Algoritmo BBP per π

Esperienze di Programmazione

Antonio Pitasi

19/01/2018

1 Introduzione al problema

In questo progetto, si cerca di dare una buona implementazione della formula di Bailey-Borwein-Plouffe, che modificata in modo appropriato, consente di calcolare l'n-esima cifra esadecimale di π senza conoscerne le precedenti. La formula originale, denominata *BBP formula*, è la seguente:

$$\pi = \sum_{k=0}^{\infty} \left[\frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \right]$$

Pur rappresentando un buon metodo per il calcolo di π, non è il più veloce conosciuto oggi (record attualmente detenuto dall'algoritmo dei fratelli Chudnovsky). Tuttavia, come già accennato, sarà possibile ricavarne un modo per calcolare una cifra arbitraria di π. Un vantaggio di questo approccio, consiste nella possibilità di avere macchine che calcolano cifre diverse parallelamente, poiché il risultato di un calcolo non influenza gli altri.

La formula scoperta da Plouffe è stata ispirata da una formula già nota, anch'essa in forma di sommatoria:

$$\log(2) = \sum_{k=1}^{\infty} \frac{1}{2^k \cdot k}$$

All'interno del programma viene implementata pure quest'ultima, utilizzabile per calcolare l'n-esima cifra binaria di $\log(2)$, ancora una volta senza conoscerne le precedenti.

2 Algoritmi

2.1 Il calcolo di log(2)

Borwein e Plouffe osservarono come la formula scritta in precedenza possa essere usata per calcolare le cifre binarie di $\log(2)$ a partire da una posizione arbitraria.

Il ragionamento percorso è stato il seguente:

$$\log(2) = \sum_{k=1}^{\infty} \frac{1}{2^k \cdot k}$$

L'obiettivo dunque è calcolare le cifre *binarie* che partono alla posizione $d+1$. Si può notare come questo sia equivalente a calcolare $\{2^d \cdot \log(2)\}$, dove $\{\}$ indica la parte frazionaria[1].

Si effettuano alcune trasformazioni:

$$\begin{aligned}
\left\{2^d \cdot \log(2)\right\} &= \left\{\sum_{k=1}^{\infty} \frac{2^{d-k}}{k}\right\} \\
&= \left\{\left\{\sum_{k=1}^{d} \frac{2^{d-k}}{k}\right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k}\right\} \\
&= \left\{\left\{\sum_{k=1}^{d} \frac{2^{d-k} \bmod k}{k}\right\} + \sum_{k=d+1}^{\infty} \frac{2^{d-k}}{k}\right\}
\end{aligned}$$

Nel primo passaggio la sommatoria è stata divisa in due sommatorie per separare la prima che possiede una parte intera, dalla seconda che invece è sicuramente minore di 1.[2]

Nel secondo passaggio è stato aggiunto *mod k*, giustificato dal fatto che interessa solo la parte frazionaria del quoziente della divisione per k.

Adesso si può notare come i termini della seconda sommatoria diventino piccoli molto velocemente, quindi in base alla precisione di macchina disponibile bastano poche iterazioni per un'approssimazione sufficiente a trovare la cifra in posizione d+1, ma anche qualcuna delle successive.

Per l'implementazione si rimanda alla *Sezione 3*.

2.2 Il calcolo di π

Dopo essere riusciti con log(2), Borwein e Plouffe sono andati alla ricerca di formule analoghe che funzionassero con altre costanti. Ed è così che hanno trovato la formula per il calcolo di π:

$$\begin{aligned}
\pi &= \sum_{k=0}^{\infty} \left[\frac{1}{16^k}\left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{5}{8k+5} - \frac{1}{8k+6}\right)\right] \\
&= 4 \cdot S_1 - 2 \cdot S_4 - S_5 - S_6, \\
S_j &= \sum_{k=0}^{\infty} \frac{1}{16^k \cdot (8k+j)}
\end{aligned}$$

[1] La definizione di parte frazionaria è: $\{x\} = x - \lfloor x \rfloor$.
[2] Infatti $2^{d-k} < 1$ per $k > d$.

