David Grunheidt Vilela Ordine - 16202253

INE5429 - 2021.2 - Trabalho Números Primos

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Ciência da Computação

Florianópolis

2021

Sumário

1 Números aleatórios

Os algoritmos escolhidos para geração de um número aleatório foram o algoritmo *Xorshift* de 32 *bits* e o algoritmo de Park–Miller, os quais foram ambos implementados em C. As referências a cada um dos algoritmos podem ser encontradas em comentários de código na seção 3. Ambos algoritmos foram aplicados em variáveis de tipo *uint32_t*, logo, foi possível gerar números aleatórios para todas as variações de bits pedidas no trabalho (40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096). A geração dos números aleatórios toma em conta o número mínimo de *uint32_t*s em que cada variação de *bits* cabe dentro. Para as variações divisíveis por 32, esse resultado será exato. Para os que não são divisíveis por 32, haverá uma sobra de *bits*, que será excluída ao final da geração do número, retornando só os *bytes* que cabem dentro de cada variação. Como todas as variações são divisíveis por 8 (1 *byte*), é possível fazer o que foi citado acima.

O algoritmo itera sobre cada 4 *bytes* de cada variação de *bits* e aplica um dos dois algoritmos de geração de números aleatórios citados acima (o mesmo durante toda iteração). No final somente os *bytes* que cabem dentro de uma certa variação de *bits* são usados, de forma que o *array* de *bytes* da iteração atual tenha exatamente a quantidade de *bits* pedida. Algumas conversões não triviais de tipos também são feitas durante esse processo. No final, é medido o tempo levado para a geração do número aleatório. A saída da execução do programa, a qual mostra todos os resultados pedidos no trabalho, também pode ser vista na seção 4. O programa foi executado em um *notebook MacBook Air* (M1, 2020). A tabela abaixo mostra os tempos levados para a geração de cada número aleatório. Os números primos gerados serão mostrados na seção 2.

n.º bits	**[Xorshift] Tempo p/ gerar(us)**	**[Park-Miller] Tempo p/ gerar(us)**
40	1	< 1
56	< 1	< 1
80	1	< 1
128	~1	~1
168	1	~1
224	~1	~1
256	~1	~1
512	~1	~1
1024	~3	~3
2048	~6	~5
4096	~11	~12

2 Números primos

Para a verificação de primalidade dos números aleatórios gerados, foram escolhidos os algoritmos de Miller-Rabin e de Fermat. Novamente, as referências a cada um dos algoritmos podem ser encontradas em comentários de código na seção 3.

A principal dificuldade encontrada foi em relação a operações aritméticas com o *array de bytes* contendo o número aleatório de N bits. Por isso, foi usado a biblioteca de aritmética de precisão múltipla da GNU (libgmp) para lidar com as operações aritméticas sobre os *BigInt* gerados.

Primeiramente executa-se o algoritmo de *Fermat*. Caso ele retorne que o número é primo, é feita uma segunda verificação com o algoritmo de *Miller-Rabin*. Um novo número aleatório para cada variação de *bits* é gerado em *loop* e testado com esses 2 algoritmos, até que ambos retornem que o número é primo. Quando um número primo é encontrado, o algoritmo avança pra próxima variação de *bits*.

O algoritmo de *Fermat* foi escolhido por sua similaridade ao de *Miller-Rabin*. Em todos os testes realizados, o tempo de sua execução foi aproximadamente o dobro do tempo de execução do teste de *Miller-Rabin*. As tabelas abaixo mostram os números aleatórios primos gerados (**NPG**) via *Xorshift* e *Parkmiller* e os tempos de verificação de primalidade de cada um dos algoritmos (**T1 para *Miller-Rabin* e T2 para *Fermat***), assim como o tempo total de execução para achar aquele número (**T3**).

Tabela 1 – Números primos gerados via Xorshift

n.º bits	T1 (us)	T2 (us)	T3 (ms)	Número primo gerado
40	5	7	40.94	Veja em 2.0.1.1
56	4	5	6	Veja em 2.0.1.2
80	10	19	27.99	Veja em 2.0.1.3
128	18	35	271	Veja em 2.0.1.4
168	29	57	176.99	Veja em 2.0.1.5
224	47	89	1258.99	Veja em 2.0.1.6
256	55	112	345.98	Veja em 2.0.1.7
512	250	481	192.02	Veja em 2.0.1.8
1024	1375	2732	30364.01	Veja em 2.0.1.9
2048	8988	17834	26446	Veja em 2.0.1.10
4096	65451	130939	75128	Veja em 2.0.1.11

