Silna i słaba liczba

Kamil Niemczyk

17 kwietnia 2024

1 Wstęp

Zadanie polegało na obliczeniu dwóch zależnych liczb od imienia i nazwiska. Było trzeba najpierw wygenerowac nick, który składałby się z trzech pierwszych liter imienia i trzech pierwszych liter nazwiska i na końcu je połączyć, żeby stworzyć ciąg znaków długości sześć. Potem przekonwertować te litery na odpowiadające im kody ASCII. Następnie obliczenia takiej liczby dla której silnia jej zawiera w sobie wszystkie warości numeryczne kodów ASCII tego nicku. Ostatnim podpunktem tego zadania było obliczenie drugiej liczby, która byłaby liczbą wywołań funckji Fibonacciego, której wynik byłby najbardziej zbliżony do liczby poprzedniej.

2 Program

2.1 Plik main.go

```go
package main

import (
    "fmt"
)

func main() {
    name := inputNick()
    ascii := generateNickASCII(name)
    strongNumber := int(strongNumber(ascii))
    weakNumberList := weakNumberList(30)
    weakNumber := weakNumberFind(strongNumber, weakNumberList)
    fmt.Println("Nick:", name)
    fmt.Println("ASCII:", ascii)
    fmt.Println("Strong number: ", strongNumber)
    fmt.Println("Fibonnaci calls counter: ", weakNumberList)
    fmt.Println("Weak number: ", weakNumber)
}

```

Plik main.go

Plik main.go zawiera w sobie całą konfiguracje jak program będzie się wyświetlał użytkownikowi. Na poczatku generuje nick, następnie z nicku tworzy tablice kodów ASCII odpowiadający literom z nicku, następnie oblicza silną liczbe, potem generuje mape liczby wystąpień wywołań ciągu Fibonacciego dla argumentu 30 i generuje słabą liczbę. Na końcu te wyniki wyświetla użytkownikowi. Poniżej zdjęcia obrazujące wyniki tego programu.

```
$ go run *.go
Enter your name:
Kamil
Enter your last name:
Niemczyk
Nick: kamnie
ASCII: [107 97 109 110 105 101]
Strong number:  297
Fibonnaci calls counter:  map[0:514229 1:832040 2:514229 3:317811 4:196418 5:121
393 6:75025 7:46368 8:28657 9:17711 10:10946 11:6765 12:4181 13:2584 14:1597 15:
987 16:610 17:377 18:233 19:144 20:89 21:55 22:34 23:21 24:13 25:8 26:5 27:3 28:
2 29:1 30:1]
Weak number:  18
```

Outputs

2.2 Plik nickreader.go

```go
package main

import (
    "fmt"
    "strings"
)

func inputNick() string {
    var name string
    var lastname string
    fmt.Println("Enter your name:")
    fmt.Scanln(&name)
    fmt.Println("Enter your last name:")
    fmt.Scanln(&lastname)
    nameToLower := strings.ToLower(name)[:3]
    lastnameLower := strings.ToLower(lastname)[:3]
    nick := nameToLower + lastnameLower
    return nick
}

```

Plik nickreader.go

Plik ten zawiera funkcję inputNick(), która pobiera od użytkownika imię i nazwisko, konwertuje wprowadzone dane na małe litery, bierze po pierwsze 3 znaki z imienia i nazwiska, a na końcu łączy w jednego stringa.

2.3 Plik ascci.go

```go
package main

import "strconv"

func generateNickASCII(name string) []string {
    var ascii []string
    for i := 0; i < len(name); i++ {
        ascii = append(ascii, strconv.Itoa(int(name[i])))
    }
    return ascii
}

```

Plik ascii.go

Ten plik dostaje stringa i potem konwertuje każdą litere na jej numer ASCII i ten numer zmienia na stringa, żeby uzyskać stringa o wartości numeru ASCII danej litery i na końcu dodaje go do listy.

2.4 Plik strongNumber.go

```go
package main

import (
    "math/big"
    "strings"
)

func factorial(n int64) *big.Int {
    result := big.NewInt(1)
    for i := int64(2); i <= n; i++ {
        result.Mul(result, big.NewInt(i))
    }
    return result
}

func factorialString(n int64) string {
    return factorial(n).String()
}

func contains(s string, substr string) bool {
    return strings.Contains(s, substr)
}
```

Plik strongNumber.go 1/2

```go
func ifContainsAllNumbers(asciiList []string, factorialString string) bool {
    for _, ascii := range asciiList {
        if !contains(factorialString, ascii) {
            return false
        }
    }
    return true
}

func strongNumber(asciiList []string) int64 {
    var i int64 = 1
    for {
        factorialString := factorialString(i)
        if ifContainsAllNumbers(asciiList, factorialString) {
            return i
        }
        i++
    }
}
```

Plik strongNumber.go 2/2

Ten plik tworzy nam silną liczbe. Funckja factorial liczy silnie podanej liczby i w efekcie zwraca tą liczbe typu *big.int, ponieważ to może być absurdalnie wielka liczba. Funckja factorialString zmienia liczbe zwróconą przez funckje factorial na stringa. Funkcja contains sprawdza czy w danym ciągu znaków znajduje się dany podciąg i zwaraca wartość boolową. Funckja ifContainsAllNumbers w argumentach przyjmuje naszą poprzednio wygenerowaną tablice kodów ASCII i jakąś liczbę, którą została wygenerowana przez silnie i iteracyjnie przechodząc po każdym elemencie tablicy ASCII sprawdza czy każdy numer zawiera się w tej silni, jeśli tak to zwraca true jeśli nie to zwraca false. Ostatnia funckja strongNumber sprawdza jaki argument musi przyjąć funckja factorialString, żeby zawierała wszystkie te liczby w sobie i na końcu zwraca jaka liczba po zastosowaniu silni spełnia wszystkie te warunki.

