Mathematics of Cryptography – Lab Assignment

Jonas Lorenz

Exercise 1

a) Using RandomPrime, make a series of examples of how long it takes to generate primes with 500 digits and 1000 digits (say at least 20 examples each). Comment on why the timing fluctuates.

In the following, we will simply loop through 20 iterations of RandomPrime, timing every single one. The times are the output.

```
In[ ]:= L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^100, 10^101}]];
         L = Append[L, t]]
        L
        L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^50, 10^51}]];
         L = Append[L, t]]
        L
Out[ ]=
        {0.001473, 0.002259, 0.004975, 0.002329, 0.001801, 0.004745,
         0.001051, 0.00208, 0.003047, 0.001633, 0.006092, 0.007035, 0.005124,
         0.003616, 0.001478, 0.01183, 0.001528, 0.003755, 0.003083, 0.006029}
Out[ ]=
        {0.000836, 0.001907, 0.000444, 0.000593, 0.000375, 0.001961,
         0.00177, 0.001663, 0.000999, 0.000563, 0.000575, 0.002944, 0.001846,
         0.003623, 0.000739, 0.002781, 0.000461, 0.00037, 0.001219, 0.000614}
```

The timing fluctuates because it might take longer or shorter for a random integer to be prime.

b) Use the formula in Section 3.4.1 (as in Example 3.22) to approximate, in the cases above, how many integers the program needs to check in order to find a prime.

Here we will just use Mathematica for simple computations.

```
In[•]:= (10^100 / Log[10^100] - 10^99 / Log[10^99]) / (10^100 - 10^99)
Out[•]=
        (- (1 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⋱
                 000 000 000 000 000 000 000 000 000 000 000 000 / Log[
                 1 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⋱
                  000 000 000 000 000 000 000 000 000 000 000 000]) +
           10 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⋱
               000 000 000 000 000 000 000 000 000 000 000 / Log[
               10 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⋱
                000 000 000 000 000 000 000 000 000 000 000 000]) /
         9 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 000 ⋱
          000 000 000 000 000 000 000 000 000 000

In[•]:= N[%68]
Out[•]=
        0.00433807

In[•]:= N[%66]
Out[•]=
        3.90426 × 10^97
```

We now want to find out when exactly the probability of not finding a prime by randomly choosing numbers when choosing x numbers is below one percent.

```
In[•]:= Reduce[{(1 - 0.004338070582423054)^x < 0.01, x > 0}, x, Reals]

        ••• Reduce : Reduce was unable to solve the system with inexact coefficients. The answer was obtained by
              solving a corresponding exact system and numericizing the result.
Out[•]=
        x > 1059.27
```

Testing about 1060 numbers reduces the chances to not finding a prime to under one percent.

c) Approximately how large prime numbers can we generate if we let the computer work at most 60 seconds?

In the following we time how long it takes to generate random primes. We use steps of 300 digits to avoid waiting too long.

```
In[•]:= L = {};
        For[i = 0, i < 10, i++,
         {t, p} = Timing[RandomPrime[{10^(i * 300), 10^(i * 300 + 1)}]];
         L = Append[L, t]]
        L
Out[•]=
        {0.00011, 0.139781, 0.319675, 0.351218,
         7.44567, 1.55288, 10.1345, 79.7093, 4.09257, 57.9476}
```

```
In[ ]:= L = {};
        For[i = 1, i < 11, i++,
         {t, p} = Timing[RandomPrime[{10^(i*300), 10^(i*300 + 1)}]];
         L = Append[L, t]]
        L
Out[ ]=
        {0.059422, 0.110699, 1.80998, 14.3081,
         0.295549, 6.77845, 3.41971, 250.36, 34.7922, 141.046}
```

It seems like going above some 2000 digits starts getting risky if one wants to avoid computing for longer than one minute. It is still however possible to luck out and take way less.

d) How long does it take for Mathematica to do modular arithmetic (summing, multiplying and taking powers) with numbers of this size?