Dove la sommatoria viene divisa in quattro sommatorie più piccole permettendo una notazione più compatta.

Da qui si può fare un ragionamento molto simile al precedente: per trovare le cifre *esadecimali* che partono alla posizione $d+1$, occorre calcolare $\{16^d \cdot \pi\}$, ricordando che $\{\}$ indica la parte frazionaria. Ovvero:

$$\{16^d \pi\} = \{4 \cdot \{16^d S_1\} - \{2 \cdot \{16^d S_4\}\} - \{16^d S_5\} - \{16^d S_6\}\}$$

Applichiamo le stesse trasformazioni fatte nel paragrafo precedente alle quattro sommatorie:

$$\begin{aligned}
\{16^d S_j\} &= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k}}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\} \\
&= \left\{ \left\{ \sum_{k=0}^{d} \frac{16^{d-k} \bmod 8k+j}{8k+j} \right\} + \sum_{k=d+1}^{\infty} \frac{16^{d-k}}{8k+j} \right\}
\end{aligned}$$

Si nota infatti la somiglianza con il risultato trovato per $\log(2)$. La principale differenza sta nel dover calcolare quattro diversi risultati, per poi combinarli opportunamente insieme.

Il calcolo della prima sommatoria potrebbe sembrare costoso visto l'esponente molto grande al numeratore, ma è reso possibile dall'operazione di *modulo*. Infatti viene utilizzato un algoritmo ottimizzato, spiegato nella *Sezione 3.1*.

3 Implementazione degli algoritmi

3.1 Elevamento a potenza in modulo

Operazioni come $b^e \bmod k$ (dove $b \in \mathbb{R}$, $e \in \mathbb{Z}$, $k \in \mathbb{N}$) vengono calcolate utilizzando un algoritmo noto con il nome di *Right-to-Left binary exponentiation* [2, par. 4.6.3].

Non è possibile calcolare prima b^e come verrebbe spontaneo, perché nel nostro caso sarebbe un numero troppo grosso da rappresentare. Per superare questo problema basta notare come:

$$c = (a \cdot b) \bmod m$$

sia uguale a

$$c = (a \bmod m \cdot b \bmod m) \bmod m$$

Per cui ad ogni iterazione salveremo solo il modulo del risultato intermedio, e non tutto il prodotto.

Adesso per ottimizzare il numero di operazioni necessarie possiamo scrivere e in notazione binaria:

$$e = \sum_{i=0}^{n-1} a_i 2^i$$

Dove a_i può essere 0 o 1, $a_{n-i} = 1$, e n è il numero minimo di bit necessario per rappresentare e.

Quindi:

$$b^e = \prod_{i=0}^{n-1} (b^{2^i})^{a_i}$$

Per maggiore chiarezza, lo pseudocodice risultante è:

```
function ModPow(b, e, k)
    if k = 1 then return 0
    result := 1
    b := b mod k
    while e > 0
        if (e mod 2 == 1):
            result := (result * b) mod k
        e := e / 2
        b := (b * b) mod k
    return result
```

3.2 Altre scelte implementative

La quantità di cifre calcolate a partire dalla posizione specificata, dipende dalla precisione di macchina disponibile. In particolare Bailey stesso durante dei test ha ottenuto almeno nove cifre corrette usando aritmetica a 64 bit, mentre usando 128 bit se ne hanno almeno ventiquattro. [1, p. 4-5]

Nel programma presentato vengono utilizzati numeri in virgola mobile da 128 bit, implementati mediante una libreria standard.

4 Test e verifica della correttezza

Il programma è stato sviluppato seguendo il modello del *Test Driven Development*, questo significa che sono stati realizzati i test per ogni singola funzione prima della funzione stessa.

Ogni file sorgente è accompagnato da un secondo file contenente i relativi test.

La maggior parte della suite di test utilizza la tecnica dell'oracolo: viene confrontato l'output della funzione implementata con risultati noti a priori.

