EDIN01 – Project 1

Tony Jin (to1643ji-s@student.lu.se)

November 23, 2018

1 Exercise 1

Let N be a 25 digit number and $\lfloor\sqrt{N}\rfloor$ is a 12 digit number.

The 12 digit is obviously bounded by $10^{11} \leq \lfloor\sqrt{N}\rfloor < 10^{12}$, where $\lfloor\sqrt{N}\rfloor \in \mathbb{N}$. Thus, the seconds needed to factor the 25 digit number is a positive integer t bounded by

$$10^4 \leq t < 10^5,$$

as $10^{11} \cdot 10^{-7} = 10^4$ and $10^{12} \cdot 10^{-7} = 10^5$.

2 Exercise 2

We know that $\pi(10^{11}) = 4118054813$ and $\pi(10^{12}) = 37607912018$ [1]. Our new bounds for t is then

$$4118054813 \cdot 10^{-7} \leq t < 37607912018 \cdot 10^{-7}.$$

Assuming that every integer is 12 digts number with an equivalent binary representation of 40 bits, the total amount of memory needed is 164722192520 bits $\approx$ 20 gigabyte. A student budget is enough.

3 Exercise 3

The prime facrtors for 253808893609854792191119 are $p = 496469737391, q = 511227320609$.

The quadratic sieve program below is written in Go.

```
package main

import (
  "fmt"
  "math"
  "math/big"
  "log"
  "os"
  "strings"
  "reflect"
  "strconv"
  "os/exec"
  "bufio"
)

// Big pi multiplication
func mult(factors []*big.Int) *big.Int {
  product := big.NewInt(1)
  for i := 0; i < len(factors); i++ {
```

```
    product.Mul(product, factors[i])
  }

  return product
}

// Return all primes less than ‘value‘
func Factorbase(value int) []*big.Int {
  var primes []*big.Int
  f := make([]bool, value)
  for i := 2; i <= int(math.Sqrt(float64(value))); i++ {
    if f[i] == false {
      for j := i * i; j < value; j += i {
        f[j] = true
      }
    }
  }
  for i := 2; i < value; i++ {
    if f[i] == false {
      primes = append(primes, big.NewInt(int64(i)))
    }
  }

  return primes
}

// Check if a number ‘n‘ is B-smooth over a factorbase F
func FactorOverF(F []*big.Int, n *big.Int) ([]int, []*big.Int, error) {
  nCopy := new(big.Int).Set(n)
  var a []*big.Int

  vec := make([]int, len(F))
  i := 0

  for i < len(F) {
    // Check if numbers in ‘F‘ divides ‘n‘
    if new(big.Int).Rem(nCopy, F[i]).Cmp(big.NewInt(0)) == 0 {
      // Append to ‘a‘ and increment the right index in ‘vec‘
      a = append(a, F[i])
      vec[i] = ((vec[i] + 1) % 2)

      // Divide away the factor just added
      nCopy = nCopy.Div(nCopy, F[i])
    } else if F[i].Cmp(n) == 1 {
      break
    } else {
      i = i + 1
    }
  }

  if mult(a).Cmp(n) != 0 {
    return nil, nil, nil
  }

  return vec, a, nil
}

func GenerateNumber(N *big.Int, k *big.Int, j *big.Int) *big.Int {
  kN := new(big.Int).Mul(k, N)
```

```go
  root := new(big.Int).Sqrt(kN)
  r := root.Add(root, j)

  return r
}

type group struct {
  point []int64
  r *big.Int
  r2 *big.Int
  factors []*big.Int
  vector []int
}

func solve(solution []string, G []group, N *big.Int) *big.Int {
  // Create left hand side and right hand side, non-multiplicated
  lhsSlice := make([]*big.Int, 0)
  rhsSlice := make([]*big.Int, 0)
  for i, v := range solution {
    if v == "1" {
      lhsSlice = append(lhsSlice, G[i].r)
      rhsSlice = append(rhsSlice, G[i].factors...)
    }
  }

  // Multiplicate the slice of lhs
  lhs := mult(lhsSlice)
  lhs = lhs.Mod(lhs, N)