Tabela 2 – Números primos gerados via ParkMiller

n.º bits	T1 (us)	T2 (us)	T3 (ms)	Número primo gerado
40	6	8	4.86	Veja em 2.0.2.1
56	3	5	49.98	Veja em 2.0.2.2
80	11	20	10.02	Veja em 2.0.2.3
128	17	30	200	Veja em 2.0.2.4
168	29	60	50.03	Veja em 2.0.2.5
224	50	90	1480.05	Veja em 2.0.2.6
256	52	106	2067.01	Veja em 2.0.2.7
512	228	455	4477.5	Veja em 2.0.2.8
1024	1361	2701	6060.32	Veja em 2.0.2.9
2048	8773	17555	3544.2	Veja em 2.0.2.10
4096	65916	131337	103688.39	Veja em 2.0.2.11

2.0.1 Números primos gerados via Xorshift

Aqui estão todos os números primos gerados via cada um dos algoritmos de geração de números aleatórios.

2.0.1.1 40 Bits

1080897758459

2.0.1.2 56 Bits

6740289462465089

2.0.1.3 80 Bits

94145264245105812837359

2.0.1.4 128 Bits

280511325997986961258318950099540774287

2.0.1.5 168 Bits

312502748493994987444052344015429162512258721637333

2.0.1.6 224 Bits

1016431035184878683181347597089249422507654018290654143256550152417

2.0.1.7 256 Bits

68313270081264285813291851584955757173364545152715491141009321448687694
914221

2.0.1.8 512 Bits

67486992288949865384258841853499815392819096550883972283362053925549535
08481921082023149347717037231628427107525419393597260332039948874046308061264
377217

2.0.1.9 1024

91105619047657996629149291475279660477669921230167486942446408347445317
15963491137917525320381035555423653875900403945015790140699521395469102402552
07169255338743310721858017252922602693500911961488190584885960977887175396058
20027961996338235648689636358398329993735507019832805457634676317086034478477
053923

2.0.1.10 2048 Bits

24505523783130559650854822679107813106614816875254751525052193561095442
44731430638861280102357500213492923581492617819911812957143971770116240231271
73437711451956514590645901198100432334332096124195139664014650313267850645343
71346599832433956988410077506223704984287780320317141415242272061783029489307
87879308657314606036378647553386993884079725272009793137391898437764821915420
82992025547637513310690032088316091641135810042641820769750274798333316980040
39031010541457879127174212894645203828691334976470070511481539891715121379130
17906049615041414854192052347854027947623810988336395787060676305509332410333
6595757

2.0.1.11 4096

78957695762336614411420361921532780149829097239673866858708107326064836
11528746364266388211007738138909377260138140787065856253332470028352156278018
71115074116247635744925493502653667600613750032793127084546362293933242069225
70933375030942472074116956174240104781436159895591673616593275688290189585604
90217463974293393568904330390370374741148391777941319891455597891399278732046
15644239150948497876119559265081821455995418978453111216720886444737930099689
40395787330426192590902312294374639169293815219759850986856265484456258673604
00601720555282094391838458028631077090462084809748336917329247753776572836625
58522938276887723705304563586525715539873973107543245327306778303017446631961

9433604860794305515065487817482115543343959593406207658272846572726715026361387708279042653371676044876957053942284675147436874157883633340360684325520421801064789717060530161294010222191222995272445960194959114760170648990495643131598058087338787791417232972234198430054821864524282777537560256832128544777658507550253947307253447774439872036580632716113383625391107091384033854305999688757705879234416023369444933610461993200229035353191122077429165404859704463025526725312589089833263664909225232083467672913063004447243332232597615839920638
7

2.0.2 Números primos gerados via ParkMiller

2.0.2.1 40 Bits

410410230623

2.0.2.2 56 Bits

5119639974719561

2.0.2.3 80 Bits

422564292559709587003771

2.0.2.4 128 Bits

42975904470358790365139909623512025307

2.0.2.5 168 Bits

157318367992577513796632311044363761562601471842667

2.0.2.6 224 Bits

11921549956851743882258417174599201275172457601682488715406811044 93

2.0.2.7 256 Bits

52248040541754896294375225716928004260342884395974106404281586662609414203511

2.0.2.8 512 Bits

9276814289831341563209253914875412874174974236145141341715550914217080373408118820063355470592430022608872625730112914439033762066040641830314737995 30717

