

Esperienze di Programmazione

Algoritmo BBP per π

Antonio Pitasi

03/06/2017

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:

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

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 partendo dalla formula per il calcolo di $\log(2)$ potessero calcolarne le cifre binarie a partire da una posizione arbitraria.

Il loro ragionamento parte dalla formula scritta nel capitolo precedente:

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

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 frazionaria¹.

Si effettuano alcune trasformazioni:

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

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 Pi Greco

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 :

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

¹La definizione di parte frazionaria è: $\{x\} = x - \lfloor x \rfloor$.

²Infatti $2^{d-k} < 1$ 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 *Sezione 3.1*.

3 Implementazione degli algoritmi

3.1 Elevamento a potenza in modulo

Operazioni come $b^e \bmod k$ 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 della macchina utilizzata. In particolare Bailey stesso durante dei test ha ottenuto almeno nove cifre corrette usando aritmetica a 64 bit, mentre con 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, in pratica 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 : 243F6A88, e le cifre a partire dalla quinta: 6A8885A3. 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 di stare usando veramente 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:

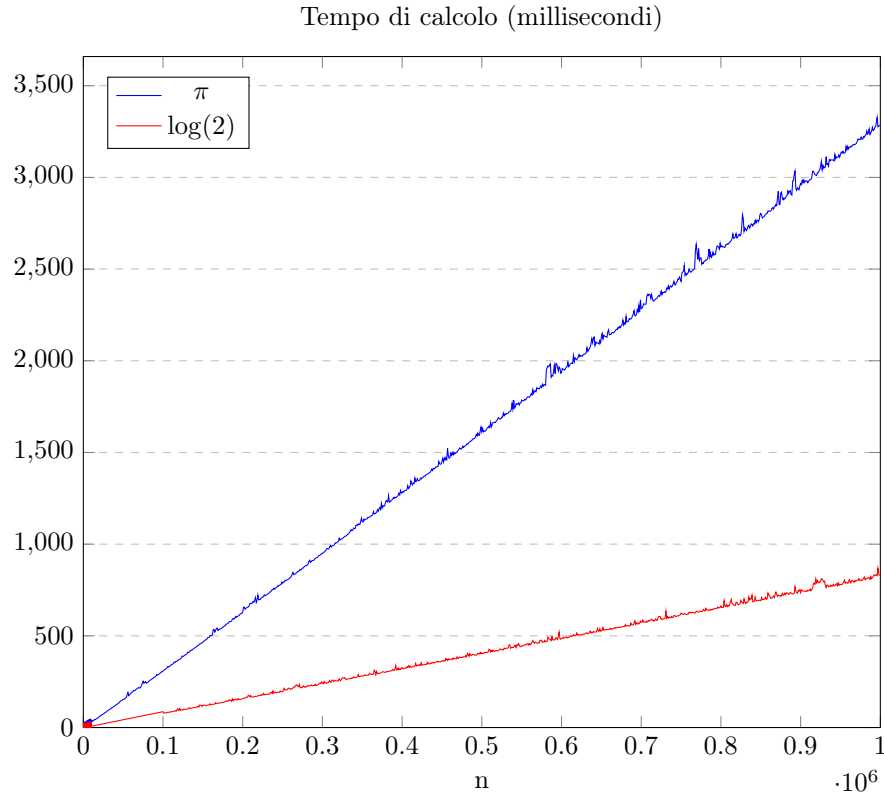
Binary digits of log2 starting at position 1: **10110001**
Binary digits of log2 starting at position 100: **00011111**
Binary digits of log2 starting at position 10000: **00101101**
Binary digits of log2 starting at position 1000000: **11010100**
Binary digits of log2 starting at position 100000000: **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 100000000: **17AF5863**

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 :

³Link: <http://www.wolframalpha.com/>



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 .

