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

Álgebra A
Trabalho Prático 1 (v1.1.1)

Esse trabalho prático consiste em implementar as primitivas necessárias para criptografar mensagens usando o RSA "livro-texto".

Objetivo e Avisos

O objetivo deste Trabalho Prático é explorar como implementar as rotinas que vimos em sala mesmo que elas não existam na sua linguagem. Você só pode 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, a versão submetida do seu TP não pode 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. A documentação do GMP fica em

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

   e é muito boa. O Apêndice B 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.

3. Vimos algoritmos recursivos em sala. 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 dessa lista 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 este Trabalho Prático serve para reforçar tal aprendizado matemático. Por simplicidade, nenhum cuidado será feito para evitar ataques aos algoritmos aqui implementados.

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/`.

1 Aritmética

Questão 1. Implemente o algoritmo estendido de Euclides: Dados números a e b, sua função deve computar números x, y e g tais que $\text{mdc}(a, b) = g$ e $ax + by = g$.

Valor de retorno Não há
Assinatura
```
void mdc_estendido(mpz_t g, mpz_t x, mpz_t y,
                   const mpz_t a, const mpz_t b)
```

	Nome	Tipo	Descrição
Entrada	`a`	`mpz_t`	
	`b`	`mpz_t`	
Saída	`g`	`mpz_t`	O maior divisor comum de a e b.
	`x`	`mpz_t`	Inteiro tal que $ax + by = g$.
	`y`	`mpz_t`	Inteiro tal que $ax + by = g$.

Como testar: Só existe um valor de g correto, que você pode conferir com a função `mpz_gcd(mpz_t g, mpz_t a, mpz_t b)`. Quanto a (x, y), existe a função `mpz_gcdext`, com mesma assinatura da função `mdc_estendido`, mas é fato que existem vários pares (x, y) válidos. Você pode facilmente verificar se a equação $ax + by = g$ é satisfeita pelos seus números.

Questão 2. Use a questão anterior para implementar uma função que calcula o inverso modular de um número. Sua função deve retornar um inteiro que indica se o número tem inverso modular; caso o número tenha inverso modular, sua função deve preencher o argumento `r` com o valor.

Valor de retorno 1 se o inverso modular existe e foi calculado,
0 se o inverso modular não existe.
Assinatura
```
int inverso_modular(mpz_t r,
                    const mpz_t a,
                    const mpz_t n)
```

	Nome	Tipo	Descrição
Entrada	`a`	`mpz_t`	
	`n`	`mpz_t`	
Saída	`r`	`mpz_t`	Um número tal que $ar \equiv 1 \pmod{n}$, caso exista.

Como testar: Você pode comparar sua resposta com a função `mpz_invert`, que tem mesma assinatura.

Questão 3. Implemente o algoritmo de exponenciação rápida visto em sala, usando que

$$b^e = \begin{cases} \left(b^{e/2}\right)^2 & \text{se } e \text{ for par} \\ b \cdot \left(b^{\lfloor e/2 \rfloor}\right)^2 & \text{se } e \text{ for ímpar.} \end{cases}$$

O número de chamadas à função `mpz_mul` deverá ser proporcional ao número de bits do argumento `e`.

Como vimos em sala, tanto a versão recursiva quanto a versão iterativa dessa função são razoavelmente simples. Se você preferir, pode implementar essa função iterativamente; isso irá acelerar sua função `descriptografa` (da Questão 11).

Valor de retorno Não há.

Assinatura
```
void exp_binaria(mpz_t r,
                 const mpz_t b,
                 const mpz_t e,
                 const mpz_t n)
```

	Nome	Tipo	Descrição
Entrada	`b`	`mpz_t`	
	`e`	`mpz_t`	
	`n`	`mpz_t`	
Saída	`r`	`mpz_t`	Um número tal que $b^e \equiv r \pmod{n}$ e $0 \leqslant r < n$.

Como testar: Você pode comparar sua resposta com a função `mpz_powm`, que tem mesma assinatura.

2 Testes de primalidade

Questão 4. Implemente uma função que verifica se um número n passa no teste de Miller com base a. Para facilitar, sua função irá receber alguns valores redundantes que simplificam as contas (ver tabela abaixo).

Valor de retorno
- 0 se o número é definitivamente composto,
- 1 se o número é primo ou pseudoprimo forte para a base a.

Assinatura

```
int talvez_primo(const mpz_t a,
                 const mpz_t n,
                 const mpz_t n1,
                 unsigned int t,
                 const mpz_t q)
```

	Nome	Tipo	Descrição
Entrada	`a`	`mpz_t`	Base do teste de primalidade
	`n`	`mpz_t`	Número cuja primalidade está sendo testada
	`n1`	`mpz_t`	Variável auxiliar: $n1 = n - 1$
	`t`	`unsigned int`	Variável auxiliar: $n - 1 = 2^t q$.
	`q`	`mpz_t`	Variável auxiliar: $n - 1 = 2^t q$.