We do some simple calculations and time them.

```
In[ ]:= {t, p} = Timing[RandomPrime[{10^2000, 10^2001}]];
        {t, q} = Timing[RandomPrime[{10^2000, 10^2001}]];

In[ ]:= {t, a} = Timing[p + q];

In[ ]:= t
Out[ ]=
        0.000016

In[ ]:= {t, a} = Timing[p*q];

In[ ]:= t
Out[ ]=
        0.000023

In[ ]:= {g, {m, n}} = ExtendedGCD[p, q];

In[ ]:= {t, a} = Timing[Mod[p*q, m]];
        t
Out[ ]=
        0.000032

In[ ]:= {t, a} = Timing[PowerMod[p, q, m]];
        t
Out[ ]=
        0.000013
```

Modular arithmetic goes fast still even if the numbers are very large.

e) Finally, approximately how large a prime can we deterministically check using ProvablePrimeQ if we let the computer work at most 60 seconds?

```
In[ ]:= ClearAll["Global`*"]

In[190]:=
        << PrimalityProving`
```

```
In[◦]:= L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^100, 10^101}]];
         {z, b} = Timing[ProvablePrimeQ[p]];
         L = Append[L, z]]
        Mean[L]
Out[◦]=
        2.81228
```

```
In[◦]:= L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^150, 10^151}]];
         {z, b} = Timing[ProvablePrimeQ[p]];
         L = Append[L, z]]
        Mean[L]
Out[◦]=
        14.6367
```

```
In[◦]:= L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^170, 10^171}]];
         {z, b} = Timing[ProvablePrimeQ[p]];
         L = Append[L, z]]
        Mean[L]
Out[◦]=
        20.9552
```

```
In[◦]:= L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^200, 10^201}]];
         {z, b} = Timing[ProvablePrimeQ[p]];
         L = Append[L, z]]
        Mean[L]
Out[◦]=
        $Aborted
Out[◦]=
        63.8925
```

While the computation got aborted since it seemed to be going for a bit too long, the mean being about 64 seconds gives a good indication that 201 digits is a good approximation.

```
In[191]:=
        L = {};
        For[i = 0, i < 20, i++, {t, p} = Timing[RandomPrime[{10^199, 10^200}]];
         {z, b} = Timing[ProvablePrimeQ[p]];
         L = Append[L, z]]
        Mean[L]
Out[193]=
        44.7471
```

One digit less takes far under 60 seconds, so 201 should be a good estimate for the digits.

Exercise 2

a) Find out approximately how large an integer of the form n = p*q, with p and q primes with the same number of digits, Mathematica can factor in less than 60 seconds.

We first want to generate two random primes with the same number of digits and then factor the product using Mathematica.

```
In[ ]:= {t, p} = Timing[RandomPrime[{10^100, 10^101}]]
        {t, q} = Timing[RandomPrime[{10^100, 10^101}]]
        {t, r} = Timing[ FactorInteger[p * q]]
Out[ ]=
        {0.001102,
         31 681 524 918 849 037 065 188 763 682 263 758 181 905 209 809 276 042 081 116 080 618 165
           356 569 653 169 697 128 458 441 586 006 347}
Out[ ]=
        {0.003865,
         18 757 312 073 663 383 236 839 831 947 082 909 098 133 502 226 773 153 763 767 564 042 275
           094 548 244 762 515 142 480 247 395 320 191}
Out[ ]=
        $Aborted
```

The computation had to be aborted.

```
In[ ]:= L = {};
        For[i = 0, i < 5, i++, p = RandomPrime[{10^29, 10^30}];
         q = RandomPrime[{10^29, 10^30}];
         {z, b} = Timing[ FactorInteger[p * q]];
         L = Append[L, z]]
        Mean[L]
Out[ ]=
        48.7828

In[ ]:= L = {};
        For[i = 0, i < 3, i++, p = RandomPrime[{10^30, 10^31}];
         q = RandomPrime[{10^30, 10^31}];
         {z, b} = Timing[ FactorInteger[p * q]];
         L = Append[L, z]]
        Mean[L]
Out[ ]=
        121.496
```

The previous computations give a good idea about the size of numbers we can factorize under one minute. About 29 digits seem to be the maximum.

b) Generate larger and larger random primes p and q (make say 25 examples and in each case increase the number of digits of p and q by 1) and time (take the mean time for 5 examples) how long it takes to factor n=p*q (using FactorInteger). Does the complexity seem to be O(sqrt(n))? Write both an

answer and a comment.

