - n$ e per calcolare e con milioni di termini della serie (2) è indispensabile l'utilizzo della aritmetica in multipla precisione.

Data un implementazione della multi-precisione in cui l'addizione costa O(n) e le moltiplicazione M(n) = Ω(n) il calcolo di $n!$ con l'algoritmo banale richiede

O(n M(n log n)) operazioni su bit,

usando l'algoritmo veloce sopra descritto si ottiene un costo di

$O(n^2 \log^3 n)$) operazioni su bit.

Esiste una tecnica alternativa detta *Binary Splitting* che equilibra meglio il succedersi delle operazioni. Definiamo la quantità:

$$Q(a,b) = \prod_{x=a+1}^{b} x = \frac{b!}{a!}$$

vale $Q(0,n) = n!$ e il calcolo si può organizzare in modo ricorsivo, sia

$$m = \left\lfloor \frac{a+b}{2} \right\rfloor$$

allora

$$Q(a,b) = 1, \quad \text{se} \quad a = b$$
$$Q(a,b) = Q(a,m)\, Q(m,b) \quad \text{se} \quad a > b$$

Considerando che $Q(a,b)$ ha un numero di cifre $O((b\text{-}a) \log b)$ e che il calcolo di $Q(m, b)$ è più costoso di quello di $Q(a, m)$, la complessità $Cq(a,b)$ del calcolo di $Q(a,b)$ si può scrivere:

$$Cq(a,b) \leq O(M((b\text{-}a) \log b)) + 2\, Cq(m, b),$$

da cui, usando il fatto che $M(n) = \Omega(n)$

$$Cq(a,b) = O(\log (b\text{-}a)\, M((b\text{-}a) \log b)),$$

e per il fattoriale, nel caso della moltiplicazione con la FFT, si ottiene

$$Cq(0,n) = O(\log (n)\, M(n \log n)) = O(n \log^3 n),$$

con un guadagno estremamente significativo, dovuto al fatto che in questo modo il calcolo avviene sempre su fattori di dimensioni pressoché simili.

5 *Calcolo di e con il Binary Splitting*

Consideriamo ora la serie (2) troncata al *n*-esimo termine; mettendo a comune denominatore tutti i termini tranne il primo si ottiene:

$$s_n = 1 + 1 + \frac{1}{2} + \frac{1}{2\,3} + \ldots + \frac{1}{n!} = 1 + \frac{P(0,n)}{Q(0,n)},$$

con

$$P(a,b) = \sum_{c=a+1}^{b} Q(c,b)$$

Vale la anche relazione ricorrente

$$P(a,b) = P(a,m)\, Q(m,b) + P(m,b)$$

che permette di calcolare contemporaneamente $P(0,m)$ e $Q(0,m)$ con un'unica ricorsione (conservando opportunamente i risultati intermedi) con un costo totale

$$Cpq(\mathrm{n}) = \mathrm{O}(\log(n)\, \mathrm{M}(n \log n)) = \mathrm{O}(n \log^3 n),$$

Per ottenere l'approssimazione di *e* si deve calcolare $P(0,n)$ $Q(0,n)$ che sono interi e quindi sommare 1 al loro rapporto, calcolato a sua volta con un numero adeguato di cifre di precisione.

4 Digressione: il calcolo del rapporto e del reciproco

Poiché il rapporto $P(0,n)/Q(0,n)$ vale circa 1.718… i due numeri interi hanno lo stesso numero di cifre e, a tutti gli effetti, possono essere considerati due numeri reali compresi tra 0 e 1 anteponendo un punto decimale prima della cifra più significativa. Per esempio per n = 20 si ha:

$$\frac{P(0,20)}{Q(0,20)} = \frac{41804113110714400 01}{24329020081766400 00} = \frac{0.41804113110714400 01}{0.24329020081766400 00} = 1.7182818284590452\ldots$$

Il metodo d'elezione per il calcolo del reciproco e del rapporto è il metodo di Newton che, nelle sue varianti, garantisce una convergenza quadratica. Si ha

$$\frac{1}{x} = \frac{1}{1-\delta} = \sum_{i=0}^{\infty} \delta^i = (1+\delta)\left(1+\delta^2\right)\left(1+\delta^4\right)\ldots \qquad (3)$$

Nel nostro caso per ottimizzare i tempi di esecuzione è necessario dividere il calcolo in 4 fasi.