Como testar: Além dos casos usuais, não esqueça de testar os casos $a = n$ e $n \mid a$. Ao contrário da simplificação vista em sala, a não é necessariamente menor que n.

Questão 5. Implemente a funcao `provavelmente_primo`, com o teste de primalidade de Miller–Rabin: Dado um número de iterações `iter` e um número `n`, seu programa deve, por `iter` vezes, gerar uma base aleatória `a` entre 2 e $n-1$ e usar a função `talvez_primo` para verificar a primalidade de `n`. Como mencionado em sala, a probabilidade dessa função errar é no máximo $4^{-\texttt{iter}}$.

Recomenda-se ler o Apêndice A, que fala de números aleatórios e contém a função `void numero_aleatorio(mpz_t r, const mpz_t n)`, cujo resultado é um número aleatório `r` entre 1 e `n`.

Valor de retorno 0 se o número é definitivamente composto,
1 se o número é provavelmente primo.

Assinatura
```
int provavelmente_primo(const mpz_t n,
                        unsigned int iter,
                        gmp_randstate_t rnd)
```

	Nome	Tipo	Descrição
Entrada	`n`	`mpz_t`	Número a testar primalidade.
	`iter`	`unsigned int`	Número de iterações do teste de Miller.
E/S	`rnd`	`gmp_randstate_t`	O estado do gerador aleatório.

Como testar: Você pode comparar sua resposta com o resultado da função `mpz_probab_prime_p`, que tem a mesma assinatura. A função do GMP retorna um valor não-nulo (não necessariamente 1) se o número for provavelmente primo.

Questão 6. Implemente uma função que, dado $b \geqslant 1$, retorna um primo aleatório no intervalo $[2, 2^b)$. A função `mpz_urandomb(r, rnd, b)` gera um número aleatório com até `b` bits e colocá-lo na variável `r` (ou seja, $0 \leqslant r < 2^b$). O teste de primalidade que você usar deve ter probabilidade no máximo 4^{-20} de dizer que um número é primo erroneamente.

Valor de retorno Não há.

Assinatura
```
void primo_aleatorio(mpz_t r,
                     unsigned int b,
                     gmp_randstate_t rnd)
```

	Nome	Tipo	Descrição
Entrada	`b`	`unsigned int`	
Saída	`r`	`mpz_t`	Um número primo aleatório entre 2 e 2^b.
E/S	`rnd`	`gmp_randstate_t`	O estado do gerador aleatório.

Dica: Tome cuidado para não enviesar sua escolha de primos. Um primo p tal que $p-2$ também é primo deve ser gerado com mesma probabilidade que um número p que é antecedido de muitos números compostos.

3 RSA

Nas próximas questões, você vai gerar um par de chaves e criptografar uma mensagem. Apesar de ser essencial para a segurança do RSA na vida real, não usaremos *padding* ao criptografar nossas mensagens (vide *disclaimer*).

Questão 7. Escreva uma função `gera_chaves`, que faz o seguinte:

1. Gera dois primos aleatórios p e q no intervalo $[2, 2^{2048})$. Seja $n = p \cdot q$.
2. Acha o menor $e \geqslant 65537$ tal que (n, e) é uma chave pública válida.
3. Gera d tal que (n, d) seja a chave privada correspondente à chave pública (n, e).

Valor de retorno Não há
Assinatura
```
void gera_chaves(mpz_t n, mpz_t e, mpz_t d,
                 gmp_randstate_t rnd)
```

	Nome	Tipo	Descrição
Saída	`n`	`mpz_t`	Um número da forma $n = p \cdot q$.
	`e`	`mpz_t`	Um número $e \geqslant 65537$ tal que (n, e) é uma chave pública válida.
	`d`	`mpz_t`	Um número d tal que (n, d) é a chave privada correspondente a (n, e).
E/S	`rnd`	`gmp_randstate_t`	O estado do gerador aleatório.

Usando sua função, gere um par de chaves. **Guarde sua chave privada e poste a chave pública correspondente no fórum "Chaves públicas" do Moodle até o dia 03 de julho**.

Agora, vamos escrever funções que permitem criptografar e descriptografar mensagens pequenas. Primeiro precisamos converter texto em números.

Questão 8. Implemente uma função `codifica`, que recebe um texto com até 500 caracteres e retorna um número correspondente a ver esse texto como um número em base 256.