Nel calcolo dell'elevamento a potenza in modulo vengono usati:

$$2^{100} \bmod 2 = 0$$

$$50^{19800} \bmod 6 = 4$$

$$367^{447} \bmod 739 = 687$$

I risultati corretti sono stati trovati utilizzando il calcolatore online *Wolfram Alpha*[3].

Per quanto riguarda invece il calcolo vero e proprio delle cifre di $\log(2)$ e π, oltre all'oracolo, sono stati effettuati anche test sull'*overlapping* delle cifre: ad esempio vengono trovate le prime otto cifre di π: 243F<u>6A88</u>, e le cifre a partire dalla quinta: <u>6A88</u>85A3. Si verifica poi che ci sia corrispondenza tra le quattro finali del primo calcolo e le quattro iniziali del secondo calcolo.

Più in generale, questo metodo è utilizzabile anche per capire quante cifre successive alla posizione desiderata sono corrette.

Oltre a quelle già descritte, viene eseguita anche una semplice funzione per verificare che si stiano usando effettivamente numeri in virgola mobile da 128 bit. Il test consiste nel calcolare $\frac{1}{49} \cdot 49$ e verificare che il risultato sia 1.

Su sistemi a 64 bit fallirebbe poiché $\frac{1}{49}$ non è rappresentabile senza errore.

5 Risultati

Vengono riportati alcuni risultati di esempio trovati eseguendo il programma:

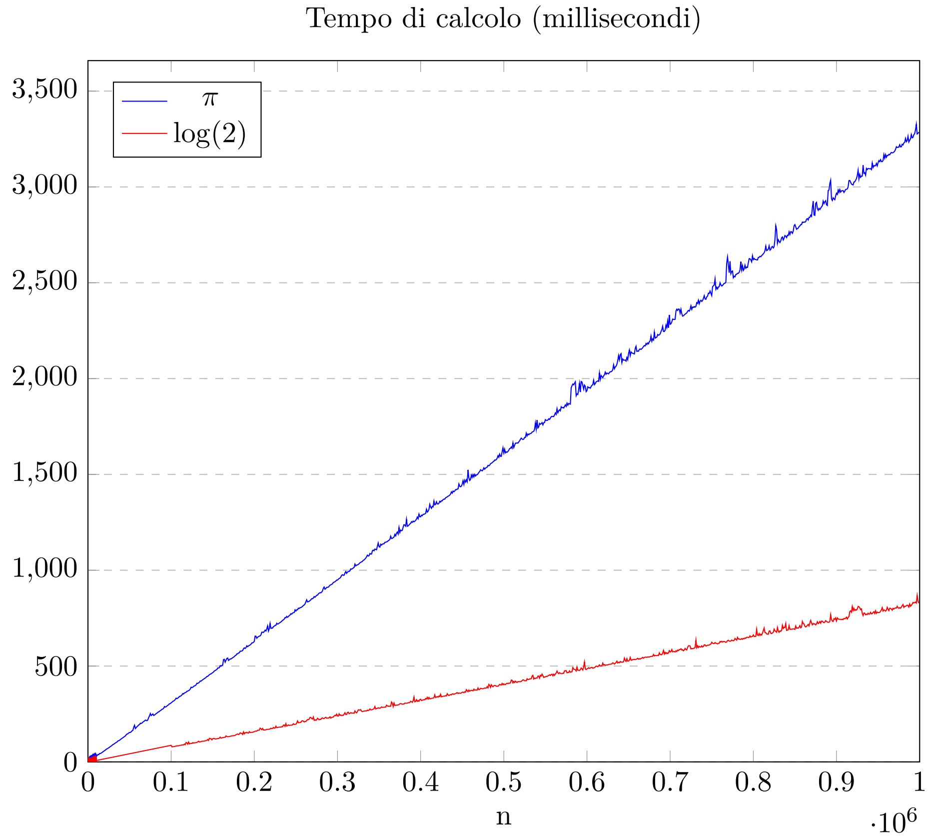
Binary digits of log2 starting at position 1: **10110001**
Binary digits of log2 starting at position 100: **00011111**
Binary digits of log2 starting at position 10000: **00101101**
Binary digits of log2 starting at position 1000000: **11010100**
Binary digits of log2 starting at position 100000000: **01100111**