We want to make some examples and time them. In the end we first square the time and then take the logarithm to see if the progression seems linear.

```
In[ ]:= L = {};
T = {};
R = {};
For[i = 0, i < 25, i++;
 For[j = 0, j < 5, j++, p = RandomPrime[{10^i, 10^(i + 1)}];
  q = RandomPrime[{10^i, 10^(i + 1)}];
  {z, b} = Timing[ FactorInteger[p * q]];
  T = Append[T, z]];
 L = Append[L, Mean[T]];
 R = Append[R, Log[Mean[T]^2]];
 T = {};
]
L
R
```

```
Out[ ]=
{0.0000104, 7.2 × 10^-6, 9.8 × 10^-6, 0.0000386, 0.0005278, 0.0005748,
 0.0007694, 0.001103, 0.0015372, 0.0024598, 0.003403, 0.0040926,
 0.0077496, 0.0259212, 0.0312154, 0.051829, 0.0601392, 0.101819,
 0.143627, 0.241978, 0.347848, 0.660611, 1.72257, 3.13345, 4.29716}
```

```
Out[ ]=
{-22.9474, -23.6829, -23.0663, -20.3245, -15.0936, -14.923,
 -14.3398, -13.6194, -12.9556, -12.0154, -11.3662, -10.9971,
 -9.72023, -7.30539, -6.93369, -5.91961, -5.62219, -4.56911,
 -3.88108, -2.83782, -2.11198, -0.829179, 1.08763, 2.28427, 2.91591}
```

The complexity could very well be the square root of n where n is the approximate size of the prime factors. We take the logarithm of the squared times to adjust for the fact that the inputs are scaled exponentially.

c) You are now given the integer: n=928342276875151655453067207230702172100445455166524892821980417805423272154669996174491922126161226555677161044539663542241316251870342261768001956310893642278288195626794098168307587471718593874971725253 Do you expect to be able to factor it using FactorInteger within 60 seconds? Try this and comment upon the result.

```
In[ ]:= Timing[FactorInteger[
          928 342 276 875 151 655 453 067 207 230 702 172 100 445 455 166 524 892 821 980 417 805 423 ⋱
           272 154 669 996 174 491 922 126 161 226 555 677 161 044 539 663 542 241 316 251 870 342 ⋱
           261 768 001 956 310 893 642 278 288 195 626 794 098 168 307 587 471 718 593 874 971 725 ⋱
           253]]
Out[ ]=
        {0.002709, {{1151, 1},
          {806 552 803 540 531 412 209 441 535 387 230 384 101 168 944 540 855 684 467 402 621 898 ⋱
            717 004 478 427 451 063 850 497 068 776 043 923 264 258 075 186 501 774 319 127 933 857 ⋱
            812 564 524 762 776 986 006 639 685 741 264 662 722 934 985 497 469 567 088 265 747 151 ⋱
            803, 1}}}
```

Factorization here went so fast since one of the factors is very small. Testing whether primes up to that number divide n can be done fast as modular arithmetic can be done fast as seen in exercise 1d).

Exercise 3

a) Make a series of examples (letting the number of digits of p grow, and for each choice of number of digits take the mean value of at least 5 examples) to compare the time it takes to solve the DLP using each of these three functions and also MultiplicativeOrder.

In the following we will use the functions and time the duration it takes to finish. We take the logarithm in the end to adjust for the exponential scaling of the input.

```
In[ ]:= Naive = Function[{g, p, h}, Module[{i, st, b, c, res}, st = 0;
            b = 1;
            c = Mod[h, p];
            For[i = -1, st == 0 && i < p - 1, i++, If[b == c, st = 1];
             b = Mod[b * g, p]];
            If[st == 0, res = "no solution", res = i];
            res]];
```

```
In[]:= L = {};
       T = {};
       R = {};
       For[i = 0, i < 6, i++;
        For[j = 0, j < 4, j++, g = RandomInteger[{10^i, 10^(i + 1)}];
         p = RandomPrime[{10^(i + 1), 10^(i + 2)}];
         h = RandomInteger[{10^i, 10^(i + 1)}];
         {z, b} = Timing[ Naive[g, p, h]];
         T = Append[T, z]];
        L = Append[L, Mean[T]];
        R = Append[R, Log[Mean[T]]];
        T = {};
       ]
       L
       R
Out[]=
       {0.0007235, 0.010651, 0.0556888, 0.458596, 5.30692, 53.1257}
Out[]=
       {-7.23141, -4.5421, -2.88798, -0.779585, 1.66901, 3.97266}
```

