Leonardo Schlüter Leite

SEGURANÇA EM COMPUTAÇÃO: TRABALHO INDIVIDUAL 2

Florianópolis

2019

SUMÁRIO

1 GERADORES DE NÚMEROS PSEUDOALEATÓRIOS

Neste capítulo serão apresentados os dois algoritmos implementados para a geração de números pseudoaleatórios. Ambos os algoritmos selecionados produzem números múltiplos de 32, portanto números com 56 bits, 168, 224 e etc. não serão gerados.

1.1 GERADOR CONGRUENTE LINEAR

Abaixo temos o código utilizado para gerar números aleatório usando LGC, baseando-se no algoritmo mostrado em https://aaronschlegel.me/linear-congruential-generator-r.html

```
public class LinearCongruential {

    long modulus;
    long a; // multiplicador
    long c; // incremento
    long seed;

    // inicializacao do gerador
    // Para que o periodo da sequencia seja do
       tamanho
    // do modulo eh necessario que o deslocamento
       ou
    // seja relativamente primo com o modulo
    // que a-1 seja divisivel por todos
    // os fatores primos de m
    public LinearCongruential(long modulus, long a,
      long c, long seed){

        this.modulus = modulus;
        this.a = a;
        this.c = c; // se c for = 0, caimos em
            outro
                //tipo de gerador.
        this.seed = seed;
    }

    public long generate(){
```

```
        this.seed = (a*this.seed +c) % modulus;
        // multiplica a semente, soma com constante
        // e faz o modulo.
        // a seed eh atualizada e entao retornada.
        return seed;
    }
}
```

Tendo este código implementado, podemos percerber que os números gerados são de 64bit. Para gerar números que sejam múltiplos de 64, como 32, 128, 1024 e 4096 foi construído um loop que a cada iteração cria uma nova semente, inicializa um objeto do tipo LinearConruential e pede o próximo número. Então esses números são concatenados até termos a quantidade de bits necessária. Na tabela 1 temos exemplos de números de vários tamanhos gerados utilizando o algoritmo LGC.

Tabela 1: Exemplo de números gerados com LGC.

Tamanho em bits	Número Gerado
31	1982274587
63	7647551033894467049
511	448733919789616510395911804421880027744398060922530070164471134528228906460616080851038371269032048288673820170285582936925941589749859507139475959842961383571268110914608281662202010389543914966884215215093924643653085181491982274587
1023	8122377077377152333927489494126459097921584156200002770059826252922922678883260357074016455741630890410891662500833631101437142680354748697783158066515804092307503207788008428240372656456711313615119209623855698128160229770033050315635554383573681109146082816620220103895439149668842152150939246436530851814919822745871982274587

2047	133875388389516398504794679886249058273696 3550505146432481406994129611692239266191110381182 9788109415713816627857207083012074316855387160589 8706811036192399219716020261922133532670520158054 4359048522073952207009051872071128440534844287149 0748686386866224474579717907067239357655113674851 2839884696560513677369673605173781255925654694451 6383060636108090887768503211080391927733196451434 3834228396222280458809625433748472226619983851991 3862510479680144492499378614729667166761507437305 4571115905226657988969455696562940111326545485325 5386955561234927144984046907733764712284919250 02 6987657231838100982214283648766262412

1.2 MULTIPLY-WITH-CARRY

Abaixo temos o código utilizado para gerar números aleatório usando MWC, baseando-se no algoritmo mostrado em https://www.javamex.com/tutorials/rando

```
public class MultiplyWIthCarry {

    final long a = 0xffffda61L;
    long seed = System.nanoTime() & 0xffffffffL;

    //Parecido com o LGC, porem passa em
    //alguns testes estatisticos que o LGC falha,
    //se forem escolhidos bons parametros.
    public int nextInt() {
        seed = (a * (seed & 0xffffffffL)) + (seed
            >>> 32);
        // primeiro zera os 32 bits
        // mais significativo da semente,
        // depois multiplica e soma com a semenete
        // dividida por 2^32 com shift right.
        return Math.abs((int) seed);
    }

}
```

Tendo este código implementado, podemos percerber que os números gerados são de 32bit. Para gerar números que sejam múltiplos de 32, como 64, 128,

1024 e 4096 foi construído um loop que a cada iteração cria um novo objeto do tipo MultiplyWithCarry e pede o próximo número. Então esses números são concatenados até termos a quantidade de bits necessária. Abaixo temos alguns exemplos de números gerados com este algoritmo:

Tabela 2: Exemplo de números gerados com MWC.