Hexadecimal digits of pi starting at position 1: **243F6A88**
Hexadecimal digits of pi starting at position 100: **C29B7C97**
Hexadecimal digits of pi starting at position 10000: **68AC8FCF**
Hexadecimal digits of pi starting at position 1000000: **26C65E52**
Hexadecimal digits of pi starting at position 10000000: **17AF5863**

[3]Link: http://www.wolframalpha.com/

5.1 Tempo di computazione

Durante l'esecuzione del programma sul mio computer, ho ottenuto i seguenti tempi di calcolo in funzione della posizione della cifra richiesta n:



Si può notare come gli algoritmi siano lineari rispetto all'input n.

Il peso viene dato dalle sommatorie presenti sia nel calcolo di $log(2)$ che per π, in entrambi i casi hanno $O(n)$ termini.

Come intuibile, si vede anche come il calcolo per $log(2)$ sia meno costoso rispetto a quello per π.

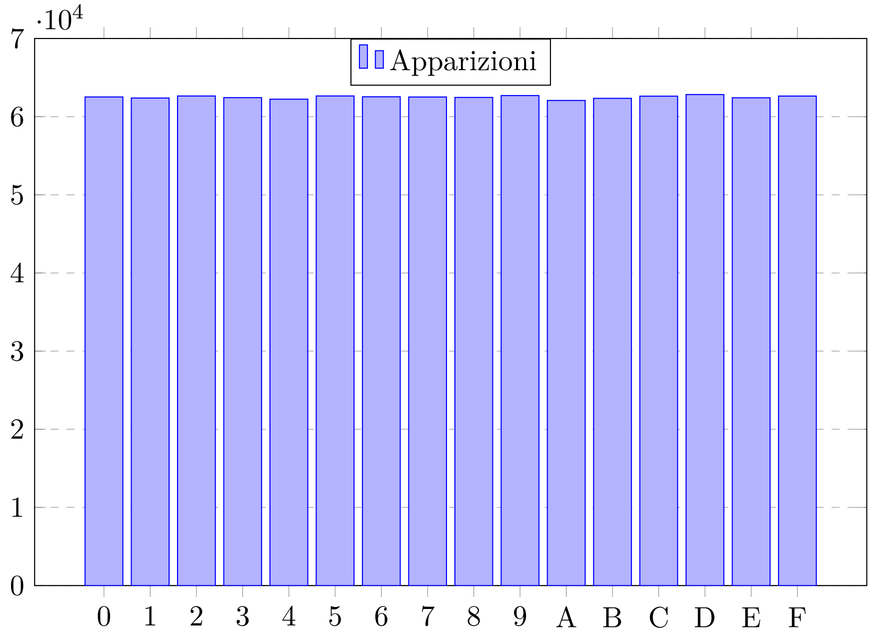
5.2 Possibili (futuri) utilizzi

Ammettendo che le cifre in π compaiano equiprobabilmente[4], si potrebbe dedurre che prima o poi sia possibile incontrare qualunque sequenza di cifre.

Per questo motivo si può pensare di codificare un file, che non è altro che una sequenza di cifre, *all'interno* di π. Sarebbe sufficiente memorizzare solo la posizione della cifra iniziale e la sua lunghezza. Recuperarlo in seguito usando il calcolo dell'n-sima cifra sarebbe *relativamente facile*.

[4] Non è stato dimostrato per il momento.

Sebbene per il momento il calcolo richiederebbe troppo tempo, rendendo questa tecnica inutilizzabile, il grafico che segue mostra i dati ottenuti sulla frequenza di comparsa delle singole cifre all'interno del primo milione di cifre:



Risulta evidente che ogni cifra appare più o meno la stessa quantità di volte delle altre. Questo potrebbe in futuro aprire nuove strade sull'utilizzo di costanti come π per la codifica e compressione di file, o altro ancora.

