Algoritmos (em alta precisão)

Introdução

Estudamos uma pequena série de métodos com o objetivo de construir ferramentas pra compreendermos criptossistemas como o RSA, El Gammal e protocolos criptográficos populares como o Diffie-Hellman. O quanto compreendemos sobre esses métodos? Conseguimos colocá-los em alta precisão? Os computadores são excelentes coadjuvantes na tarefa de expressarmos ideias em alta precisão: se penso que compreendi algum procedimento, uma forma eficaz de verificação é transmiti-lo a um computador.

Regras

1. O que entregar? Um **arquivo-texto** contendo uma apresentação em língua portuguesa dos exercícios abaixo. Os exercícios abaixo são procedimentos de computador a serem escritos, então você deve apresentá-los usando a língua portuguesa e exibindo os *relevantes procedimentos* que você escreveu.

   Definição (arquivo-texto)**.** Neste curso definimos que um *arquivo-texto* é uma sequência de caracteres Unicode codificados por UTF-8.

2. Escreva Python. (Ensinaremos Python a você. Pergunte-nos como.)

3. Se o exercício pede por um procedimento `f` que você venha a escrever usando outros procedimentos auxiliares `g`, `h`, você deve apresentar `f` e não `g` e `h`, exceto quando `g` e `h` fizerem parte essencial da estratégia de `f` em si. Por quê? Porque queremos ver sua expressão do método e não sua contabilidade inteira. Por exemplo, seu procedimento `mdc(m, n)` pode vir a usar um procedimento `divides(a,b)` como auxiliar. Basta nos apresentar o que `divides(a,b)` faz em língua portuguesa e assim não precisamos o código de `divides`: ele não faz parte essencialmente do método de Euclides. É claro que esse julgamento é subjetivo: escrever e apresentar demanda sensibilidade. Precisamos desse tipo de sofisticação pra escrever trabalhos acadêmicos e você já está a caminho do seu trabalho de conclusão de curso.

4. Qualquer procedimento solicitado nos exercícios abaixo não conterá uma chamada a procedimentos como `printf`, `print`, `input` e outros que leiam dados da entrada padrão ou que imprimam dados na saída padrão. Serão procedimentos silenciosos, portanto. (Eles não falam e nem ouvem.) Pergunte-nos por quê.

É claro que você pode usar `printf`, `input` *et cetera* em procedimentos auxiliares pertencentes às idiossincracias dos seus protocolos de trabalho, mas esses procedimentos auxiliares seus não nos serão apresentados — estamos apenas interessados nos métodos científicos e não em interfaces amigáveis para usá-los.

5. Os procedimentos solicitados nos exercícios abaixo devem sempre consumir pelo menos um argumento e retornar pelo menos um valor. São esses — argumentos e retornos — os mecanismos de comunicação com seus procedimentos. Em outras palavras, a informação que desejamos passá-los é provida como argumentos; a informação que desejamos receber nos é produzida como o retorno de um procedimento.

Exercícios

Exercício 1. Redija uma apresentação de um procedimento necessariamente recursivo e necessariamente chamado `mdc(a, b)` que consome dois inteiros `a, b` e retorna o maior inteiro positivo que divide ambos `a, b`.

Exercício 2. Redija uma apresentação de um procedimento recursivo necessariamente chamado `xmdc(a, b)` que consome dois inteiros a, b e retorna uma tripla `m, x, y` representando a identidade de Bézout.

Exercício 3. Escreva um procedimento chamado `bezout(a, b)` que necessariamente usa seu procedimento `xmdc(a,b)`, obtém os coeficientes x, y cuja existência é garantida por Étienne Bézout e retorna a string `mdc(a,b) = xa + yb`, sendo que `x`, `y`, `a`, `b` tomam seus devidos valores na string, assim como `mdc(a,b)`.

Exercício 4. Rediga uma apresentação de um procedimento chamado `inverse_of(a, m)` que usa seu procedimento `xmdc(a, b)` pra obter o inverso multiplicativo de `a` módulo `m`. O retorno de `inverse_of(a, m)` é necessariamente positivo, ou seja, retorna o representante principal de sua classe de equivalência.

Comentário. Oos representantes principais das classes de equivalência módulo 7 são $0, 1, 2, 3, 4, 5$ e 6. É fato que $[0] = [7]$, mas 7 não é o representante principal de sua classe. Os representantes principais de $\mathbb{Z}_n$ são $0, 1, ..., n - 1$.

Exercício 5. Considere o procedimento `trial` abaixo. Ele termina se o próximo candidato a fator de `n` for igual a `floor(sqrt(n))`. Se não for, ele verifica se `nxt` divide `n`. Se sim, encontramos um fator de n. Se não, tentamos o próximo.

```
def trial(n, nxt = 2):
  assert nxt >= 2, "but the smallest prime is 2, isn't it?"
  assert n >= 2
  if nxt > floor(sqrt(n)):
    return n
  if divides(nxt, n):
    return nxt
  return trial(n, nxt + 1)
```

Esse procedimento é evidentemente *tail recursive*, o que poderia ser otimizado pelo compilador, mas Python por *design* não implementa *tail-call optimization*, então não poderemos fatorar números que não tenham minúsculos fatores primos como 8164489, cujo menor fator é 1031. Reescrevamos o procedimento para:

```
def itrial(n, nxt = 2):
  assert nxt >= 2
  assert n >= 2
  for nxt in range(nxt, floor(sqrt(n)) + 1):
    if divides(nxt, n):
      return nxt
  return n
```

Melhor, não? Redija uma explicação do porquê o procedimento não precisa verificar divisores acima de `floor(sqrt(n))`. (Só isso.)