2.5 Plik weakNumber.go

```go
func fibonacci(n int, result map[int]int) int {
    if n <= 1 {
        result[n]++
        return n
    } else {
        result[n]++
    }
    return fibonacci(n-1, result) + fibonacci(n-2, result)
}
func weakNumberList(n int) map[int]int {
    result := make(map[int]int)
    fibonacci(n, result)
    return result
}
func absolute(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
func weakNumberFind(n int, weakNumberList map[int]int) int {
    key := 0
    diff := 0
    for k, v := range weakNumberList {
        if absolute(n-v) < diff || diff == 0 {
            key = k
            diff = absolute(n - v)
        }
    }
    return key
}
```

Plik weakNumber.go

Funkcja fibonacci oblicza ciąg Fibonacciego i przy każdym wywołaniu pojedynczej funckji dodaje do mapy klucz i wartość, klucz jest równy argumentowi funkcji, a wartość zawsze jest zwiększana o jeden

dla danego klucza i dzięki temu można sprawdzić ile razy funkcja została wywołana rekurencyjnie. Funkcja weakNumberList generuje tą ilość wystąpień za pomocą funkcji poprzedniej. Funckja absolute liczy wartość bezwzględną liczby. Funkcja weakNumberFind() za pomocą podanej liczby i liczby wystąpień Fibonacciego spradza za pomocą funkcji absolute, najbardziej zbliżoną ilość wystąpień tej liczby i zwraca jej klucz.

3 Ciąg Fibonacciego

Algorytm ciągu Fibonacciego jest wykonywany rekurencyjnie co oznacza, że każda funckja wywołuje na końcu funkcje do momentu aż nie zajdzie jakiś warunek. W przypadku ciągu Fibonacciego tym warunkiem jest to, że argument funkcji jest równy 1 lub mniejszy to wtedy funckja zwraca po prostu wartość i nie kontynuuje tego piętrzenia się funkcji. Algorytm Fibonacciego nie wywołuje tylko jednej funckji na końcu ale dwie. Złożoność czasowa ciągu rekurencyjnego Fibonacciego jest wykładnicza co oznacza, że ilość wywołań funckji z mniejszymi argumentami rośnie bardzo szybko. Porównałbym to trochę do efektu kuli śnieżnej, która na początku ma małą wielkość, ale z każdym obrotem staje się coraz większa i przez zwiększanie swojej wielkości zwiększa się też jej powierzchnia do której może przylepiać się śnieg, więc im jest większa tym będzie szybciej rosnąć. Nawet ten efekt można zaobserwować przy wywołaniu Fibonacciego dla argumentu 30 tak jak było w programie. Na zdjęciu umieszczonym na początku dokumentu widać, że na przykład dla wiekszych wartości ilość wywołań rośnie o jeden może dwa, ale im mniejszy jest argument tej funckji to ilość wywołań potrafi rosnąć nawet o pareset tysięcy. Dlatego przypuszczam, że znalezienie wyniku dla ciągu Fibonacciego byłoby bardzo obciążające dla komputera i wyniku dla tego wywołania prawdopodobnie bym nie ujrzał w najbliższym czasie.

4 Funkcja Ackermanna

Funckja Ackermanna w porównaniu do funckji Fibonacciego jest bardziej złożona. W funckji Ackermanna pomimo stosunkowo małych liczb można wygenerowac naprawdę ogromne liczby. Co wyróżnia funkcję Ackermanna jest to, że posiada rekurencje w rekurencji. W większości przypadków jako wywołanie na końcu wywołują samą siebie z argumentem wywołania się co powoduje bardzo szybkie rośnięcie ilości wywołań jak i samego wyniku. Na przykład funckja Ackermanna dla argumentów (1,1) wywoła się 3 razy.

- Wywoła się raz dla funckji ackermann(1,1)
- Następnie wywoła się dla funkcji ackermann(0, ackermann(1,0))
- I wywoła wewnętrzną funkcje ackermann(1,0)

Na trudniejszych przykładach łatwiej byłoby zauważyć jak szybko rośnie liczba wywołań tej funkcji, ale jestem pewny, że przy którymś wywołaniu pomyliłbym się z jej liczeniem. Ale można na pewno zauważyc, że ta funkcja ma ogromny potencjał na tworzenie absurdalnej ilości wywołań pomniejszych funkcji.

5 Wnioski

Zadanie powyżej nauczyło mnie jak radzić sobie z dużymi liczbami w języku Go. Do tego była to też moja pierwsza styczność z funckją Ackermanna co spowodowało, że teraz myślac o szybko rosnącej rekurencyjnej funkcji nie będę myślał o Fibonaccim, ale o Ackermannie. To zadanie jeszcze uświadomiło mi jakim użytecznym sposobem mogą być mapy w Go.