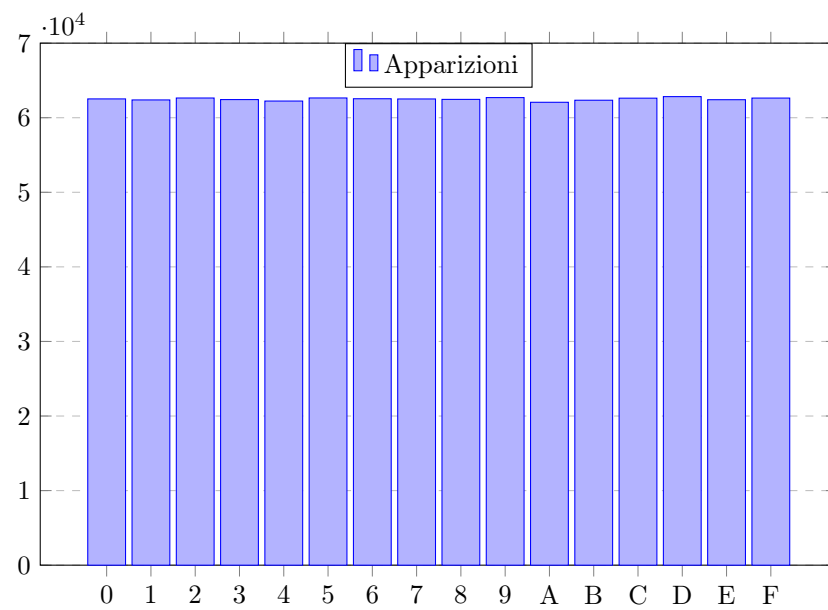
5.2 Possibili (futuri) utilizzi

Ammettendo che le cifre in π compaiano equiprobabilmente⁴, 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*.

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:

⁴Non è stato dimostrato per il momento.



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

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6     "strconv"
7     "time"
8     b "./bbpformula"
9 )
10
11 func main() {
12     filename, args := os.Args[0], os.Args[1:]
13
14     invalidArguments :=
15         len(args) != 2 ||
16         (args[0] != "log2" && args[0] != "pi")
17
18     if invalidArguments {
19         help(filename)
20         return
21     }
22
23     // Parse second argument as integer
24     d, err := strconv.Atoi(args[1])
25     if err != nil || d <= 0 {
26         fmt.Println("ERROR Invalid number n provided.")
27         fmt.Println("Try running:", filename, "pi 10000")
28         return
29     }
30
31     // Start computing
32     var res string
33     fmt.Printf("Computing...")
34     timeStart := time.Now()
35
36     // Case switch for log2 or pi
37     if args[0] == "log2" {
38         res, err = b.ToStringBase(b.Log2(d), 2, 8)
39         fmt.Printf("\rBinary")
40     } else if args[0] == "pi" {
41         res, err = b.ToStringBase(b.Pi(d), 16, 8)
42         fmt.Printf("\rHexadecimal")
43     }
```

```

43     }
44
45     // Print results
46     elapsed := time.Since(timeStart)
47     fmt.Printf(" digits of %s starting at position %s: ",
48         args[0], args[1])
49     if err != nil {
50         fmt.Println("Unexpected error:", err)
51     }
52     fmt.Println(res)
53     fmt.Println("Computation took", elapsed)
54 }
55
56 func help(filename string) {
57     fmt.Println("This program computes n-th digit of binary log2 or
58         ↪ hexadecimal pi.")
59     fmt.Println("Usage:", filename, "[log2/pi] <n>")
60 }

```

7.2 bbpformula/log2.go

```

1 package bbpformula
2
3 import (
4     "math"
5     "math/big"
6 )
7
8 // Log2 computes log2 at binary digit specified, returns a
9 ↪ big.Float
10 func Log2(digit int) (*big.Float) {
11     digit--
12     k := 1
13     res := big.NewFloat(0).SetPrec(128)
14
15     // First summation (from 1 to d)
16     for k <= digit {
17         numerator, err := ModPow(2, digit-k, k)
18         if err != nil {
19             // TODO: handle err
20         }
21         tmpRes := new(big.Float).SetPrec(128).
22             Quo(Int64ToFloat(numerator), IntToFloat(k)) // quotient =
23             ↪ modpow/k
24         res.Add(res, tmpRes)
25         k++
26     }
27     return res
28 }

```

```

24     }
25     res = FractionalPart(res)
26
27     // Seconds summation (from d+1 to infinite/precision)
28     floatPrecision, _ := new(big.Float).SetString(PRECISION)
29     delta := IntToFloat(1)
30     for delta.Cmp(floatPrecision) >= 0 {
31         denominator := IntToFloat(k)
32         numerator := big.NewFloat(math.Pow(2, float64(digit-k)))
33         delta.Quo(numerator, denominator)
34         res.Add(res, delta)
35         k++
36     }
37
38     return res
39 }

```

7.3 bbpformula/log2_test.go