2.0.2.9 1024

7403351375549588368203811110169543945685409263666838043352889610029323262199174740918244826640378930139730426915462797915963479415014073882215050185300199005673200093087234772423290678535668270279437041695263380661430827079274607130969096676841924564294140654499636340105029776327953301983414838559118830 9863

2.0.2.10 2048 Bits

84645275968462855939730830206355139472557471386245114386177318150370057664166202044017554797710957572969531065968479440468254498723271652191139218294668893045556407639421275901217627851349347394135060515300997817023149865851907540346770619955195837327508647474159133824613104239445212374353242356490799398927721859545294684495231794182044778967020956617657603007557355399920100700854401143063711081199607804964731359160512665407025382237184136615342193785729768630203068240444918775488034910080213522898633285839094160269106636693816301466356817411617439068691077992956675436387487001567502190555927322779572018968 17

2.0.2.11 4096

2375830996232439010094819441081634804573726742809081363102294897850279055480799485523807707981437540322111076352661599596616028139233506208542755410514920534423653739348809336054662136869931244687254938893445323920231026816893041016753194816740814268657080270556364595905619092063348430196578334016183846982823209636188419953790109376846132983198027674864601701945207653754814951564570419118282139768253732757791441719956188602415764295181019082288574851234984051511813890944875666377529434536327647072468105968277496435029980144323078403819371575374824130787597248158301269383446669986994013910448910423851458664466017007312043849402727147872711751417515943718942079573936165518039094430145905229164412862812373567094396709546423473149878585260353711804101694231949434846162798030323575932620053396916880957169647736133335339877138310201936192762433165560448618461810955748726536380211283831371886431890265810499397553647476982386368671939528834680136109864586546355023623634183050638189221515234351199035126827629405704307466940576165400348680559905498885239036914108201839822247119729693594606747405558079166673164538278604747840129587171321801817239462331660670862818373533372718060922271958825092851592332819710041154345769 34907