Tamanho em bits	Número Gerado
31	1912360281
31	1697062396
63	8081633200613964543
63	7733642056664092968
127	139146135588267251463073470093232054794
127	142489858770018732778875980919996352105
255	348311646905702541767615237659744820453175766557708925451323983693993994889285
511	645125651205765754764882286551159244718955096747233878511475272983179178137820214456751926338795849037774611597961106016979593759252393518713154068292566 9
1023	843234234783062032061594719460187510965959165949705526449389660902731306653894097981693202770311094597090074742525306093839703240434063817374330155116049622936240225394626571025992239553322270068099356543488501420021198313110382245434512910248797317571809893797631884085882875442465747431544498774905513647822045	234770526248041897243529882238857384426575675682329022394697865754733923292356942880487807390403851116003449872364143142598563842862443282682822623273164679719961176400321538832913884277954840130632076028098452131254708636923727977028689522814816075742964783123559933815942019644918263036836284858913062917608434283318898496483642051374272849615710701005732385964820562253385861250239229528477842866099156768631228999838733546550937048284575111086003311425874880159565583762686068197904802269211504602791438771222104644788992833655639736878315613542079871822645891670988417379640460762198174542639509214971342685881

1.3 COMPARAÇÃO

Segundo a 3, podemos concluir que em média, o LGC executa em menos tempo utilizando as implementações apresentadas. Porém, é sabido que as implementações apresentadas podem não ser as mais eficientes, visto que a linguagem Java não é exatamente propícia para operações bitwise.

Tabela 3: Exemplo de números gerados com MWC.

Tamanho em bits	Elapsed Time LGC	ElapsedTime MWC	LGC - MWC
32	2.41671E-4s	3.77667E-4s	-1.3599598605651408E-4
64	5.482E-6s	5.161E-6s	3.209997885278426E-7
128	7.859E-6s	7.045E-6s	8.140000318235252E-7
256	1.2502E-5s	2.5166E-5s	-1.2664000678341836E-5
512	2.0995E-5s	2.1003E-5s	-7.999915396794677E-9
1024	5.0945E-5s	3.1665E-5s	1.9279999833088368E-5
2048	1.56663E-4s	2.00992E-4s	-4.4329004595056176E-5
4096	9.995E-5s	1.27286E-4s	-2.733599831117317E-5
8192	2.52915E-4s	8.7043E-5s	1.6587198479101062E-4
16384	1.79728E-4s	1.011E-4s	7.86279997555539E-5

2 NÚMEROS PRIMOS

Neste capítulo será mostrado os dois métodos utilizados para verificar a primalidade de um número. Primeiro apresentaremos quatro números inteiros para cada potência de dois, começando com o expoente igual 5 e terminando em 12, gerados pelos algoritmos apresentados anteriormente. Depois será mostrado os códigos dos dois métodos verificadores, de forma comentada. Por último será apresentado uma comparação baseada em complexidade do algoritmo e nos tempos apresentados pela implementações propostas.

Dado esses números, agora será apresentados os métodos que serão utilizados para verificar se esses números são primos