The practical complexity seems to be linear as expected. The number of examples here is limited by the poor performance of the naive algorithm.

```
In[41]:= Shanks =
           Function[{g, p, h}, Module[{res, NN, n, gn, gst, hst, lst1, lst2, ag, ah, st, i},
             If[h == 1, res = 0, NN = MultiplicativeOrder[g, p];
              n = Round[Sqrt[NN]] + 1;
              gn = PowerMod[g, -n, p];
              lst1 = {1}; lst2 = {h};
              gst = 1; hst = h; ag = 0; ah = 0; st = 0;
              For[i = 0, st == 0 && i < n, i++, gst = Mod[g * gst, p];
               lst1 = Append[lst1, gst];
               If[MemberQ[lst2, gst], st = 1; ag = Position[lst2, gst]];
               hst = Mod[gn * hst, p]; lst2 = Append[lst2, hst];
               If[MemberQ[lst1, hst], st = 2; ah = Position[lst1, hst]]];
              If[st == 1, res = Mod[(ag[[1, 1]] - 1) * n + i, NN],
               If[st == 2, res = Mod[ah[[1, 1]] - 1 + i * n, NN], res = "no solution"]]];
             res]];
```

```
In[ ]:= L = {};
        T = {};
        R = {};
        For[i = 0, i < 8, i++;
         For[j = 0, j < 4, j++, g = RandomInteger[{10^i, 10^(i + 1)}];
          p = RandomPrime[{10^(i + 1), 10^(i + 2)}];
          h = RandomInteger[{10^i, 10^(i + 1)}];
          {z, b} = Timing[ Shanks[g, p, h]];
          T = Append[T, z]];
         L = Append[L, Mean[T]];
         R = Append[R, Log[Mean[T]]];
         T = {};
        ]
        L
        R
Out[ ]=
        {0.0002335, 0.0004745, 0.00098825,
         0.0103345, 0.0651123, 0.648951, 5.58057, 81.8119}
Out[ ]=
        {-8.36233, -7.65325, -6.91957, -4.57227, -2.73164, -0.432398, 1.71929, 4.40442}
```

The times seem to go somewhat linear, which might be due to high memory usage.

```
In[ ]:= PollardRhoFp = Function[{g, p, h}, Module[
        {res, i, ng, nh, ngh, xi, yi, ai, bi, ci, di, n1, n2, st, j, d, w, stt, k}, i = 0;
        If[h == 1, res = 0, ng = MultiplicativeOrder[g, p];
         nh = MultiplicativeOrder[h, p];
         ngh = GCD[ng, nh];
         xi = 1;
         yi = 1;
         ai = 0;
         bi = 0;
         ci = 0;
         di = 0;
         n1 = Floor[p / 3];
         n2 = Floor[2 * p / 3];
         st = 0; While[i < p - 1 && st == 0, i ++;
          Which[xi < n1, xi = Mod[g * xi, p];
           ai = ai + 1, xi < n2, xi = PowerMod[xi, 2, p];
           ai = Mod[2 * ai, ng];
           bi = Mod[2 * bi, ngh], True, xi = Mod[h * xi, p];
           bi = bi + 1];
          For[j = 0, j < 2, j ++, Which[yi < n1, yi = Mod[g * yi, p];
             ci = ci + 1, yi < n2, yi = PowerMod[yi, 2, p];
             ci = Mod[2 * ci, ng];
             di = Mod[2 * di, ngh], True, yi = Mod[h * yi, p];
             di = di + 1];];
          If[xi == yi && Mod[bi - di, ngh] ≠ 0, st = 1; d = GCD[di - bi, ng];
           w = Mod[(ai - ci) * ExtendedGCD[di - bi, ng][[2, 1]], ng];
           stt = 0;
           k = -1;
           While[stt == 0 && k < d, k ++;
            If[Mod[PowerMod[g, w / d + k * ng / d, p] - h, p] == 0, res = w / d + k * ng / d;
             stt = 1]]]];
         If[st == 0, res = "no solution found?", If[stt == 0, res = "no solution"]]];
        res]];
```

```
In[•]:= L = {};
T = {};
R = {};
For[i = 0, i < 11, i++;
 For[j = 0, j < 4, j++, g = RandomInteger[{10^i, 10^(i + 1)}];
  p = RandomPrime[{10^(i + 1), 10^(i + 2)}];
  h = RandomInteger[{10^i, 10^(i + 1)}];
  {z, b} = Timing[ PollardRhoFp[g, p, h]];
  T = Append[T, z]];
 L = Append[L, Mean[T]];
 R = Append[R, Log[Mean[T]]];
 T = {};
]
L
R
Out[•]=
{0.00035475, 0.001134, 0.00333475, 0.0091915, 0.017949,
 0.0481133, 0.0739178, 0.621775, 1.60452, 16.0367, 64.0106}

{-7.944097241924122`, -6.7820040736765765`,
 -5.70335756494312`, -4.689476135041348`, -4.02022087596133`,
 -3.034197672029235`, -2.604802290423651`, -0.4751773900945192`,
 0.47282168604114844`, 2.774878660827929`, 4.159047983825342`}
```