3 Código

O código também está disponível neste repositório do GitHub

```c
#include <stdio.h>
#include <stdint.h>
#include <sys/time.h>
#include <time.h>
#include <errno.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <stdlib.h>
#include <gmp.h>
#include <unistd.h>

/*
 * Xorshift 128 bits.
 * Taken from: https://en.wikipedia.org/wiki/Xorshift#Example_implementation
 */

typedef struct xorshift32_state {
  uint32_t value;
} xorshift32_state;

/* The state word must be initialized to non-zero */
uint32_t xorshift32(xorshift32_state *state) {
    /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
    uint32_t x = state->value;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return state->value = x;
}

/*
 * Park-Miller.
 * Taken from: https://en.wikipedia.org/wiki/Xorshift#Example_implementation
 */

 uint32_t lcg_parkmiller(uint32_t *state) {
    uint64_t product = (uint64_t)*state * 48271;
    uint32_t x = (product & 0x7fffffff) + (product >> 31);

    x = (x & 0x7fffffff) + (x >> 31);
```

```c
    return *state = x;
}

/*
 * Miller Rabin
 * Taken from https://www.sanfoundry.com/c-program-implement-rabin-miller-primality-test-check-number-prime/
 */

int miller_rabin_test(mpz_t p, int iterations) {
    mpz_t two;
    mpz_init_set_ui(two, 2);

    // Macro: int mpz_cmp_ui (const mpz_t op1, unsigned long int op2)
    // Compare op1 and op2. Return a positive value if op1 > op2, zero if op1 = op2, or a negative value if op1 < op2.
    if (mpz_cmp_ui(p, 2) < 0) return 0;

    mpz_t p_mod_2;
    mpz_init(p_mod_2);
    mpz_mod(p_mod_2, p, two);

    if (mpz_cmp_ui(p, 2) != 0 && mpz_cmp_ui(p_mod_2, 0) == 0) return 0;

    mpz_t y;
    mpz_init(y);
    mpz_sub_ui(y, p, 1);

    mpz_t y_mod_2;
    mpz_init(y_mod_2);

    // Divide y by 2 and assign the result to y only if y % 2 == 0
    while(1) {
        // Function: unsigned long int mpz_fdiv_r_ui (mpz_t r, const mpz_t n, unsigned long int d)
        mpz_fdiv_r_ui(y_mod_2, y, 2);

        if (mpz_cmp_ui(y_mod_2, 0) == 0) {
            mpz_fdiv_qr_ui(y, y_mod_2, y, 2);
        } else {
            break;
        }
    }

    // z = p - 1;
    mpz_t z;
    mpz_init(z);
```

```
    mpz_sub_ui(z, p, 1);

    for (int i = 0; i < iterations; i++) {
        mpz_t rand_num;
        mpz_init_set_ui(rand_num, rand());

        mpz_t a;
        mpz_init(a);

        // Function: void mpz_mod (mpz_t r, const mpz_t n, const mpz_t d)
        mpz_mod(a, rand_num, z);

        // Function: void mpz_add_ui (mpz_t rop, const mpz_t op1, unsigned long int
    op2)
        // Set rop to op1 + op2.
        mpz_add_ui(a, a, 1);

        mpz_t temp;
        mpz_init_set(temp, y);

        mpz_t mod;
        mpz_init(mod);

        // Function: void mpz_powm (mpz_t rop, const mpz_t base, const mpz_t exp,
    const mpz_t mod)
        // Set rop to (base raised to exp) modulo mod.
        mpz_powm(mod, a, temp, p);

        while (mpz_cmp(temp, z) != 0 && mpz_cmp_ui(mod, 1) != 0 && mpz_cmp(mod, z) !=
    0) {
            // Function: void mpz_mul (mpz_t rop, const mpz_t op1, const mpz_t op2)
            // Set rop to op1 times op2.
            mpz_mul(mod, mod, mod);
            mpz_mod(mod, mod, p);

            // Function: void mpz_mul_ui (mpz_t rop, const mpz_t op1, unsigned long
    int op2)
            // Set rop to op1 times op2.
            mpz_mul_ui(temp, temp, 2);
        }

        mpz_t temp_mod_2;
        mpz_init(temp_mod_2);
        mpz_mod(temp_mod_2, temp, two);

        if (mpz_cmp(mod, z) != 0 && mpz_cmp_ui(temp_mod_2, 0) == 0) {
            return 0;
```

```
        }
    }

    return 1;
}

/*
 * Fermat's test for checking primality
 */
int fermat_test(mpz_t p, int iterations) {
    // Macro: int mpz_cmp_ui (const mpz_t op1, unsigned long int op2)
    // Compare op1 and op2. Return a positive value if op1 > op2, zero if op1 = op2, or a negative value if op1 < op2.
    if (mpz_cmp_ui(p, 1) == 0) return 0;

    // z = p - 1;
    mpz_t z;
    mpz_init(z);
    mpz_sub_ui(z, p, 1);

    for (int i = 0; i < iterations; i++) {
        mpz_t rand_num;
        mpz_init_set_ui(rand_num, rand());

        mpz_t a;
        mpz_init(a);

        // Function: void mpz_mod (mpz_t r, const mpz_t n, const mpz_t d)
        mpz_mod(a, rand_num, z);

        // Function: void mpz_add_ui (mpz_t rop, const mpz_t op1, unsigned long int op2)
        // Set rop to op1 + op2.
        mpz_add_ui(a, a, 1);

        mpz_t mod;
        mpz_init(mod);

        // Function: void mpz_powm (mpz_t rop, const mpz_t base, const mpz_t exp, const mpz_t mod)
        // Set rop to (base raised to exp) modulo mod.
        mpz_powm(mod, a, z, p);

        if (mpz_cmp_ui(mod, 1) != 0) return 0;
    }

    return 1;
```

```c
}

/*
 * -------- MAIN ----------
 */

#define N_RAND_NUMBERS 11 // Generate numbers with 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096 bits.
#define UINT32_T_SIZE_IN_BITS 32 // Minimum of bits to generate a random number
#define BYTE_SIZE_IN_BITS 8 // Each position of a byte array
#define BYTES_IN_UINT32_T 4 // Number of bytes in uint32_t type

static const int n_bits[N_RAND_NUMBERS] = {40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096};

// This is a random algorithm to generate a seed. Created by myself.
uint32_t generate_seed() {
    struct timeval first_tv, second_tv, third_tv;

    gettimeofday(&first_tv, NULL);
    const unsigned long long first_seed = (unsigned long long)(first_tv.tv_sec) * 1e3 + (unsigned long long)(first_tv.tv_usec) / 1e3; // millisecondsSinceEpoch

    gettimeofday(&second_tv, NULL);
    const unsigned long long second_seed = (unsigned long long)(second_tv.tv_sec) * 1e3 + (unsigned long long)(second_tv.tv_usec) / 1e3; // millisecondsSinceEpoch

    gettimeofday(&third_tv, NULL);
    const unsigned long long third_seed = (unsigned long long)(third_tv.tv_sec) * 1e3 + (unsigned long long)(third_tv.tv_usec) / 1e3; // millisecondsSinceEpoch

    return ((first_seed * second_seed) & third_seed) % UINT32_MAX;
}

void generate_rand(unsigned char *rand_bytes, int n_uint32_t, int n_chars, int generation_step, int is_xorshift) {
    // Generate each random 32 bits in a loop.
    for (int uint32_t_index = 0; uint32_t_index < n_uint32_t; uint32_t_index++) {
        xorshift32_state state;
        state.value = generate_seed();

        const uint32_t rand = is_xorshift > 0 ? xorshift32(&state) : lcg_parkmiller(&state.value);

        const unsigned char bytes[BYTES_IN_UINT32_T] = { (rand >> 24) & 0xFF, (rand >> 16) & 0xFF, (rand >> 8) & 0xFF, rand & 0xFF };
```

```c
        const uint32_t temp_rand = (uint32_t) bytes[0] << 24 | (uint32_t) bytes[1] << 16 | (uint32_t) bytes[2] << 8 | (uint32_t) bytes[3];

        // Check if conversion from uint32_t to byte array and back worked.
        if (rand != temp_rand) {
            printf("[Error] Wrong conversion (Xorshift: %d) | uint32_t_index: %d | generation_step: %d | Expected: %lu | Got: %lu", is_xorshift, uint32_t_index, generation_step, (unsigned long) rand, (unsigned long) temp_rand);
            return exit(0);
        }

        // Make sure that it will have the numbers of bytes we want (defined by n_chars), since n_uint32_t can have more bytes than n_chars.
        const int offset = uint32_t_index * BYTES_IN_UINT32_T;
        const int size = offset + BYTES_IN_UINT32_T > n_chars ? n_chars - offset : BYTES_IN_UINT32_T;
        memcpy(&rand_bytes[offset], bytes, size * sizeof(unsigned char));
    }
}


void generate_random_number(char *rand_number, int generation_step, int is_xorshift, double* time_taken) {
    const int result_bits_size = n_bits[generation_step];
    int bits_size = result_bits_size;

    // Set "bits_size" to the closest number to "result_bits_size" that is greather than "result_bits_size" and divisible by 32.
    if (bits_size % UINT32_T_SIZE_IN_BITS != 0) {
        bits_size = UINT32_T_SIZE_IN_BITS;
        while (bits_size < result_bits_size) {
            bits_size += UINT32_T_SIZE_IN_BITS;
        }
    }

    const int n_uint32_t = bits_size / UINT32_T_SIZE_IN_BITS;
    const int n_chars = result_bits_size / BYTE_SIZE_IN_BITS;

    unsigned char rand_bytes[n_chars];

    // Measure how much time it took to generate each number;
    struct timeval start, end, result;

    gettimeofday(&start, NULL);
    generate_rand(rand_bytes, n_uint32_t, n_chars, generation_step, is_xorshift);
    gettimeofday(&end, NULL);