Mais precisamente, cada um dos caracteres do texto tem um código ASCII. Se o código ASCII do i-ésimo caractere da string é s_i e a string tem n bytes, você deve retornar o número

$$\sum_{i=0}^{n-1} s_i \cdot 256^i.$$

Valor de retorno Não há
Assinatura `void codifica(mpz_t r, const char *str)`

	Nome	Tipo	Descrição
Entrada	`str`	`const char *`	O texto a ser codificado.
Saída	`r`	`mpz_t`	O resultado da fórmula acima.

Questão 9. Implemente uma função `decodifica`, que desfaz a função `codifica`, retornando um `char *`. A função deve alocar memória, e não é responsabilidade da função liberar tal memória. Você pode assumir que o resultado tem no máximo 500 caracteres.

Valor de retorno Uma string, o número n decodificado.
Assinatura `char *decodifica(const mpz_t n)`

	Nome	Tipo	Descrição
Entrada	`n`	`mpz_t`	O número a ser decodificado.

Como testar: Para qualquer string `str` de até 500 caracteres, tem que valer que `decodifica(codifica(str)) == str`.

As funções abaixo irão fazer a criptografia RSA em si.

Questão 10. Implemente uma função `criptografa`, que recebe números n, e e M e retorna C, a versão criptografada do número M.

Valor de retorno Não há
Assinatura
```
void criptografa(mpz_t C,
                 const mpz_t M,
                 const mpz_t n,
                 const mpz_t e)
```

	Nome	Tipo	Descrição
Entrada	`M`	`mpz_t`	O número a ser criptografado.
	`n`	`mpz_t`	(n, e) é a chave pública.
	`e`	`mpz_t`	
Saída	`C`	`mpz_t`	Versão criptografada de M.

Questão 11. Implemente uma função `descriptografa`, que recebe `n`, `d` e `C` e retorna M, a versão descriptografada do número C.

Valor de retorno Não há
Assinatura
```
void descriptografa(mpz_t M,
                    const mpz_t C,
                    const mpz_t n,
                    const mpz_t d)
```

	Nome	Tipo	Descrição
Entrada	`C`	`mpz_t`	O número a ser descriptografado.
	`n`	`mpz_t`	(n, d) é a chave privada.
	`d`	`mpz_t`	
Saída	`M`	`mpz_t`	Versão descriptografada de C.

Parabéns!

Você implementou tudo que o TP pede. Agora você pode aproveitar para testar seu código mandando mensagens criptografadas para seus amigos que tiverem postado as chaves públicas no fórum.

4 Detalhes sobre a correção

Logo serão adicionados detalhes sobre a correção. Para a correção, utilizarei sua chave pública (que foi postada no fórum) para criptografar uma mensagem, que deverá ser lida por você. **É importante que você guarde sua chave privada**, e é recomendado que você teste suas funções com as demais pessoas.

A Números aleatórios

No GMP, o estado do gerador de números aleatório é explícito. Você deve criar **uma única variável** do tipo `gmp_randstate_t`, inicializá-la e passá-la para todas as funções que podem precisar de números aleatórios.

O gerador de números aleatórios requer um *seed* para inicializar o processo, e irá gerar sempre os mesmos números se o *seed* for fixo. Isso parece ruim, mas é ótimo para debugar. Não esqueça de escolher seu próprio *seed* (ao invés de 12394781 no código abaixo) na hora de gerar suas chaves!

O seguinte programa lê um número n da entrada e imprime um número aleatório no intervalo $[1, n]$. Você pode usá-lo como exemplo.

```
#include <stdio.h>
#include <gmp.h>

void numero_aleatorio(mpz_t r, const mpz_t n, gmp_randstate_t rnd) {
    mp_bitcnt_t num_bits = mpz_sizeinbase(n, 2);
    do {
        mpz_urandomb(r, rnd, num_bits);
    } while (!(mpz_cmp_ui(r, 1) >= 0 && mpz_cmp(r, n) <= 0));
}

int main(int argc, char **argv) {
    gmp_randstate_t rnd;
    gmp_randinit_default(rnd);
    gmp_randseed_ui(rnd, 12394781);

    mpz_t n, aleatorio;
    mpz_init(n);
    mpz_init(aleatorio);

    gmp_scanf("%Zd", n);
    numero_aleatorio(aleatorio, n, rnd);
    gmp_printf("%Zd\n", aleatorio);

    mpz_clear(aleatorio);
    mpz_clear(n);
}
```

B 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. O Apêndice A tem um exemplo simples de uso das funções `gmp_scanf` e `gmp_printf`. Para usá-las, você deve incluir `stdio.h` *antes* de incluir `gmp.h`.