1. Si approssima $1/Q(0,n)$ con qualche centinaio di cifre applicando la formula (3) con precisione fissa (per esempio 200 cifre).
2. Si aumenta la precisione applicando un passo dell'algoritmo di Newton e raddoppiando ogni volta la precisione fino a raggiungere il numero di cifre desiderato.
3. Si applica un altro passo a precisione piena per depurare il risultato dagli errori di arrotondamento.
4. Si moltiplica $1/Q(0,n)$ per $P(0,n)$ a precisione piena.

6 *Esperimenti con Mathematica*

Mathematica con la sua implementazione standard della aritmetica in multipla precisione (e anche simbolica e anche razionale esatta) è lo strumento più adatto a sperimentare algoritmi di questo tipo, anche se molto del divertimento va perso. Il calcolo diretto di 300 milioni di cifre di e con l'algoritmo interno richiede meno di 20 minuti (438 secondi).

Calcolo diretto

```
Timing[e = N[E, 300 001 000];]
```

```
{438.869, Null}
```

L'algoritmo (1) arriva a 10000 cifre in 9 secondi.

Algoritmo 1

```
p[n_, k_] := (
  x = 1 + N[1 / 2^n, k];
  Do[x = x * x, {n}];
  N[x - N[E, k]])

Timing[p[10 000, 4000]]
```

$$\{0.496216,\ -6.812520880815603 \times 10^{-3011}\}$$

```
Timing[p[33 300, 11 000]]
```

$$\{9.29568,\ -6.829813987391071 \times 10^{-10\,025}\}$$

Mentre con l'algoritmo (2) per 35000 cifre basta mezzo secondo e con 27500 secondi (meno di 8 ore) si arriva quasi a 10 milioni.

Algoritmo 2

```
s[n_, k_] := (
  x = 1 + N[1 / n];
  Do[x = SetPrecision[(1 + x / j), 5 (n - j)], {j, n - 1, 1, -1}];
  N[SetPrecision[x, k] - N[E, k]])
```

```
Timing[s[2100, 8000]]
```

$$\{0.018509,\ -9.45055336902268 \times 10^{-6071}\}$$

```
Timing[s[10 000, 49 000]]
```

$$\{0.454405,\ -3.513146535769827 \times 10^{-35\,664}\}$$

```
Timing[s[25 300, 111 000]]
```

$$\{3.3751,\ -4.984362726962477 \times 10^{-100\,419}\}$$

```
Timing[s[200 000, 1 100 000]]
```

$$\{174.569,\ 1.151371870483530 \times 10^{-956\,588}\}$$

```
Timing[s[210 000, 1 200 000]]
```

$$\{191.511,\ 1.170342964408962 \times 10^{-1\,006\,588}\}$$

```
Timing[s[2 100 000, 11 000 000]]
```

$$\{27\,562.2,\ -1.670434652368520 \times 10^{-9\,956\,588}\}$$

La superiorità del *Binary Splitting* è evidente, un minuto invece di 8 per per i 10 milioni.