```

```
    timersub(&end, &start, &result);

    *time_taken = (result.tv_sec * 1e6) + (result.tv_usec);

    // Converting unsigned char* to char*
    // Taken from https://stackoverflow.com/a/22260250/16808367O)
    for(int index = 0; index < n_chars; index++) {
        sprintf(rand_number + (index * 2), "%02x", rand_bytes[index]);
    }
    rand_number[n_chars * 2] = '\0';
}

int check_primality(mpz_t rand_num, double* time_taken, int is_miller_rabin) {
    // Measure how much time it took to check each number;
    struct timeval start, end, result;

    gettimeofday(&start, NULL);
    const int is_prime = is_miller_rabin > 0 ? miller_rabin_test(rand_num, 5) : fermat_test(rand_num, 10);
    gettimeofday(&end, NULL);

    timersub(&end, &start, &result);

    *time_taken = (result.tv_sec * 1e6) + (result.tv_usec);

    return is_prime;
}

int main(int argc, char **argv) {
    printf("\n---- STARTING ----\n\n");

    for (int step = 0; step < N_RAND_NUMBERS; step++) {
        const int n_bits_step = n_bits[step];
        const int n_chars = n_bits_step / BYTE_SIZE_IN_BITS;

        printf("-------------- %d BITS --------------\n\n", n_bits_step);

        // rand_algorithm = 0 -> Parkmiller || rand_algorithm = 1 -> Xorshift
        for (int rand_algorithm = 0; rand_algorithm < 2; rand_algorithm++) {
            int rand_is_prime = 0;

            double rand_generation_time = 0;
            double miller_rabin_time = 0;
            double fermat_time = 0;
            double elapsed_time = 0;

            char rand_number[(2 * n_chars) + 1];
```

```c

        mpz_t rand_num;
        mpz_init(rand_num);

        // Measure the number of iterations to find a rand prime number and the elapsed time (ms).
        int iterations = 0;
        struct timeval start, end, result;

        gettimeofday(&start, NULL);

        while (rand_is_prime < 1) {
            iterations++;
            if (iterations == INT_MAX - 2) iterations = 0;

            // printf("\rRunning iteration: %d | Max: %d | Until Max: %d", iterations, INT_MAX, INT_MAX - iterations);
            // fflush(stdout);

            // printf("\r");
            // fflush(stdout);

            // Generate a random number and copy it to rand_number byte array
            generate_random_number(rand_number, step, rand_algorithm, &rand_generation_time);

            // Use libgmp mpz_t type for big integers
            mpz_set_str(rand_num, rand_number, 16);

            // Run primality test for both algorithms, fermat first and then miller-rabin.
            rand_is_prime = check_primality(rand_num, &miller_rabin_time, 1);
            if (rand_is_prime > 0) {
                rand_is_prime = check_primality(rand_num, &fermat_time, 0);
            }
        }

        gettimeofday(&end, NULL);
        timersub(&end, &start, &result);

        elapsed_time = ((result.tv_sec * 1e6) + (result.tv_usec)) / 1e3;

        const char *algorithm = rand_algorithm > 0 ? "Xorshift" : "Parkmiller";
        printf("[Found Prime] Time Taken(ms): %.2f || Iterations: %d\n\n", elapsed_time, iterations);
        printf("[%s] Random Number Generation Time Taken(us): %.2f\n", algorithm, rand_generation_time);
```

```
            printf("[Miller Rabin] Primality Check Time Taken(us): %.2f\n",
    miller_rabin_time);
            printf("[Fermat] Primality Check Time Taken(us): %.2f\n", fermat_time);
            gmp_printf("[%s] Rand Prime Number: %Zd\n\n", algorithm, rand_num);
            fflush(stdout);
        }
    }
}
```