```

1  package bbpformula
2
3  import (
4      "testing"
5  )
6
7  func TestLog2(t *testing.T) {
8      test := func(d int, expected string) {
9          calc, _ := ToStringBase(Log2(d), 2, 8)
10         if calc != expected {
11             t.Error("Expected", expected, "got", calc)
12         }
13     }
14
15     test(1, "10110001")
16 }
17
18 func TestLog2Overlap(t *testing.T) {
19     test := func(d int) {
20         first, _ := ToStringBase(Log2(d), 2, 8)
21         second, _ := ToStringBase(Log2(d+4), 2, 8)
22         if first[4:8] != second[0:4] {
23             t.Error("Digits don't overlap. Expected", first[4:8],
24                 "and", second[0:4], "to be equal")
25         }
26     }
27 }

```

```

28     test(1)
29     test(1000)
30     test(1234143)
31 }

```

7.4 bbpformula/pi.go

```

1  package bbpformula
2
3  import (
4      "math"
5      "math/big"
6  )
7
8  // Pi computes pi at hexadecimal digit specified, returns a
9  ↪ big.Float
10 func Pi(digit int) (*big.Float) {
11     digit--
12     res := S(4, 1, digit)
13     res.Sub(res, S(2, 4, digit))
14     res.Sub(res, S(1, 5, digit))
15     res.Sub(res, S(1, 6, digit))
16     return FractionalPart(res)
17 }
18
19 // S computes x*Summation(j,d)
20 func S(x int, j int, d int) (*big.Float) {
21     sum := Summation(j, d)
22     if x == 1 {
23         return sum
24     }
25     return sum.Mul(sum, IntToFloat(x))
26 }
27
28 // Summation computes {16^d*Sj}, see PDF attached for references
29 func Summation(j int, d int) (*big.Float) {
30     k := 0
31     res := IntToFloat(0)
32
33     // First summation (from 1 to d)
34     for k <= d {
35         denominator := (8*k)+j
36         numerator, err := ModPow(16, d-k, denominator)
37         if err != nil {
38             // TODO: handle error

```

```

39     }
40     tmpRes := new(big.Float).SetPrec(128).
41         Quo(Int64ToFloat(numerator), IntToFloat(denominator))
42     res.Add(res, tmpRes)
43     k++
44 }
45 res = FractionalPart(res)
46
47 // Seconds summation (from d+1 to infinite/precision)
48 floatPrecision, _ :=
49     ↪ new(big.Float).SetPrec(128).SetString(PRECISION)
50 delta := IntToFloat(1)
51 for delta.Cmp(floatPrecision) >= 0 {
52     denominator := (8*k)+j
53     numerator := math.Pow(16, float64(d-k))
54     delta.Quo(big.NewFloat(numerator), IntToFloat(denominator))
55     res.Add(res, delta)
56     k++
57 }
58 return FractionalPart(res)
59 }

```

7.5 bbpformula/pi_test.go

```

1 package bbpformula
2
3 import (
4     "testing"
5 )
6
7 func TestS(t *testing.T) {
8     // Single tests for partial summations
9     // Expected values taken from:
10    // http://www.davidhbailey.com/dhbpapers/bbp-alg.pdf, page 4
11
12    test := func(j int, d int, expected string) {
13        calc := S(1, j, d).Text('f', 30)[0:30] // truncate at 28
14        ↪ decimal digits
15        if calc != expected {
16            t.Error("Expected", expected, "got", calc)
17        }
18    }
19
20    d := 1000000
21    test(1, d, "0.1810395338014360678534893462")

```

```

21     test(4, d, "0.7760655498078074613722975943")
22     test(5, d, "0.3624585640705741420683343355")
23     test(6, d, "0.3861386739520148480012151865")
24 }
25
26 func TestPi(t *testing.T) {
27     test := func(d int, expected string) {
28         calc, err := ToStringBase(Pi(d), 16, 8)
29         if err != nil {
30             t.Error("Unexpected error", err)
31         }
32         if calc != expected {
33             t.Error("Expected", expected, "got", calc)
34         }
35     }
36
37     test(1, "243F6A88")
38     test(1000001, "6C65E52C")
39 }
40
41 func TestPiOverlap(t *testing.T) {
42     test := func(d int) {
43         first, _ := ToStringBase(Pi(d), 16, 8)
44         second, _ := ToStringBase(Pi(d+4), 16, 8)
45         if first[4:8] != second[0:4] {
46             t.Error("Digits don't overlap. Expected", first[4:8],
47                 "and", second[0:4], "to be equal")
48         }
49     }
50
51     test(1)
52     test(10)
53     test(12323)
54 }

```

7.6 bbpformula/utils.go