  // Multiplicate the slice of rhs
  rhs := mult(rhsSlice)
  rhs = rhs.Sqrt(rhs)
  rhs = rhs.Mod(rhs, N)

  // Calculate gcd(|lhs - rhs|, N)
  gcd := new(big.Int).GCD(nil, nil, new(big.Int).Abs(new(big.Int).Sub(lhs, rhs)), N)
  return gcd
}

func main() {
  // Primes are 496469737391 and 511227320609
  N, _ := new(big.Int).SetString("253808893609854792191119", 10)

  F := Factorbase(2000)
  G := make([]group, 0)

  log.Println("Add random points to try...")

  // Add k, j-points to slice ‘P‘
  for k := 1; k < 1500; k++ {
    for j := 1; j < 1500; j++ {
      p := []int64{int64(k), int64(j)}
      G = append(G, group{p, nil, nil, nil, nil})
    }
  }

  // Generate numbers based on the points from P
  for i := 0; i < len(G); i++ {
    r := GenerateNumber(
```

```
    N, // N
    big.NewInt(G[i].point[0]), // k
    big.NewInt(G[i].point[1]), // j
  )

  G[i].r = r
  G[i].r2 = new(big.Int).Exp(r, big.NewInt(2), N)
}

log.Println("Generating matrix...")
// Add vector and factors to G iff it passes checks
for i := 0; i < len(G); i++ {
  vector, factors, _ := FactorOverF(F, G[i].r2)

  dup := false
  for i := 0; i < len(G); i++ {
    if reflect.DeepEqual(G[i].vector, vector) == true {
      dup = true
      break
    }
  }

  if err != nil || dup == true {
    G[i].vector = nil
  }

  if dup == false {
    G[i].vector = vector
    G[i].factors = factors
  }
}

// Create a copy of G for reducing slice, and initalize a zero vector
// for checking
Gcopy := G[:0]
zero := make([]int, len(F))
for i := 0; i < len(zero); i++ {
  zero[i] = 0
}

// Reduce slice of degenerate cases, etc.
for _, g := range G {
  if g.vector != nil && reflect.DeepEqual(g.vector, zero) != true {
    Gcopy = append(Gcopy, g)
  }
}
G = Gcopy

log.Println("Matrix finished and printed!")

// Write to file
file, err := os.Create("input")
if err != nil {
  log.Fatal("Cannot create file", err)
}
defer file.Close()

fmt.Fprintf(file, strconv.Itoa(len(G)))
fmt.Fprintf(file, " ")
```

```
  fmt.Fprintf(file, strconv.Itoa(len(F)))
  fmt.Fprintf(file, "\n")
  for i := 0; i < len(G); i++ {
    fmt.Fprintf(file, strings.Trim(strings.Join(strings.Fields(fmt.Sprint(G[i].vector)), " ")
    fmt.Fprintf(file, "\n")
  }

  // Run GaussBin.exe
  cmd := exec.Command("GaussBin.exe", "input", "output")
  log.Printf("Running GaussBin.exe...")
  err = cmd.Run()
  log.Printf("Command finished with error: %v", err)

  // Read file line by line
  solutions := make([][]string, 0)
  outFile, err := os.Open("output")
  if err != nil {
    log.Fatal(err)
  }
  defer outFile.Close()

  scanner := bufio.NewScanner(outFile)
  for scanner.Scan() {
    solutions = append(solutions, strings.Fields(scanner.Text()))
  }

  if err := scanner.Err(); err != nil {
    log.Fatal(err)
  }

  solutions = solutions[1:]

  log.Println("Trying the solution vectors...")

  var factor *big.Int
  for i := 0; i < len(solutions); i++ {
    gcd := solve(solutions[i], G, N)
    if gcd.Cmp(big.NewInt(0)) != 0 && gcd.Cmp(big.NewInt(1)) != 0 {
      fmt.Println("===========Found factor!===========")
      factor = gcd
      break
    }
  }

  fmt.Println(factor)
}
```

References

[1] Prime Counting function,
`http://mathworld.wolfram.com/PrimeCountingFunction.html`