BinarySplitting

```
Clear[P, Q, p];
Q[a_, a_] := 1;
Q[a_, b_] := Product[x, {x, a + 1, b}] /; b < a + 80;
Q[a_, b_] := Q[a, b] = Module[{m}, m = Floor[(a + b) / 2]; Q[a, m] Q[m, b]]
P[a_, b_] := p[a, b] /; b <= a + 20;
P[a_, b_] := Module[{m}, m = Floor[(a + b) / 2]; P[a, m] Q[m, b] + P[m, b]];

p[a_, a_] = 0;
p[a_, b_] := 1 /; b == a + 1;
p[a_, b_] := 1 + Sum[Product[x, {x, z, b}], {z, a + 2, b}]

b[n_] := N[N[1 + P[0, n] / Q[0, n], k] - N[E, k]]

k = 1 100 000;
Timing[b[210 000]]
```

$$\{3.65524,\ -2.552034324035886 \times 10^{-1\,026\,473}\}$$

```
k = 10 500 000;
Timing[b[1 750 000]]
```

$$\{64.9614,\ -9.87376183830800 \times 10^{-10\,165\,312}\}$$

7 *Implementazione Java*

Per una implementazione estesa degli algoritmi in un linguaggio standard è stato deciso di usare Java per le sue doti di pulizia e di facilità di implementazione, indispensabili per una elegante implementazione della aritmetica multi precisione.

```
package big;
import FFT.*;

public class BigInt {
 protected byte[] A; // BIG endian;
 public final static int MAXDIGIT = 100;

 protected BigInt(byte[] A) {this.A = A;}

 public BigInt(long x) {
      if (x == 0) this.A = new byte[1];
      else {
            int n = 1;
            long z = x;
            while ((z = z / MAXDIGIT) > 0) n++;
            this.A = new byte[n];
            for (int i = 0; i < n; i++) {
                  this.A[i] = (byte) (x % MAXDIGIT);
                  x /= MAXDIGIT;
            }
      }
 }
```

```
 public void normalize() {
       int i = A.length - 1;
       while (i >= 0 && A[i] == 0) i--;
       if (i == -1) this.A = new byte[1];
       else if (i == A.length - 1) return;
       byte[] res = new byte[i + 1];
       System.arraycopy(A, 0, res, 0, i + 1);
       A = res;
 }

 final static int LIM = 40;



 public static BigInt multiply(BigInt val1, BigInt val2) {
       if(val1.A.length<LIM || val2.A.length<LIM) return simpleMultiply(val1, val2);
       int n = makePowerOfTwo(Math.max(val1.A.length, val2.A.length)) * 2;
       double[] AF = padWithZeros(val1.A, n);
       double[] BF = padWithZeros(val2.A, n);
       FFT.realFT(AF, FFT.DIRECT);
       FFT.realFT(BF, FFT.DIRECT);
       double c = Math.sqrt(n);
       AF[0] *= (c * BF[0]);
       AF[1] *= (c * BF[1]);
       for (int i = 2; i < n; i += 2) {
             double ai = c * AF[i], ai1 = c * AF[i + 1];
             AF[i] = ai * BF[i] - ai1 * BF[i + 1];
             AF[i + 1] = ai1 * BF[i] + ai * BF[i + 1];
       }
       FFT.realFT(AF, FFT.INVERSE);
       BigInt res = new BigInt(propagateCarries(AF));
       res.normalize();
       return res;
}
```

```java
private static BigInt simpleMultiply(BigInt val1, BigInt val2) {
	if (val1.A.length < val2.A.length) return simpleMultiply(val2, val1);
	int[] acc = new int[val1.A.length + val2.A.length];
	for (int k = 0; k < val2.A.length; k++)
		for (int i = 0; i < val1.A.length; i++)
			acc[i + k] += val1.A[i] * val2.A[k];
	BigInt res = new BigInt(propagateCarries(acc));
	return res;
}

 public static BigInt add(BigInt val1, BigInt val2) {
	if(val1.A.length<val2.A.length)return add(val2,val1);
	int[] res = new int[val1.A.length+1];
	for (int i = 0; i < val2.A.length; i++) res[i]=val1.A[i]+val2.A[i];
	for (int i = val2.A.length; i < val1.A.length; i++) res[i]=val1.A[i];
