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

Álgebra A
Trabalho Prático 2: Primalidade

Questão 1. 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: $n - 1 = 2^t q$.

Aviso: Ao contrário do que foi visto em aula, não estamos garantindo a condição $1 < a < n$. Em aula, usamos isso para garantir que $n \nmid a$. No caso em que $n \mid a$, o teste é sempre inconclusivo.

Questão 2. 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 `k` 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^{-k}.

Recomenda-se ler o Apêndice A, que fala de números aleatórios. Você pode usar a função `void mpz_urandomm(mpz_t r, gmp_randstate_t rnd, const mpz_t n)`, que gera um número aleatório entre 0 e $n-1$, ou estudar a implementação da função `void numero_aleatorio(mpz_t r, gmp_randstate_t rnd, const mpz_t n)` que está no apêndice, 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 k,
                        gmp_randstate_t rnd)
```

	Nome	Tipo	Descrição
Entrada	`n`	`mpz_t`	Número a testar primalidade.
	`k`	`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 do GMP `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 3. Implemente uma função que, dado $b \geqslant 1$, retorna um primo aleatório no intervalo $[2^{b-1}, 2^b)$. 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 no intervalo $[2^{b-1}, 2^b)$.
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.

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. Para testar seu código, você deve escolher seu próprio *seed* (ao invés de 12394781 no código abaixo) e inicializar o gerador *uma única vez*, como no código abaixo. Depois de inicializá-lo, o estado do gerador será atualizado toda vez que você usá-lo (sorteando um número com as funções `mpz_urandomb` ou `mpz_urandomm`, por exemplo), o que faz com que o gerador possa emitir vários números diferentes. Um erro comum é inicializar o gerador toda vez que usá-lo: nesse caso, pedir X números aleatórios ao gerador resultará em receber o mesmo número X vezes.

Ao enviar seu programa para o Moodle, você *não deve* inicializar o gerador de números aleatórios ou passar um *seed*; ele já estará inicializado para você.

O seguinte programa lê um número n da entrada e imprime um número aleatório no intervalo $[1, n]$, usando *amostragem por rejeição*: sorteamos um número de b bits, onde b é o número de bits do número n, testamos se ele está entre 1 e n e repetimos o procedimento até que tal condição seja verdadeira. Você pode usá-lo como exemplo.

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

void numero_aleatorio(mpz_t r, gmp_randstate_t rnd, const mpz_t n) {
    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, rnd, n);
    gmp_printf("%Zd\n", aleatorio);

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