4 Saída do código

```

---- STARTING ----

-------------- 40 BITS --------------

[Found Prime] Time Taken(ms): 4.86 || Iterations: 2080

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 6.00
[Fermat] Primality Check Time Taken(us): 8.00
[Parkmiller] Rand Prime Number: 410410230623

[Found Prime] Time Taken(ms): 40.94 || Iterations: 27598

[Xorshift] Random Number Generation Time Taken(us): 0.00
[Miller Rabin] Primality Check Time Taken(us): 5.00
[Fermat] Primality Check Time Taken(us): 7.00
[Xorshift] Rand Prime Number: 1080897758459

-------------- 56 BITS --------------

[Found Prime] Time Taken(ms): 49.98 || Iterations: 50620

[Parkmiller] Random Number Generation Time Taken(us): 0.00
[Miller Rabin] Primality Check Time Taken(us): 3.00
[Fermat] Primality Check Time Taken(us): 5.00
[Parkmiller] Rand Prime Number: 5119639974719561

[Found Prime] Time Taken(ms): 6.00 || Iterations: 6053

[Xorshift] Random Number Generation Time Taken(us): 0.00
[Miller Rabin] Primality Check Time Taken(us): 4.00
[Fermat] Primality Check Time Taken(us): 5.00
[Xorshift] Rand Prime Number: 6740289462465089