6 Bibliografia

[1] H. D. Bailey, The BBP Algorithm for Pi, 2006;

[2] D. Knuth, The Art of Computer Programming Vol. 2, 2009;

[3] Bailey-Borwein-Plouffe formula, Wikipedia, 2017;

7 Codice sorgente

7.1 main.go

```go
package main

import (
  "fmt"
  "os"
  "strconv"
  "time"
  b "./bbpformula"
)

func main() {
  filename, args := os.Args[0], os.Args[1:]

  invalidArguments :=
    len(args) != 2 ||
    (args[0] != "log2" && args[0] != "pi")

  if invalidArguments {
    help(filename)
    return
  }

  // Parse second argument as integer
  d, err := strconv.Atoi(args[1])
  if err != nil || d <= 0 {
    fmt.Println("ERROR Invalid number n provided.")
    fmt.Println("Try running:", filename, "pi 10000")
    return
  }

  // Start computing
  var res string
  fmt.Printf("Computing...")
  timeStart := time.Now()

  // Case switch for log2 or pi
  if args[0] == "log2" {
    res, err = b.ToStringBase(b.Log2(d), 2, 8)
    fmt.Printf("\rBinary")
  } else if args[0] == "pi" {
    res, err = b.ToStringBase(b.Pi(d), 16, 8)
    fmt.Printf("\rHexadecimal")
  }

  // Print results
  elapsed := time.Since(timeStart)
  fmt.Printf(" digits of %s starting at position %s: ",
    args[0], args[1])
  if err != nil {
    fmt.Println("Unexpected error:", err)
  }
  fmt.Println(res)
  fmt.Println("Computation took", elapsed)
}

func help(filename string) {
  fmt.Println("This program computes n-th digit of binary log2 or hexadecimal pi.")
```

```
  fmt.Println("Usage:", filename, "[log2/pi] <n>")
}
```

7.2 log2.go

```go
package bbpformula

import (
  "math"
  "math/big"
)

// Log2 computes log2 at binary digit specified, returns a big.Float
func Log2(digit int) (*big.Float) {
  digit--
  k := 1
  res := big.NewFloat(0).SetPrec(128)

  // First summation (from 1 to d)
  for k <= digit {
    numerator, err := ModPow(2, digit-k, k)
    if err != nil {
      // TODO: handle err
    }
    tmpRes := new(big.Float).SetPrec(128).
      Quo(Int64ToFloat(numerator), IntToFloat(k)) // quotient = modpow/k
    res.Add(res, tmpRes)
    k++
  }
  res = FractionalPart(res)

  // Seconds summation (from d+1 to infinite/precision)
  floatPrecision, _ := new(big.Float).SetString(PRECISION)
  delta := IntToFloat(1)
  for delta.Cmp(floatPrecision) >= 0 {
    denominator := IntToFloat(k)
    numerator := big.NewFloat(math.Pow(2, float64(digit-k)))
    delta.Quo(numerator, denominator)
    res.Add(res, delta)
    k++
  }

  return res
}
```

7.3 log2_test.go

```go
package bbpformula

import (
  "testing"
)

func TestLog2(t *testing.T) {
  test := func(d int, expected string) {
```

```
    calc, _ := ToStringBase(Log2(d), 2, 8)
    if calc != expected {
      t.Error("Expected", expected, "got", calc)
    }
  }

  test(1, "10110001")
}

func TestLog2Overlap(t *testing.T) {
  test := func(d int) {
    first, _ := ToStringBase(Log2(d), 2, 8)
    second, _ := ToStringBase(Log2(d+4), 2, 8)
    if first[4:8] != second[0:4] {
      t.Error("Digits don't overlap. Expected", first[4:8],
        "and", second[0:4], "to be equal")
    }
  }

  test(1)
  test(1000)
  test(1234143)
}
```