	return new BigInt(propagateCarries(res));
 }


private static byte[] propagateCarries(int[] A) {
	byte[] res = new byte[A.length];
	long carry = 0;
	for (int i = 0; i < A.length; i++) {
	long x = ((long) A[i]) + carry;
		carry = x / MAXDIGIT;
		res[i] = (byte) (x % MAXDIGIT);
	}
	if (carry == 0) return res;

	byte[] res1 = new byte[A.length + 1];
	System.arraycopy(res, 0, res1, 0,res.length);
	res1[A.length] = (byte) carry;
	return res1;
 }
```

```
 private static byte[] propagateCarries(double[] A) {
      byte[] res = new byte[A.length];
      long carry = 0;
      for (int i = 0; i < A.length; i++) {
            long x = ((long) Math.round(A[i])) + carry;
            carry = x / MAXDIGIT;
            res[i] = (byte) (x % MAXDIGIT);
      }
      if (carry == 0) return res;

      byte[] res1 = new byte[A.length + 1];
      System.arraycopy(res, 0, res1, 0,res.length);
      res1[A.length] = (byte) carry;
      return res1;
 }

 private static double[] padWithZeros(byte[] A, int n) {
      double[] res = new double[n];
      for (int i = 0; i < A.length; i++)res[i] = A[i];
      return res;
 }

 private static int makePowerOfTwo(int max) {
      int twoPower = 1;
      for (; twoPower < max; twoPower *= 2);
      return twoPower;
 }
}


package big;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import FFT.*;
```

```java
public class BigReal {
 public final static int MAXDIGIT = 100;
 private byte[] A; // BIG endian;
 private boolean isZero;
 private int n = 0;

 private BigReal(BigReal X) {
      this.A=X.A.clone();
      this.n=X.n;
      this.isZero=X.isZero();
 }

 private BigReal(byte[] A) {
      this.A=A;
      this.n=A.length;
      this.isZero=false;
 }

 public BigReal(BigInt x, int l) {
      int xl = x.A.length;
      this.n = makePowerOfTwo(l/2 + 1);
      l = n - 1;
      if (l > xl) l = xl;
      this.A = new byte[n];
      if (x.A[xl - 1] > 9) System.arraycopy(x.A, xl - l, A, n - l - 1, l);
      else {
            for (int i = 0; i < l - 1; i++) A[n-i-2] =
                  (byte) ((x.A[xl-i-1]%10)*10 + (x.A[xl-i- 2]/10));
            A[n-l-1] = (byte) ((x.A[0] % 10) * 10);
      }
      this.isZero = false;
 }
```

```
public void addOne() {
      A[A.length - 1] = (byte) (A[A.length - 1] + 1);
}

public BigReal complement() {
      for (int i = 0; i < A.length - 1; i++) A[i] = (byte) (99 - A[i]);
      return this;
}

public void square() {
      double[] AF = padWithZeros(A, 2 * n);
      FFT.realFT(AF, FFT.DIRECT);
      double c = Math.sqrt(2 * n);
      AF[0] *= (c * AF[0]);
      AF[1] *= (c * AF[1]);
      for (int i = 2; i < 2 * n; i += 2) {
            double ai = AF[i], ai1 = AF[i + 1];
            AF[i] = c * (ai * ai - ai1 * ai1);
            AF[i + 1] = 2 * c * ai * ai1;
      }
      FFT.realFT(AF, FFT.INVERSE);
      isZero = true;
      long carry = 0;
      for (int i = 0; i < AF.length - 1; i++) {
            long x = ((long) Math.round(AF[i])) + carry;
            carry = x / MAXDIGIT;
            if (i > n - 2) {
                  A[i - n + 1] = (byte) (x % MAXDIGIT);
                  if (A[i - n + 1] != 0) isZero = false;
            }
      }
}

public boolean isZero() {return isZero;}
```

```
public  void multiply(BigReal delta) {  // this = this * delta+
	int m=2*n;
	double[] AF = padWithZeros(A, m);