```

1 package bbpformula
2
3 import (
4     "math/big"
5     "errors"
6 )
7
8 // PRECISION Stop iterating when difference is smaller that this
   ↪ value

```



```

50 floatBase := IntToFloat(base)
51 tmp := new(big.Float).Copy(number)
52 res := make([]byte, length)
53
54 for i := 0; i < length && tmp.Sign() != 0; i++ {
55     frac := FractionalPart(tmp)
56     tmp = tmp.Mul(floatBase, frac)
57     whole := WholePart(tmp)
58
59     if 0 <= whole && whole <= 9 {
60         res[i] = byte('0' + whole)
61     } else {
62         res[i] = byte('A' + whole - 10)
63     }
64
65 }
66
67 return string(res), nil
68 }
69
70 // ModPow computes (b^n) modulo k, in a memory efficient way
71 // This is the Right-to-Left binary exponentiation
72 // For references: Art of Programming vol. 2, par. 4.6.3
73 func ModPow(b int, n int, k int) (int64, error) {
74     if k < 1 {
75         return -1, errors.New("Modulo must be greater than or equal
76             ↳ to 1")
77     }
78
79     if k == 1 {
80         return 0, nil
81     }
82
83     b = b % k
84     var result int64 = 1
85
86     for n > 0 {
87         if n % 2 == 1 {
88             result = (result * int64(b)) % int64(k)
89         }
90         n = n / 2
91         b = (b * b) % k
92     }
93
94     return result, nil

```


7.7 bbpformula/utils_test.go

```
1 package bbpformula
2
3 import (
4     "math/big"
5     "testing"
6 )
7
8 func TestEnvFloat(t *testing.T) {
9     v49 := new(big.Float).SetInt64(49)
10    v1  := new(big.Float).SetInt64(1)
11    res := new(big.Float).SetPrec(128)
12
13    res.Quo(v1, v49)           // res = 1/49
14    res.Mul(res, v49)         // res = res * 49
15    isOne := res.Cmp(v1) == 0 // res == 1
16
17    if res.Prec() != 128 {
18        t.Error("Expected to use 128-bit float, instead using",
19            ↪ res.Prec())
20    }
21
22    if !isOne {
23        t.Error("Expected (1/49 * 49 == 1) to be true")
24    }
25
26    func TestModPow(t *testing.T) {
27        _, err := ModPow(2, 100, 0)
28        if err == nil {
29            t.Error("Test an invalid modulo 0, no error were raised")
30        }
31
32        test := func(base int, exp int, mod int, expected int64) {
33            res, err := ModPow(base, exp, mod)
34            if err != nil {
35                t.Error("Unexpected error:", err)
36            }
37            if res != expected {
38                t.Error("Expected 0, got:", res)
39            }
40        }
41
42        test(2, 100, 2, 0)
43        test(50, 19800, 6, 4)
```

```

44     test(367, 447, 739, 687)
45 }
46
47 func TestWholePart(t *testing.T) {
48     test := func(number float64, expected int64) {
49         x := big.NewFloat(number)
50         if WholePart(x) != int64(expected) {
51             t.Error("Expected", expected, "got", WholePart(x))
52         }
53     }
54
55     test(456., 456)
56     test(1.3456, 1)
57     test(423512.23441235, 423512)
58 }
59
60 func TestFractionalPart(t *testing.T) {
61     test := func(number float64, expected float64) {
62         numFloat := big.NewFloat(number)
63         expectedFloat := big.NewFloat(expected)
64         frac := FractionalPart(numFloat)
65         if expectedFloat.Cmp(frac) != 0 {
66             t.Error("Expected", expectedFloat, "got", frac)
67         }
68     }
69
70     test(0.5, 0.5)
71     test(414.25, 0.25)
72     test(-16.25, 0.75)
73 }
74
75 func TestToStringBase(t *testing.T) {
76     test := func(number float64, base int, expected string) {
77         num := big.NewFloat(number)
78         str, err := ToStringBase(num, base, 4)
79         if err != nil {
80             t.Error("Unexpected error:", err)
81         }
82         if str != expected {
83             t.Error("Expected", expected, "got", str)
84         }
85     }
86
87     test(0.69, 2, "1011")
88     test(3.1415926, 16, "243F")
89 }

```