7.4 pi.go

```
package bbpformula

import (
  "math"
  "math/big"
)

// Pi computes pi at hexadecimal digit specified, returns a big.Float
func Pi(digit int) (*big.Float) {
  digit-
  res := S(4, 1, digit)
  res.Sub(res, S(2, 4, digit))
  res.Sub(res, S(1, 5, digit))
  res.Sub(res, S(1, 6, digit))

  return FractionalPart(res)
}

// S computes x*Summation(j,d)
func S(x int, j int, d int) (*big.Float) {
  sum := Summation(j, d)
  if x == 1 {
    return sum
  }
  return sum.Mul(sum, IntToFloat(x))
}

// Summation computes {16^d*Sj}, see PDF attached for references
func Summation(j int, d int) (*big.Float) {
  k := 0
  res := IntToFloat(0)

```

```
  // First summation (from 1 to d)
  for k <= d {
    denominator := (8*k)+j
    numerator, err := ModPow(16, d-k, denominator)
    if err != nil {
      // TODO: handle error
    }
    tmpRes := new(big.Float).SetPrec(128).
          Quo(Int64ToFloat(numerator), IntToFloat(denominator))
    res.Add(res, tmpRes)
    k++
  }
  res = FractionalPart(res)

  // Seconds summation (from d+1 to infinite/precision)
  floatPrecision, _ := new(big.Float).SetPrec(128).SetString(PRECISION)
  delta := IntToFloat(1)
  for delta.Cmp(floatPrecision) >= 0 {
    denominator := (8*k)+j
    numerator := math.Pow(16, float64(d-k))
    delta.Quo(big.NewFloat(numerator), IntToFloat(denominator))
    res.Add(res, delta)
    k++
  }

  return FractionalPart(res)
}
```

7.5 pi_test.go

```
package bbpformula

import (
  "testing"
)

func TestS(t *testing.T) {
  // Single tests for partial summations
  // Expected values taken from:
  // http://www.davidhbailey.com/dhbpapers/bbp-alg.pdf, page 4

  test := func(j int, d int, expected string) {
    calc := S(1, j, d).Text('f', 30)[0:30] // truncate at 28 decimal digits
    if calc != expected {
      t.Error("Expected", expected, "got", calc)
    }
  }

  d := 1000000
  test(1, d, "0.1810395338014360678534893462")
  test(4, d, "0.7760655498078074613722975943")
  test(5, d, "0.36245856407075741420683343355")
  test(6, d, "0.3861386739520148480012151865")
}

func TestPi(t *testing.T) {
  test := func(d int, expected string) {
    calc, err := ToStringBase(Pi(d), 16, 8)
```

```
    if err != nil {
      t.Error("Unexpected error", err)
    }
    if calc != expected {
      t.Error("Expected", expected, "got", calc)
    }
  }

  test(1, "243F6A88")
  test(1000001, "6C65E52C")
}

func TestPiOverlap(t *testing.T) {
  test := func(d int) {
    first, _ := ToStringBase(Pi(d), 16, 8)
    second, _ := ToStringBase(Pi(d+4), 16, 8)
    if first[4:8] != second[0:4] {
      t.Error("Digits don't overlap. Expected", first[4:8],
        "and", second[0:4], "to be equal")
    }
  }

  test(1)
  test(10)
  test(12323)
}
```