Comentário. O `i` de `itrial` significa "iterativo", mas não é verdade que "iterativo" é o que não é recursivo. A palavra "iteração" sugere qualquer coisa que se repita. Portanto, qualquer laço é uma iteração e recursão é iteração também.

Exercício 6. O procedimento `itrial` é capaz de identificar quando `n` é primo? Se sim, como se faz? Por que não? Redija sua explicação.

Exercício 7. Escreva um procedimento chamado `is_prime(n)` que retorna `True` quando `n` é primo e `False` quando não é. Se o procedimento `itrial` for capaz de detectar quando `n` é primo, então você precisa necessariamente usá-lo em `is_prime(n)`.

Comentário. O procedimento `is_prime(n)` é um "predicado", isto é, ele necessariamente retorna `True` ou `False` e nenhum outro valor. Um "predicado" é uma propriedade característica de alguma coisa ou alguém. O predicado `is_prime(n)` reporta se `n` tem a propriedade de ser primo (ou não).

Exercício 8. Escreva um procedimento chamado `primes_by_trial(n)` que seja necessariamente recursivo e que necessariamente use o procedimento `itrial(n)` pra produzir uma lista de tuplas contendo a fatoração prima de `n`. Por exemplo, se `n` é 12, seu procedimento deve produzir a lista `[(2, 2), (3, 1)]`, que representa a fatoração prima $2^2 \cdot 3 = 12$.

Exercício 9. Escreva um procedimento chamado `primes(n)` que necessariamente usa `primes_by_trial(n)` (*ou qualquer procedimento extra que você tenha escrito para auxiliar sua redação de* `primes_by_trial`) que produza uma lista dos primos que dividam `n`. Por exemplo, `primes(831875000)` deve produzir a lista `[2, 5, 11]`.

Exercício 10. Escreva um procedimento chamado `mod_mul(a, b, m)` que compute a multiplicação de `a` por `b` módulo `m`.

Exercício 11. Escreva um procedimento chamado `sq(b, m)` que compute o quadrado de `b` módulo `m`.

Exercício 12. Escreva um predicado chamado `is_even(n)` que detecte se `n` é par.

Exercício 13. Escreva um predicado chamado `is_odd(n)` que necessariamente use `is_even(n)` do exercício anterior e que detecte quando `n` é ímpar.

Comentário. A palavra *even* significa "par" e *odd*, "ímpar" (#eu-já-sabia).

Exercício 14. Escreva um procedimento chamado `square_and_multiply(b, e, m)` que seja necessariamente recursivo e compute $b^e \bmod m$ usando o algoritmo chamado *square and multiply*.

Comentário. Neal Koblitz chama o *square and multiply* de *repeated squaring method* [1, capítulo I, seção 3]. Douglas Stinson o chama de *square-and-multiply* [2, capítulo 6, seção 6.3, algoritmo 6.5].

Exercício 15. Faça com que o procedimento `square_and_multiply(b, e, m)` atenda *também* pelo nome `mod_exp(b, e, m)` (de *modular exponentiation*).

Exercício 16. Escreva um predicado chamado `generator_try_d(c, d, p)` que consome um candidato c a gerador do grupo $\mathbb{Z}_p$ e um divisor d da ordem do grupo $\mathbb{Z}_p$. Escreva-o pra que ele retorne `True` se $c^{(p-1)/d} \not\equiv 1 \bmod p$ e retorne `False` caso contrário. Em outras palavras, seu predicado experimenta o divisor d e diz `True` se c ainda é um candidato a gerador; se não for, ele retorna `False` (quando desistiríamos de c).

Exercício 17. Escreva um predicado chamado `is_generator(c, p)` que pegue cada primo `d` da fatoração de $|\mathbb{Z}_p|$ e necessariamente invoque `generator_try_d(c, d, p)` pra testar a candidatura de `c` a gerador de $\mathbb{Z}_p$. Seu procedimento é um predicado. Diga `True` se `c` for gerador.

Exercício 18. Redija uma explicação do método de `is_generator(c, p)`. Sua explicação deve conter necessariamente em que teorema `is_generator(c, p)` se apoia e por quê.

Exercício 19. Escreva um procedimento chamado `generator(p)` que consuma um primo `p` e produza um gerador qualquer do grupo $\mathbb{Z}_p$.

Exercício 20. Redija um parágrafo (ou parágrafos) explicando o que seria necessário fazer em seus procedimentos (e que novos procedimentos precisariam ser escritos) pra generalizar a busca por geradores pra que os argumentos `p` não precisem mais ser números primos. Em outras palavras — o que é preciso fazer pra que possamos trabalhar com $\mathbb{Z}_n$, sendo n composto?

> To think, you have to write. To really think, you have to write. If you're thinking without writing, chances are you're fooling yourself. – Leslie Lamport, Microsoft Faculty Summit, 2014. Fonte: https://youtu.be/-4Yp3j_jk8Q?t=176s

Referências

[1] Neal Koblitz. *A course in number theory and cryptography.* 2ª ed. Graduate texts in mathematics. Springer, 1994. ISBN: 0-387-94293-9.

[2] Douglas R. Stinson. *Cryptography, Theory and Practice.* 3ª ed. Chapman e Hall, CRC, 2006. ISBN: 978-1-58488-508-5.