2.1 MILLER-RABIN

Este método contém alguns contribuidores, sendo os principais Gary L. Miller e Michael O. Rabin. O primeiro conseguiu desenvolver um teste determinístico e o segundo propôs uma modificação para trazer probabilidade para dentro do verificador. Segue um exemplo de implementação:

```
public class MillerRabin {

    private int rounds;

    public MillerRabin(int numberOfRounds){
        this.rounds = numberOfRounds;
    }

    public boolean test(BigInteger number){
        // verifica se o numero e par.
        if (number.divideAndRemainder(BigInteger.
            TWO)[1]
                    .equals(BigInteger.ZERO)) {
             return false;
        }
        //incrementador para a decomposicao do
            numero
        int powerOf2 = 0;
        // como e impar, diminuimos 1 para decompor
        BigInteger odd = number.subtract(BigInteger
            .ONE);
```

```
//loop da decomposicao em potencia de 2
//enquanto o mod por 2 for igual a 0
while (odd.mod(BigInteger.TWO).equals(0)) {
    // atualiza o incrementador
    powerOf2++;
    //atualiza o valor para que o loop
    // tenha um fim ...
    odd = odd.divide(BigInteger.TWO);
}



Random random = new Random();
for(int i = 0; i< this.rounds; i ++){
    BigInteger randomBase =
        selectRandomBase(number, random);
    // depois de ter selecionado a base,
        eleva ela
    // a potencia impar que sobrou da
        decomposicao
    // modulo o numero que se quer testar
    // este sera o divisor
    BigInteger divisor = randomBase.modPow(
        odd, number);

    //se divisor nao for igual a um e nao
        for igual ao numero - 2
    if( !divisor.equals(BigInteger.ONE) &&
            !divisor.equals(number.subtract
                (BigInteger.TWO))){
        if(!this.teste(divisor, number,
            powerOf2)){
            return false;
        }
    }
}
// se depois de todos os rounds
// o metodo nao tiver retornado
//o numero e um possivel primo.
//se o numero nao for primo
```

```
        // o algoritmo tem 4^-(numero de rodadas)
            probabilidade
        // de dizer que o numero eh primo.
        return true;
    }

    private BigInteger selectRandomBase(BigInteger
        number, Random random){
        BigInteger randomBase;
        // seleciona uma base entre 2 e (n-2)
        do {
            randomBase = new BigInteger(number.
                subtract(BigInteger.TWO).bitLength
                (), random);
        } while (randomBase.compareTo(BigInteger.
            TWO) < 0 || randomBase.compareTo(number
            .subtract(BigInteger.TWO)) > 0);
        return randomBase;
    }

    private boolean teste(BigInteger divisor,
        BigInteger number, int powerOf2){
        // verifica se o numero eh divisivel por
        // alguma potencia do divisior antes
            encontrado
        for(int j = 0; j < powerOf2; j++){
            divisor = divisor.modPow(BigInteger.TWO
                , number);
            // se for, retorna falso.
            if(!divisor.equals(number.subtract(
                BigInteger.ONE))){
                return false;
            }
        }
        return true;
    }
}
```

2.2 SOLOVAY–STRASSEN

Agora será apresentado uma implentação do algoritmo desenvolvido por Robert M. Solovay e Volker Strassen. Apesar de um teste já superado por outros, como pelo próprio Miller-Rabin, este algoritmo foi escolhido pela importância ao demonstrar a possibilidade de por o algoritmo RSA em prática. Assim como o Miller-Rabin também é um teste probabílistico, sendo possível gerar a respsota errada.

```
public class SolovayStrassen {
    private int rounds;
    public SolovayStrassen(int rounds){
        this.rounds = rounds;
    }

    public boolean test(BigInteger number){
        // verifica se o numero e par.
        if (number.divideAndRemainder(BigInteger.
            TWO)[1].equals(BigInteger.ZERO)) {
             return false;
        }
        Random random = new Random();

        for(int i =0; i < rounds; i++){
            BigInteger randomDividend =
                selectRandomDividend(number, random
                );
            // verifica o quociente da divisao
            BigInteger quotient = randomDividend.
                divide(number);

            //cria a potencia baseada em (numero -
                1 ) /2
            // que sera utilizada para testar se
            // o numero eh composto
            BigInteger power = number.subtract(
                BigInteger.ONE).divide(BigInteger.
                TWO);

            // se o quociente for igual a 0,
```

```
            // ou se o quociente mod numero sendo
                testado
            // for igual ao dividendo elevado a
                potencia achada
            // anteriormente, entao o numero e
                composto
            if(quotient.equals(BigInteger.ZERO) ||
                quotient.mod(number).equals(
                randomDividend.modPow(power, number
                ))){
                 return false;
            }
        }
        return true;
    }

    private BigInteger selectRandomDividend(
        BigInteger number, Random random){
        // seleciona um dividendo aleatorio entre 2
            e n - 1.
        BigInteger randomDividend;
        do {
            randomDividend = new BigInteger(number.
                subtract(BigInteger.ONE).bitLength
                (), random);
        } while (randomDividend.compareTo(
            BigInteger.TWO) < 0 || randomDividend.
            compareTo(number.subtract(BigInteger.
            ONE)) > 0);
        return randomDividend;
    }

    // nao podemos esquecer tambem que a
        probabilidade de erro 2^-(numero de rodadas
        )
}
```