The relationship here seems also rather linear, however the jumps sometimes get smaller, so the times could also follow the square root.

```
In[•]:= L = {};
T = {};
R = {};
For[i = 0, i < 11, i++;
 For[j = 0, j < 4, j++, g = RandomInteger[{10^i, 10^(i + 1)}];
  p = RandomPrime[{10^(i + 1), 10^(i + 2)}];
  {z, b} = Timing[ MultiplicativeOrder[g, p]];
  T = Append[T, z]];
 L = Append[L, Mean[T]];
 R = Append[R, Log[Mean[T]]];
 T = {};
]
L
R
Out[•]=
```

$\{0.0000105, 7. \times 10^{-6}, 8. \times 10^{-6}, 8.25 \times 10^{-6}, 9.5 \times 10^{-6},$
$0.00002125, 0.00002425, 0.000037, 0.00012, 0.000136, 0.0000485\}$

```
Out[•]=
{-11.4641, -11.8696, -11.7361, -11.7053, -11.5642,
 -10.7592, -10.6271, -10.2046, -9.02802, -8.90286, -9.93395}
```

MultiplicativeOrder does not seem slow at all. It therefore probably is not the reason for our algorithms slowing down.

b) Does the theoretical complexity for the three functions above match the one in the examples? Comment upon any differences.

The naive algorithm goes slow as expected and already fails to deliver a fast result when considering numbers with 6 to seven digits. The other two algorithms, especially Pollard’s rho algorithm deliver relatively fast results up to 11 digits. However even they slow down considerably at some point. Shanks begins to use more and more memory, which also slows down algorithms when implemented. Pollard’s rho algorithm performs best and could be closest to the theoretical complexity. One would have to go to higher digit counts to verify this.

Exercise 4

a) Show that, Pr(N is prime | the Miller Rabin test fails T times) is larger than or equal to 1-ln(N)/4^T, using the (vague) statement (3.10) on page 134 in [HPS].

We want to use Bayes' theorem . First off, we have P (N prime) = 1/ln (N) and P (T fails) = P (T fails | N prime) P (N prime) + P (T fails | N composite) P (N composite) . Call "N prime" A and "T fails" B . Then P (A | B) = (P (B | A) P (A))/(P (B | A) P (A) + P (B | A^c) P (A^c) . Further clearly P (B | A) = 1. We also know that about 75 % of numbers under N are witnesses if N is composite . Thereby P (B | A^c) = 1/4^T . Thus we have P(A|B) =1/ln(N) *1/(1/ln(N) + 1/4^T*(1-1/ln(N))) = 4^T/(4^T + ln(N) - 1) when multiplying divisor and numerator by ln(N)*4^T. Further 4^T/(4^T + ln(N) - 1) = 1 - ln(N)/(4^T+ln(N)-1) + 1/(4^T+ln(N)-1) >= 1 - ln(N)/4^T since we are omitting something positive and making the divisor of the negative property smaller.