	double[] BF = padWithZeros(delta.A, m);
	FFT.realFT(AF, FFT.DIRECT);
	FFT.realFT(BF, FFT.DIRECT);
	double c = Math.sqrt(m);
	AF[0] *= (c * BF[0]);
	AF[1] *= (c * BF[1]);
	for (int i = 2; i < m; i += 2) {
		double ai = c * AF[i], ai1 = c * AF[i + 1];
		AF[i] = ai * BF[i] - ai1 * BF[i + 1];
		AF[i + 1] = ai1 * BF[i] + ai * BF[i + 1];
	}
	FFT.realFT(AF, FFT.INVERSE);
	long carry = 0;
	for (int i = 0; i < AF.length-1; i++) {
		long x = ((long) Math.round(AF[i])) + carry;
		carry = x / MAXDIGIT;
		if(i>n-2)A[i-n+1]= (byte) (x % MAXDIGIT);
	}
 }


public static BigReal multiply1(BigReal X, BigReal Y) {  // X*(Y+1)
	int m = makePowerOfTwo(Math.max(X.A.length, Y.A.length)) * 2;
	double[] AF = padWithZeros(X.A, m);
	double[] BF = padWithZeros(Y.A, m);
	BF[Y.A.length-1] ++;

	FFT.realFT(AF, FFT.DIRECT);
	FFT.realFT(BF, FFT.DIRECT);
	double c = Math.sqrt(m);
	AF[0] *= (c * BF[0]);
	AF[1] *= (c * BF[1]);
```

```
        for (int i = 2; i < m; i += 2) {
                double ai = c * AF[i], ai1 = c * AF[i + 1];
                AF[i] = ai * BF[i] - ai1 * BF[i + 1];
                AF[i + 1] = ai1 * BF[i] + ai * BF[i + 1];
        }
        FFT.realFT(AF, FFT.INVERSE);
        int n=m/2;
        byte[] A = new byte[n];
        long carry = 0;
        for (int i = 0; i < AF.length-1; i++) {
                long x = ((long) Math.round(AF[i])) + carry;
                carry = x / MAXDIGIT;
                if(i>n-2)A[i-n+1]= (byte) (x % MAXDIGIT);
        }
        return new BigReal(A);
}

public BigReal inverse() {
        complement();
        BigReal X = new BigReal(this);
        X.addOne();
         square();
        int i=0;
        while (!isZero() && i++ < 100) {
                X = BigReal.multiply1(X,this);
               square();
        }
        this.A=X.A;
        isZero=false;
        return this;
}
```

```
public static BigReal inverse(BigInt Q) {
      int l = Q.digits();
      while(l>1000)l=l/2+1;
      BigReal QF = null, X = new BigReal(Q, l).inverse();
      while (l < Q.digits()) {
            l *= 2;
            QF = new BigReal(Q, l);
            QF.multiply(X);
            int k=0;
            while (QF.A[QF.A.length - 1] > 0) {
                  X.A[k++] = 0;
                  QF = new BigReal(Q, l);
                  QF.multiply(X);
            }
            QF.complement();
            X = BigReal.multiply1(X, QF);
      }
      QF.square();
      X = BigReal.multiply1(X, QF);
      return X;
 }
 private static double[] padWithZeros(byte[] A, int n) {
      double[] res = new double[n];
      int m=n/2-1, s=A.length-1;;
      int nc = Math.min(A.length,n/2);
      for (int i = 0; i < nc; i++)res[m-i] = A[s-i];
      return res;
  }

 private static  int makePowerOfTwo(int max) {
      int twoPower = 1;
      for (; twoPower < max; twoPower *= 2);
      return twoPower;
 }
}
```

```java
package big;

import java.util.HashMap;

public class ComputePQ {
 private HashMap<Long, BigInt> qMap;

 public ComputePQ() {qMap = new HashMap<Long, BigInt>(100000);}

 public BigInt q(int a, int b) {
      long x = a + 1, y = a + 2;
      while (y <= b && x < Long.MAX_VALUE / y) {
            x *= y;
            y++;
      }
      if (y == b + 1) return new BigInt(x);
      BigInt qq = qMap.get(a * 1000000000L + b);
      if (qq != null) return qq;