2.3 COMPARAÇÃO

Será mostrado tabelas com diferentes números de rodada para o algoritmo de Miller-Rabin, enquanto o do Solovay–Strassen deixaremos fixado o número de rodadas em 2000000000. Começaremos com 1000, depois 2000 e por último 20000. A primeira comparação antes de partirmos para dados reais é da complexidade. O Miller-Rabin quando mal implementado tem O(NumeroRodadas*log$\hat{3}$(numeroSendoTestado)), mas podemos reduzir isso para O(NumeroRodadas*log$\hat{2}$(numeroSendoTestado)). Enquanto o Solovay–Strassen tem complexidade de O(NumeroRodadas*log$\hat{3}$(numeroSendoTestado)). Desta forma podemos esperar que os tempos de execução, considerando que o Miller-Rabin não tenha sido acelerado ao máximo, serão iguais. Todos os tempos aqui listados são em segundos.

Tabela 4: Tabela dos resultados e tempos dos algoritmos com 1000 rodadas. (MR = Miller-Rabin, SS = Solovay–Strassen, ET = Elapsed Time)

Índice	MR	SS	ET MR	ET SS	Diferença
1	Sim	Não	0,02721946	0,00082451	0,02639494
2	Sim	Não	0,00395581	0,00003180	0,00392401
3	Sim	Não	0,00421093	0,00001616	0,00419477
4	Sim	Não	0,00606996	0,00002150	0,00604846
5	Sim	Não	0,01363707	0,00002138	0,01361570
6	Sim	Não	0,00691495	0,00003078	0,00688417
7	Sim	Não	0,00607304	0,00003843	0,00603461
8	Sim	Não	0,00514969	0,00001684	0,00513284
9	Sim	Não	0,01137630	0,00002545	0,01135086
10	Sim	Não	0,01114735	0,00001635	0,01113100
11	Sim	Não	0,01215473	0,00003235	0,01212238
12	Sim	Não	0,01581122	0,00003540	0,01577581
13	Sim	Não	0,03879687	0,00002254	0,03877433
14	Sim	Não	0,05173407	0,00002865	0,05170542
15	Sim	Não	0,04396776	0,00001802	0,04394974
16	Sim	Não	0,02912524	0,00001844	0,02910680
17	Sim	Não	0,11564312	0,00002016	0,11562296
18	Sim	Não	0,10420472	0,00001519	0,10418953
19	Sim	Não	0,11902215	0,00002068	0,11900147
20	Sim	Não	0,13295890	0,00003048	0,13292842
21	Sim	Não	0,54929798	0,00001912	0,54927887
22	Sim	Não	0,49270492	0,00001969	0,49268523

23	Sim	Não	1,05076475	0,00001630	1,05074846
24	Sim	Não	1,06246156	0,00003153	1,06243002
25	Sim	Não	4,86770054	0,00002803	4,86767251
26	Sim	Não	4,78670085	0,00003337	4,78666748
27	Sim	Não	6,13922313	0,00002661	6,13919652
28	Sim	Não	5,04230766	0,00002859	5,04227907
29	Sim	Não	39,54266595	0,00003313	39,54263282
30	Sim	Não	38,74941144	0,00009253	38,74931891
31	Sim	Não	37,17281331	0,00003463	37,17277868
32	Sim	Não	42,53774093	0,00003393	42,53770700

Tabela 5: Tabela dos resultados e tempos dos algoritmos com 2000 rodadas. (MR = Miller-Rabin, SS = Solovay–Strassen, ET = Elapsed Time)

Índice	MR	SS	ET MR	ET SS	Diferença
1	Sim	Não	,07188966	,00358343	,06830623
2	Sim	Não	,01092295	,00004786	,01087509
3	Sim	Não	,00827426	,00001637	,00825789
4	Sim	Não	,01022792	,00001961	,01020831
5	Sim	Não	,01549784	,00015534	,01534250
6	Sim	Não	,01036804	,00002936	,01033868
7	Sim	Não	,00992531	,00002481	,00990050
8	Sim	Não	,00995677	,00003534	,00992143
9	Sim	Não	,02164093	,00001231	,02162862
10	Sim	Não	,01794020	,00002112	,01791908
11	Sim	Não	,01740474	,00001803	,01738670
12	Sim	Não	,01651986	,00001889	,01650097
13	Sim	Não	,05196184	,00001506	,05194679
14	Sim	Não	,05953822	,00002791	,05951031
15	Sim	Não	,04452142	,00001748	,04450393
16	Sim	Não	,05643745	,00001414	,05642330
17	Sim	Não	,19155928	,00001875	,19154053
18	Sim	Não	,17732421	,00001681	,17730740
19	Sim	Não	,17727300	,00002178	,17725122
20	Sim	Não	,18070891	,00002645	,18068246
21	Sim	Não	2,18059601	,00001692	2,18057909
22	Sim	Não	1,84947173	,00003092	1,84944081
23	Sim	Não	1,11954804	,00002089	1,11952716
24	Sim	Não	1,52860683	,00001426	1,52859256