```

```
-------------- 80 BITS --------------

[Found Prime] Time Taken(ms): 10.02 || Iterations: 7716

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 11.00
[Fermat] Primality Check Time Taken(us): 20.00
[Parkmiller] Rand Prime Number: 422564292559709587003771

[Found Prime] Time Taken(ms): 27.99 || Iterations: 22513

[Xorshift] Random Number Generation Time Taken(us): 0.00
[Miller Rabin] Primality Check Time Taken(us): 10.00
[Fermat] Primality Check Time Taken(us): 19.00
[Xorshift] Rand Prime Number: 94145264245105812837359

-------------- 128 BITS --------------

[Found Prime] Time Taken(ms): 200.00 || Iterations: 120332

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 17.00
[Fermat] Primality Check Time Taken(us): 30.00
[Parkmiller] Rand Prime Number: 42975904470358790365139909623512025307

[Found Prime] Time Taken(ms): 271.00 || Iterations: 152935

[Xorshift] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 18.00
[Fermat] Primality Check Time Taken(us): 35.00
[Xorshift] Rand Prime Number: 280511325997986961258318950099540774287

-------------- 168 BITS --------------

[Found Prime] Time Taken(ms): 50.03 || Iterations: 19785

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 29.00
[Fermat] Primality Check Time Taken(us): 60.00
```

```
[Parkmiller] Rand Prime Number:
    1573183679925775137966323110443637615626014718426 67

[Found Prime] Time Taken(ms): 176.99 || Iterations: 71718

[Xorshift] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 29.00
[Fermat] Primality Check Time Taken(us): 57.00
[Xorshift] Rand Prime Number:
    31250274849399498744405234401542916251225872163 7333

-------------- 224 BITS --------------

[Found Prime] Time Taken(ms): 1480.05 || Iterations: 465508

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 50.00
[Fermat] Primality Check Time Taken(us): 90.00
[Parkmiller] Rand Prime Number:
    11921549956851743882258417174599201275172457601682488715406811 04493

[Found Prime] Time Taken(ms): 1258.99 || Iterations: 392306

[Xorshift] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 47.00
[Fermat] Primality Check Time Taken(us): 89.00
[Xorshift] Rand Prime Number:
    10164310351848786831813475970892494225076540182906541432565501 52417

-------------- 256 BITS --------------

[Found Prime] Time Taken(ms): 2067.01 || Iterations: 568577

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 52.00
[Fermat] Primality Check Time Taken(us): 106.00
[Parkmiller] Rand Prime Number: 5224804054175489629437522571692800426034 28
    843959741064042815866626094142035 11