      qq = BigInt.multiply(q(a, (a + b) / 2), q((a + b) / 2, b));
      qMap.put(a * 1000000000L + b, qq);
      return qq;
 }

 public BigInt p(int a, int b) {
      if (b == a + 1) return new BigInt(1);
      return BigInt.add(
                  p((a+b)/2,b),
                  BigInt.multiply(p(a,(a+b)/2),q((a+b)/2,b))
            );
      }

 public void finalize() {
      Util.askMemory();
      qMap = null;
 }
```

```
public static BigInt P = null, Q = null;

 public static void go(int n) {
       ComputePQ CPQ = new ComputePQ();
       BigInt P0 = CPQ.p(0, n / 2);
       BigInt Q0 = CPQ.q(0, n / 2);
       CPQ.finalize();
       CPQ = new ComputePQ();
       BigInt P1 = CPQ.p(n / 2, n);
       BigInt Q1 = CPQ.q(n / 2, n);
       Q = BigInt.multiply(Q0, Q1);
       P = BigInt.add(P1, BigInt.multiply(P0, Q1));
       CPQ.finalize();
 }


package big;
public class ComputeE {

 final static String dir = " . . . ";

 /*
  * Compute the number of term SeriesSize of the series required. SeriesSize
  * is such that log(seriesSize !) > n*log(10.)
  */
 static int seriesSize(int nDigits) {
       int s = 1;
       double logFactorial = 0., logMax = (double) nDigits * Math.log(10.);
       while (logFactorial < logMax) {
             logFactorial += Math.log((double) s);
             s++;
       }
       return s;
}
```

```
static void computeE(int nDigits) {
      int n = 5 + seriesSize(nDigits);
      int lp = 0;
      Util.initTime();
      ComputePQ.go(n);
      BigInt P = ComputePQ.P;
      BigInt Q = ComputePQ.Q;
      lp = P.digits();
      BigReal X = BigReal.inverse(Q);
      BigReal E = new BigReal(P,lp);
      E.multiply(X);
      E.addOne();
}

      public static void main(String[] args) {
            computeE(300000000);
      }
}
```

```
Required Digits = 1000000, Series size = 205029
 P,Q          time = 6.625sec., memory 2.0M, max memory 11M
 inverse      time = 11.28sec., memory 1.0M, max memory 11M
 end          time = 12.27sec., memory 2.0M, max memory 11M
 final error  10^-1000040

Required Digits = 10000000, Series size = 1723514
 P,Q          time = 128.51sec., memory 14.0M, max memory 106M
 inverse      time = 266.07sec., memory 17.0M, max memory 153M
 end          time = 290.94sec., memory 25.0M, max memory 153M
 final error  10^-10000044

Required Digits = 100000000, Series size = 14842913
 P,Q          time = 1800.835sec., memory  95.0M, max memory 1312M
 inverse      time = 3188.764sec., memory 159.0M, max memory 1312M
 end          time = 3444.445sec., memory 223.0M, max memory 1312M
 final error  10^-100000050

Required Digits = 250000000, Series size = 35153475
 P,Q          time = 4097.136sec., memory 448.0M, max memory 3190M
 inverse      time = 7068.457sec., memory 366.0M, max memory 3190M
 end          time = 7630.16sec.,  memory 494.0M, max memory 3190M
 final error  10^-250000056

Required Digits = 300000000, Series size = 41745989
 P,Q          time = 8756.172sec., memory 423.0M, max memory 3866M

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
       at big.BigReal.padWithZeros(BigReal.java:210)
       at big.BigReal.multiply(BigReal.java:81)
       at big.BigReal.inverse(BigReal.java:194)
       at big.ComputeE.computeE(ComputeE.java:33)
       at big.ComputeE.main(ComputeE.java:49)
```