25	Sim	Não	11,01797639	,00004364	11,01793276
26	Sim	Não	11,19984695	,00004088	11,19980607
27	Sim	Não	10,98346072	,00002964	10,98343108
28	Sim	Não	12,17771666	,00003624	12,17768042
29	Sim	Não	87,54197128	,00003224	87,54193904
30	Sim	Não	89,05598291	,00005919	89,05592372
31	Sim	Não	82,28403497	,00002042	82,28401455
32	Sim	Não	82,27667705	,00007727	82,27659978

Tabela 6: Tabela dos resultados e tempos dos algoritmos com 20000 rodadas. (MR = Miller-Rabin, SS = Solovay–Strassen, ET = Elapsed Time)

Índice	MR	SS	ET MR	ET SS	Diferença
1	Sim	Não	0,11171003	0,00072251	0,11098752
2	Sim	Não	0,07335955	0,00011614	0,07324341
3	Sim	Não	0,08186430	0,00001901	0,08184529
4	Sim	Não	0,05165693	0,00001826	0,05163867
5	Sim	Não	0,13971304	0,00012893	0,13958411
6	Sim	Não	0,13462999	0,00003382	0,13459617
7	Sim	Não	0,11232166	0,00002404	0,11229762
8	Sim	Não	0,11554731	0,00002949	0,11551782
9	Sim	Não	0,24554998	0,00003056	0,24551941
10	Sim	Não	0,22760821	0,00004261	0,22756561
11	Sim	Não	0,19712433	0,00003155	0,19709278
12	Sim	Não	0,20150501	0,00003131	0,20147369
13	Sim	Não	0,72246317	0,00003407	0,72242911
14	Sim	Não	0,73429265	0,00003507	0,73425758
15	Sim	Não	0,71114959	0,00003585	0,71111373
16	Sim	Não	0,43449985	0,00004040	0,43445945
17	Sim	Não	2,46091771	0,00002405	2,46089366
18	Sim	Não	2,56907771	0,00004308	2,56903463
19	Sim	Não	1,63278166	0,00002478	1,63275688
20	Sim	Não	3,39448872	0,00007026	3,39441846
21	Sim	Não	14,56426880	0,00003966	14,56422914
22	Sim	Não	14,36132480	0,00007242	14,36125237
23	Sim	Não	15,49475016	0,00004771	15,49470245
24	Sim	Não	16,97889106	0,00001516	16,97887590
25	Sim	Não	112,70947719	0,00002850	112,70944869

26	Sim	Não	119,64860039	0,00001713	119,64858326
27	Sim	Não	117,71940390	0,00002893	117,71937497
28	Sim	Não	116,97791633	0,00002738	116,97788895
29	Sim	Não	903,69619122	0,00002027	903,69617095
30	Sim	Não	849,73224558	0,00007467	849,73217091
31	Sim	Não	803,39407188	0,00002740	803,39404448
32	Sim	Não	858,20336123	0,00001957	858,20334166

Nas tabelas acima temos dois comportamentos inesperados. O primeiro é que enquanto o Solovay-Strassen tem sua duração muito baixa, o Miller-Rabin tem seu tempo de execução mais que dobrando quando alteramos a quantidade de bits para o dobro. Motivos para isso podem ser vários, mas o mais óbvio é a escolha da linguagem. Sabemos que Java não é a linguagem mais robusta, muito menos seu ambiente de execução. Outro possível problema é a dependência que a implementação tem da classe BigInteger para lidar com números exageradamente grandes. O Outro comportamento é que com 1000, 2000 e 20000 rodadas, o Miller-Rabin não foi capaz de identificar que os números não eram primos. Enquanto o Solovay-Strassen identificou muito fácil. Obviamente isso acontece pois o número de rodadas é muito diferente entre as duas implementações. Porém, isto é uma barreira de hardware/software disponíveis e escolhidos para o desenvolvimento deste trabalho. Portanto, de acordo com todos os códigos apresentados e dados listados, podemos concluir que neste caso específico a implementação do Solovay-Strassen apresentou melhores resultados.