```

```
[Found Prime] Time Taken(ms): 345.98 || Iterations: 104178

[Xorshift] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 55.00
[Fermat] Primality Check Time Taken(us): 112.00
[Xorshift] Rand Prime Number: 6831327008126428581329185158495575717336454515271549114100932144868769491422 1

-------------- 512 BITS --------------

[Found Prime] Time Taken(ms): 4477.50 || Iterations: 592904

[Parkmiller] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 228.00
[Fermat] Primality Check Time Taken(us): 455.00
[Parkmiller] Rand Prime Number: 9276814289831341563209253914875412874174974
    2361451413417155509142170803734081188200633554705924300226088726257301129 1
    44390337620660406418303147379953071 7

[Found Prime] Time Taken(ms): 197.02 || Iterations: 24997

[Xorshift] Random Number Generation Time Taken(us): 1.00
[Miller Rabin] Primality Check Time Taken(us): 250.00
[Fermat] Primality Check Time Taken(us): 481.00
[Xorshift] Rand Prime Number: 6748699228894986538425884185349981539281909655088397228336205392554953508481921082023149347717037231628427107525419393 5
    97260332039948874046308061264377217

-------------- 1024 BITS --------------

[Found Prime] Time Taken(ms): 6060.32 || Iterations: 380702

[Parkmiller] Random Number Generation Time Taken(us): 3.00
[Miller Rabin] Primality Check Time Taken(us): 1361.00
[Fermat] Primality Check Time Taken(us): 2701.00
[Parkmiller] Rand Prime Number: 7403351375549588368203811110169543945685409
    263666838043352889610029323262199174740918244826640378930139730426915462 79
    791596347941501407388221505018530019900567320009308723477242329067853566 82
    702794370416952633806614308270792746071309690966768419245642941406544996 36
```

```
3401050297763279533019834148385591188309863

[Found Prime] Time Taken(ms): 30364.01 || Iterations: 1877540

[Xorshift] Random Number Generation Time Taken(us): 3.00
[Miller Rabin] Primality Check Time Taken(us): 1375.00
[Fermat] Primality Check Time Taken(us): 2732.00
[Xorshift] Rand Prime Number: 9110561904765799662914929147527966047766992123
    01674869424464083474453171596349113791752532038103555542365387590040394501
    57901406995213954691024025520716925533874331072185801725292260269350091196
    14881905848859609778871753960582002796199633823564868963635839832999373550
    7019832805457634676317086034478477053923

-------------- 2048 BITS --------------

[Found Prime] Time Taken(ms): 3544.20 || Iterations: 64124

[Parkmiller] Random Number Generation Time Taken(us): 6.00
[Miller Rabin] Primality Check Time Taken(us): 8773.00
[Fermat] Primality Check Time Taken(us): 17555.00
[Parkmiller] Rand Prime Number: 846452759684628559397308302063551394725574713
    86245114386177318150370057664166202044017554797710957572969531065968479440
    46825449872327165219113921829466889304555640763942127590121762785134934739
    41350605153009978170231498658519075403467706199551958373275086474741591338
    24613104239445212374353242356490799398927721859545294684495231794182044778
    96702095661765760300755735539992010070085440114306371108119960780496473135
    91605126654070253822371841366153421937857297686302030682404449187754880349
    10080213522898633285839094160269106636693816301466356817411617439068691077
    99295667543638748700156750219055592732277957201896817

[Found Prime] Time Taken(ms): 26446.46 || Iterations: 490418

[Xorshift] Random Number Generation Time Taken(us): 5.00
[Miller Rabin] Primality Check Time Taken(us): 8988.00
[Fermat] Primality Check Time Taken(us): 17834.00
[Xorshift] Rand Prime Number: 245055237831305596508548226791078131066148168
    75254751525052193561095442447314306388612801023575002134929235814926178199
    11812957143971770116240231271734377114519565145906459011981004323343320961
    24195139664014650313267850645343713465998324339569884100775062237049842877
```

```
    80320317141415242272061783029489307878793086573146060363786475533869938840
    79725272009793137391898437764821915420829920255476375133106900320883160916
    41135810042641820769750274798333316980040390310105414578791271742128946452
    03828691334976470070511481539891715121379130179060496150414148541920523478
    540279476238109883363957870606763055093324103336595757

-------------- 4096 BITS --------------

[Found Prime] Time Taken(ms): 103688.39 || Iterations: 212447

[Parkmiller] Random Number Generation Time Taken(us): 11.00
[Miller Rabin] Primality Check Time Taken(us): 65916.00
[Fermat] Primality Check Time Taken(us): 131337.00
[Parkmiller] Rand Prime Number: 2375830996232439010094819441081634804573726
    74280908136310229489785027905548079948552380770798143754032211107635266159
    95966160281392335062085427554105149205344236537393488093360546621368699312
    44687254938893445323920231026816893041016753194816740814268657080270556364
    59590561909206334843019657833401618384698282320963618841995379010937684613
    29831980276748646017019452076537548149515645704191182821397682537327577914
    41719956188602415764295181019082288574851234984051511813890944875666377529
    43453632764707246810596827749643502998014432307840381937157537482413078759
    72481583012693834466699869940139104489104238514586644660170073120438494027
    27147872711751417515943718942079573936165518039094430145905229164412862812
    37356709439670954642347314987858526035371180410169423194943484616279803032
    35759326200533969168809571696477361333353398771383102019361927624331655604
    48618461810955748726536380211283831371886431890265810499397553647476982386
    36867193952883468013610986458654635502362363418305063818922151523435119903
    51268276294057043074669405761654003486805599054988852390369141082018398222
    47119729693594606747405558079166673164538278604747840129587171321801817239
    46233166067086281837353337271806092227195882509285159233281971004115434576
    934907

[Found Prime] Time Taken(ms): 75128.10 || Iterations: 145677

[Xorshift] Random Number Generation Time Taken(us): 12.00
[Miller Rabin] Primality Check Time Taken(us): 65451.00
[Fermat] Primality Check Time Taken(us): 130939.00
[Xorshift] Rand Prime Number: 7895769576233661441142036192153278014982909972
    39673866858708107326064836115287463642663882110077381389093772601381407870
```

6585625333247002835215627801871115074116247635744925493502653667600613750032793127084546362293933242069225709333750309424720741169561742401047814361598955916736165932756882901895856049021746397429339356890433039037037474114839177794131989145559789139927873204615644239150948497876119559265081821455995418978453111216720886444737930099689403957873304261925909023122943746391692938152197598509868562654844562586736040060172055528209439183845802863107709046208480974833691732924775377657283662558522938276887723705304563586525715539873973107543245327306778303017446631961943360486079430551506548781748211554334395959340620765827284657272671502636138770827904265337167604487695705394228467514743687415788363334036068432552042180106478971706053016129401022219122299527244596019495911476017064899049564313159805808733878779141723297223419843005482186452428277753756025683212854477765850755025394730725344777443987203658063271611338362539110709138403385430599968875770587923441602336944493361046199320022903535319112207742916540485970446302552672531258908983326366490922523208346767291306300444724333223259761583992063870