7.6 utils.go

```
package bbpformula

import (
  "math/big"
  "errors"
)

// PRECISION Stop iterating when difference is smaller that this value
// it gives 23-24 exact digits calculating Summations for pi for reasonable d
const PRECISION string = "0.0000000000000000000000000001"

// IntToFloat create a big.Float from a simple int
func IntToFloat(x int) (*big.Float) {
  return Int64ToFloat(int64(x))
}

// Int64ToFloat create a big.Float from a int64
func Int64ToFloat(x int64) (*big.Float) {
  return new(big.Float).SetPrec(128).SetInt64(x)
}

// WholePart extracts integer part of a big.Float
func WholePart(number *big.Float) (int64) {
  whole, _ := number.Int64()
  return whole
}

// Floor performs the floor operation on big.Float, returns a int64
func Floor(number *big.Float) (int64) {
```

```
  if number.Sign() >= 0 {
    return WholePart(number)
  }

  return WholePart(number) - 1
}

// FractionalPart extracts fractional part of a big.Float
// defined as x-floor(x), for negative numbers too
func FractionalPart(number *big.Float) (*big.Float) {
  return new(big.Float).Sub(number, Int64ToFloat(Floor(number)))
}

// ToStringBase returns a string representing fractional part of number,
// written in specified base, limited in length to digits
func ToStringBase(number *big.Float, base int, length int) (string, error) {
  if base < 2 {
    return "", errors.New("Base must be greater than or equal to 2")
  }

  floatBase := IntToFloat(base)
  tmp := new(big.Float).Copy(number)
  res := make([]byte, length)

  for i := 0; i < length && tmp.Sign() != 0; i++ {
    frac := FractionalPart(tmp)
    tmp = tmp.Mul(floatBase, frac)
    whole := WholePart(tmp)

    if 0 <= whole && whole <= 9 {
      res[i] = byte('0' + whole)
    } else {
      res[i] = byte('A' + whole - 10)
    }

  }

  return string(res), nil
}

// ModPow computes (b^n) modulo k, in a memory efficient way
// This is the Right-to-Left binary exponentiation
// For references: Art of Programming vol. 2, par. 4.6.3
func ModPow(b int, n int, k int) (int64, error) {
  if k < 1 {
    return -1, errors.New("Modulo must be greater than or equal to 1")
  }

    if k == 1 {
    return 0, nil
  }

    b = b % k
    var result int64 = 1

    for n > 0 {
        if n % 2 == 1 {
      result = (result * int64(b)) % int64(k)
    }
        n = n / 2
        b = (b * b) % k
```

```
    }

    return result, nil
}
```

7.7 utils_test.go

```
package bbpformula

import (
  "math/big"
  "testing"
)

func TestEnvFloat(t *testing.T) {
  v49 := new(big.Float).SetInt64(49)
  v1  := new(big.Float).SetInt64(1)
  res  := new(big.Float).SetPrec(128)

  res.Quo(v1, v49)        // res = 1/49
  res.Mul(res, v49)        // res = res * 49
  isOne := res.Cmp(v1) == 0    // res == 1

  if res.Prec() != 128 {
    t.Error("Expected to use 128-bit float, instead using", res.Prec())
  }

  if !isOne {
    t.Error("Expected (1/49 * 49 == 1) to be true")
  }
}

func TestModPow(t *testing.T) {
  _, err := ModPow(2, 100, 0)
  if err == nil {
    t.Error("Test an invalid modulo 0, no error were raised")
  }

  test := func(base int, exp int, mod int, expected int64) {
    res, err := ModPow(base, exp, mod)
    if err != nil {
      t.Error("Unexpected error:", err)
    }
    if res != expected {
      t.Error("Expected 0, got:", res)
    }
  }

  test(2, 100, 2, 0)
  test(50, 19800, 6, 4)
  test(367, 447, 739, 687)
}

func TestWholePart(t *testing.T) {
  test := func(number float64, expected int64) {
    x := big.NewFloat(number)
    if WholePart(x) != int64(expected) {
      t.Error("Expected", expected, "got", WholePart(x))
```

```go
    }
  }

  test(456., 456)
  test(1.3456, 1)
  test(423512.23441235, 423512)
}

func TestFractionalPart(t *testing.T) {
  test := func(number float64, expected float64) {
    numFloat := big.NewFloat(number)
    expectedFloat := big.NewFloat(expected)
    frac := FractionalPart(numFloat)
    if expectedFloat.Cmp(frac) != 0 {
      t.Error("Expected", expectedFloat, "got", frac)
    }
  }

  test(0.5, 0.5)
  test(414.25, 0.25)
  test(-16.25, 0.75)
}

func TestToStringBase(t *testing.T) {
  test := func(number float64, base int, expected string) {
    num := big.NewFloat(number)
    str, err := ToStringBase(num, base, 4)
    if err != nil {
      t.Error("Unexpected error:", err)
    }
    if str != expected {
      t.Error("Expected", expected, "got", str)
    }
  }

  test(0.69, 2, "1011")
  test(3.1415926, 16, "243F")
}
```