Tabela 7: Tabela de números aleatórios primos.

Índice	Número
1	2029663043
2	1818701935
3	1269497125
4	2066721673
5	7630766861971458425
6	8305496549126277389
7	7622826425035437111
8	7040092047322450767
9	120298170997570680356444293518225934565
10	133804528563206615538729272032743639375
11	162811172248064604546734121441704956247
12	125680834290773339795990496296686958291

13	532155083967689218641582668965852684303369993186468386337902851851869973249 17
14	405333870247263026167259788537706824404514702470153574893717850444035440742 53
15	505022156239364597734448869382777643574085079171675596713409156252230368628 31
16	377347986664535848050412110198079426098136086636558184172648495977497272 57373
17	367840243432246132125590662450572122380143427498062315678456872589602184810477857670404503142832774527703592792175178308492505461418621570750 3034566100539
18	568257811125537004904474301687427493834125350012655625477486028031753852370391596453941570537170443377480260544128892227592764591285388238177 9991232856009
19	375287201257943102889370048513642439583267958094065176760970480636055650104952014766052185260587827457428974335715570652403320475451066954544 3375434319783
20	569713718113480598250938078758089827821386830748727008448469384452975996000295560604740221327203950408186257729813384481076318081250684317120 7462213122379
21	670701752415810975131761907743512871669539407816221213435414640942131165977773095340090744735172687809209125622836612169773670937086932924925 3244545419762336650855642750668113332762126675804618810400825102026210689501802632981977324424972338573753903057335024759508906758860793686103438842332983512823305611 3
22	760291364125964909791971605394571132797974330334266695024441103200865443998738026868908129527823832447840127253328190292216663380345595100757252642377871704719304371199696842112781597905137100839572701853160854507091111716263972802757345608221251697917759445569694683965497018484031176892317295875909588533333

23	661348196333970372376091515502039888175004564869572042732391418066869192664140142046744816550502314580421162016163678279885948160004851867176374371966129258674736333330706270400388563907834197998502391291482697746234057828034776636631231365877675311809321485256691730643635961661416914287079481384862449566 99
24	792620196168184386016501479061915459177839518590126453978528262329680559710239865855125167206529253122979458871585714980826441528674239985510504293607263102722750134272472381612728612825967627703593043373959011124397670664180660997900850936208871648970265765134270907407636709269867825623246827275913110100 25
25	133714793415464635973789698303199088286694669001435434202854646759739294633312220757326165019220550976142683097445604595702857143515171188507152110533138042156415169780179035231155521053338896357552437655402570660669441535592116314949616090859174303030099023036635748092221071968790954023764218223520874389024301995497375532567814079524389869303503624761242407144484167680558565484243691046207604500881039257841371097774381499030773897745523002451469258262185610232995750391836810740149725017370517911214724262558219503542441202221931275745578344373455959833468168697615072586276771875841575999246135091651538706178 73

26	125030477899006489768061570459889771198664944149710340198597920379438677180348656298718787080656030736192291694797884430332448907700525510617613302101637887065851606359558658348434064871594460770217315390252875897312893728048787949094147680811802458571320235772381962291325871408592529789786565285436471624372860992321988647079750372095051177334033412668309783646378377538417199563831740706973473563155848074850439920219476028323677849030555933812380901288635031883948444396462536338948016777188831640886768112528915590406336597504874503785784832452034060391454582970487689179339898376925734427548355926793734325545 3
27	133577224636043168223001764612334226232164344752427605011945510531768247105214463474569389079965551830170706843277456313365786453354117205258454953478037898052753381920126832801334795056704625870305381679849225444795533265365858613023170139791621027142223676521036221200098765608147354681152931061765204327103009148722745343698410289395422281239530378140486114331807599169337465677463273621233979628655309372716018440240463585154973556121986150979524223731276225404497982797757778086845843921358865956027529689846395133093924457609859405346171026351088716699034117492472263689698192672115748683130797488012972909271545

28	125263603375645780282679977951114118621084339759625243666465719659110032068044403779409841280612417997059565106342490708151211091425432792214947738046749213414359905556918786445692685719387753659889421574714367804527828550735319167294140541298146286208242970111851116051679628003840528499242442035317022542633441050922498671003332914850593976506067068467455492962155004552240887576545284997654403377905307500372363602151338931092977091057096967549885621964773963091801550322480866672475667301407828873452242660354603313264096361823071130104961098380564536389640702290575527505598930201452940781741529134390780515780 95

