Universidade Federal de Minas Gerais
Instituto de Ciências Exatas – ICEx
Departamento de Matemática

Álgebra A
Avisos para os Trabalhos Práticos

Os Trabalhos Práticos que seguirão consistem em implementar as primitivas necessárias para criptografar mensagens usando o RSA "livro-texto"[1].

Objetivo e Avisos

O objetivo deste Trabalho Prático é explorar como implementar as rotinas que vimos e veremos em sala mesmo que elas não existam na sua linguagem. Assim, você só poderá usar as funções da biblioteca GMP que generalizem as operações que o C já suporta para tipos inteiros (ou seja, adição, subtração, multiplicação, divisão e módulo).

1. Use apenas as funções cujos nomes comecem com

   `mpz_{init,set,add,sub,mul,tdiv,fdiv,mod,cmp,even,odd,urandom}`

   ou `gmp_{randinit,randseed,scanf,printf}`. Em particular, as versões submetidas dos seus TPs não podem conter funções recomendadas nas seções "Como testar" (essa não é uma lista completa das funções proibidas).

2. Você deve implementar algoritmos eficientes. Isso significa que nenhuma das suas funções deve demorar muito mais que um minuto para executar, mesmo que as entradas sejam da ordem de 2^{4096}.

Dicas

1. Para compilar programas usando a biblioteca GMP, você deve adicionar a flag `-lgmp` *no final da lista de argumentos para o compilador* (exemplo: `gcc -O2 -o prog prog.c -lgmp`). A documentação do GMP fica em

   `https://gmplib.org/manual/Integer-Functions.html`

   e é muito boa. O Apêndice A tem um resumo das pegadinhas do GMP.

2. Você pode gerar entradas aleatórias e comparar a saída do seu programa com a função pré-implementada do GMP (indicada nas questões) para saber se sua solução contém erros. Lembre-se de remover as funções proibidas antes de submeter.

[1] Veja a seção Disclaimer para entender o que a expressão "livro-texto" significa nesse contexto.

3. Preferimos algoritmos recursivos em sala, então um aviso: Por padrão, a pilha no Linux é limitada. Você pode usar o comando `ulimit -s TAMANHO` para fixar um limite maior. O limite é vinculado ao *shell* em que você rodou o comando; uma vez fixado um limite, ele só pode ser aumentado pelo superusuário (mas você pode abrir outro terminal e começar de novo se não quiser usar o *root*).

   Em C, algoritmos recursivos frequentemente são mais lentos que suas versões iterativas (isso não é verdade em todas as linguagens!). Todos os algoritmos que implementaremos admitem versões iterativas, e perguntas a respeito das versões iterativas são igualmente bem-vindas.

Disclaimer

O objetivo desse curso é treinar os fundamentos matemáticos relevantes para entender a criptografia RSA, e os Trabalhos Práticos servirão para reforçar tal aprendizado matemático. Por simplicidade, nenhum cuidado será feito para evitar ataques aos algoritmos aqui implementados. Em outras palavras, estaremos implementando o que é chamado de RSA "livro-texto".

Tomar cuidado com evitar ataques é algo extremamente complicado. Estudar os possíveis ataques e como evitá-los é a carreira de muita gente. Sendo assim, a menos para fins de estudo e/ou diversão, **não invente seu próprio método criptográfico, nem faça sua própria implementação de um método criptográfico existente**. Se precisar usar criptografia em algum projeto, prefira usar uma biblioteca confiável pronta, como `https://nacl.cr.yp.to/`.

O primeiro passo para entender melhor o que pode dar de errado é estudar os ataques já conhecidos. Para a prática, um site altamente recomendado é `https://cryptopals.com/`.

A Particularidades do GMP

- Um inteiro do GMP é do tipo `mpz_t`. Antes de usar uma variável do tipo `mpz_t`, você deve inicializá-la com `mpz_init`, e liberar a memória com `mpz_clear` ao terminar de usá-la.
- Funções não podem retornar variáveis do tipo `mpz_t`. Isso ocorre porque o tipo `mpz_t` é um array. Sendo assim, as funções que "querem" retornar números recebem argumentos `mpz_t` e os preenchem com a resposta calculada; nessa lista, esses argumentos serão chamados de *argumentos de saída*.
- Nas funções do GMP, os argumentos de saída aparecem primeiro (e a lista segue essa convenção). Isso é intuitivo: Compare a ordem dos parâmetros de somar dois inteiros do C (`x = y + z`) com somar dois inteiros do GMP (`mpz_add(x, y, z)`).
- Às vezes queremos passar um inteiro do C como um dos argumentos para uma função; por exemplo, podemos querer subtrair 1 de um número grande. O GMP fornece variantes terminadas com `_ui`, cujo tipo do último argumento é um `unsigned int` do C ao invés de um `mpz_t`. Assim, o equivalente a "`x = y - 1`" é "`mpz_sub_ui(x, y, 1)`".
- Para comparar inteiros do GMP, podemos usar a função `mpz_cmp`. Ela segue a mesma convenção das funções de comparação do C (como `strcmp`): Se `OP` é um operador de comparação qualquer (como `>=` ou `==`), então "`mpz_cmp(x, y) OP 0`" é equivalente à expressão matemática "`x OP y`". Essa função também tem a variante terminada em `_ui`.
- É fácil ler e imprimir inteiros do GMP, por exemplo para fins de depuração, bastando usar as funções `gmp_scanf` e `gmp_printf`. Do mesmo jeito que "`%d`" lê/imprime um inteiro normal, "`%Zd`" lê/imprime um inteiro do GMP. Para usar tais funções, você deve incluir o cabeçalho `stdio.h` *antes* de incluir `gmp.h`.