29	499746747975434346547803065320274051217990554840731381688263896692802660854111902411902403061984709585390010574444897735366506055088302981335651857083921318642673073029198730819262308799867533572399555731969051420374849041998870167963619586334566768917965948038618946523148678849842857603014373134525299794078691886854810560390801542366497754389864827876567743226997541666436934808287878511032540365621462812365783439815947149342550352409161212473185279723882201234636702482703811610985198797601698765040350846089538063645940015841884912159090266133857488533153697232142488724619024194165734395422200897639482130324346092642743868957401067686701311643304660683808895920912083251449317301573123698685442248411844929415396569080330032706904396425376098432102702117804651103325133487024320963295387572001417776428516000839066484952823489770750588525748485439508388069460307622122685302210448468032194082270409103505112251971512738481129282130490855126442787777504646995883111531869797042301516807796432401248092524198058212037042623111516381653020552046883547931623470338418403100633871651476961221162337212504760486428074592907899261556993425244290941359482177349655367850030580931137774445504153901241077612086885969925568804516664 7

30	432114926342767837544959489285150954279888344732304206770369544099393688515473327858725330971962099240899366022233574589952236588965669362843267151132262295635060009283229126829946664651218769638312916465970488070824746455561271543140415630215043676943245598118762907208336879339052210095927667533488906883853326663510276375504296803086034828830631509559410958277782360183029062792845278293529267816585437310868449287878891193213213961515976934853802389067863325593371534230879904910064033393431828142173135355853891914438227759932085414301988924045664298678306156612095259246184105510126134462473498713618807433661681146056501423663209344414662492116402528348449131687869447004895777579260792741131733963488863391770560789829263924405878300938694356411304690774039014610708385509375473484633963421793704283539257238171026902372961631009108740918545064851871410583619249178623129813087588198063142997631324761183981519104331159761272825294666519454203222490986401509182133072708018099358804111128757126119799891809945505624668650242393907132930393987657484948152291278353415278706823686907122629052414085228169461703773520710224314721179400573669334218922288680491298003668731953833497307941370522364693099839753174419531637012946 07

31	499847108417225235751536940132220366638729585788374125067922226399847912221665367406441386316019919749008786504188137495104771704570044258171781906334942405591895003209550712229819958348638993639554987032499451746182431430747923606787827943682450646195931806709793459545427861446166673581486204343753192001427548131039602149231034148161957399472032144667167339482484051008349418379225637301211540009993348048857914918463104413011608453461007023750445275711244203186477417166984409743346620815022580639067665928070416831401334320102962864103802615150600780595750928420770538349841488326639783208890322039522975640689774307420827158042849744363116295622495158593440420082842912954754329639410292075422563742797134288724248503355480122511726954442554501813458822108013092582139606396597704411277091773337878554082698932329124609978637258410253828247891840586878572438846045526792186519334872691828130202084302986569097880214137318213951287760105260566305668886701751572123123030744229596311085928649397049188660621649819209341335964471527284138390413795960635088224535323182310224929215435158068506580679094760737149237515791675172995820123122612946441768025046004730929852550240879355530197600773899977509270223549978989274820978348327

32	372742333145688996323365073390500197896292314960606202968608315999729983044114028859952840381222398523702755637336721939650587594947948846083895452551038228336854825276872895786224234356128641663569922373899800575418838905344675336021831464642319789751403194208976586070433110370385055926185680364430619925195197190093383445898304448843723428471165180909743597407524196736563472146132762651519392059492481647581696620455398038754205441790456710808957338029291158712491796213377088570918366022991598745398149513722049588646682012126984835436923154339859963060749253015943908392637014487652337368641022071068146610435061570703476699267445352439071087081995150601852181089675322885031475237424423825157621493369657368871491323795049083018271320180871440703206214562930817422554517978525042848309609094634861385077783117198030672816309307049699133097829526952730675856131310529091386790311055774003784016300132901113928930552186247701272535878229195775060579354446808334159421474542048448365857523764326504303143210843916454784417576896359571485558844620756445604704900932601553003095361868904046835396305151371360444659458136647671905287871319345642999654888288552136112208707528743230384397607054407903282536454297248814422335496429885