b) Using the bound above, make a Miller-Rabin test of your own to check with probability at least 99.9999% that the following number is a prime: q=863190718341645771003614298695024893990207192546856205218226372307466528127232538364530032707660046143282952063932885298283116284157856527266539861069097142611570511777538875474993819960760889161456633

We will implement MillerRabin and then test it.

```
In[ ]:= q =
         863 190 718 341 645 771 003 614 298 695 024 893 990 207 192 546 856 205 218 226 372 307 466 ⋱
           528 127 232 538 364 530 032 707 660 046 143 282 952 063 932 885 298 283 116 284 157 856 527 ⋱
           266 539 861 069 097 142 611 570 511 777 538 875 474 993 819 960 760 889 161 456 633
Out[ ]=
        863 190 718 341 645 771 003 614 298 695 024 893 990 207 192 546 856 205 218 226 372 307 466 ⋱
          528 127 232 538 364 530 032 707 660 046 143 282 952 063 932 885 298 283 116 284 157 856 527 ⋱
          266 539 861 069 097 142 611 570 511 777 538 875 474 993 819 960 760 889 161 456 633
```

```
In[ ]:= MillerRabin =
          Function[{n, prob}, Module[{res, i, T , a, test, dummy, k, q, check}, i;
            test = 0;
            T = 0;
            res = 0;
            k = FactorInteger[n - 1][[1]][[2]];
            q = (n - 1) / (2 ^ k);
            If[EvenQ[n], res = 1];
            While[test < prob && res == 0,
             check = T;
             a = RandomInteger[{2, n - 1}];
             If[(GCD[n, a] > 1 && GCD[n, a] < q), res = 1];
             a = PowerMod[a, q, n];
             If[Mod[a, n] == 1, T++,
              For[i = 0, i < k, i++, If[Mod[a, n] == n - 1, T++;
                 Break[], a = PowerMod[a, 2, n]]]];
             If[T - check == 0, res = 1];
             test = 1 - Log[n] / (4 ^ T);
            ];
            If[res == 0, Print["There is a chance the number is prime!"],
             Print["The number is composite!"]]]];
```

```
In[ ]:= MillerRabin[q, 0.999999]

        There is a chance the number is prime!
```

I unfortunately did not time this, but it took a good 40 minutes if not mistaken. This might be due to FactorInteger, which in this case is probably very slow.

c) Assuming that we know that q is prime, make another Miller-Rabin test to check with probability at least 99.9999% that p=2*q+1 is a safe prime.

```
In[ ]:= Timing[MillerRabin[2 * q + 1, 0.999999]]

        There is a chance the number is prime!

Out[ ]=
        {0.014281, Null}
```

This went really fast, which is a bit concerning. I could not come up with a reason for this. Maybe a lot of the needed computations had been made in b). Especially the factoring of q.

d) Now, find an element in F_p^* of order q. Explain why this is a simple task.

To find an element of order q, we can consider some a in F_p^* that is not -1 or 1. In that case it will not have order 1 or 2 since x^2-1 has exactly two zeroes in F_p^*. Since p-1 = 2q, a can now only have order q or p-1. If the order of a is p-1, squaring will yield an element of order q. If it already has order q, squaring it will also result in order q. The order cannot be smaller since q is prime and would otherwise be divided by the order of a and the order of a^2 cannot be 1.

e) Finally, take a random composite integer n with 200 digits and check how

many, out of 100 random integers, are Miller-Rabin witnesses for compositeness of the integer n. Comment upon the result.

```
n = RandomInteger[{10^199, 10^200}];
While[EvenQ[n], n = RandomInteger[{10^199, 10^200}]];
n
t = 0;
q = (n - 1) / (2^k);
For[i = 0, i < 100, i++,
  a = RandomInteger[{2, 10^199}];
  a = PowerMod[a, q, n];
  If[Mod[a, n] == 1, t++,
   For[j = 0, j < k, j++, If[Mod[a, n] == n - 1, t++;
     Break[], a = PowerMod[a, 2, n]]]];
 ];
100 - t
```

```
Out[236]=
68 366 325 767 593 069 049 686 702 512 714 413 699 248 718 503 501 701 425 868 947 611 962 296 ⋱
  813 680 340 107 891 742 951 457 653 377 911 760 852 024 977 007 957 627 147 197 401 037 415 ⋱
  840 638 836 199 588 038 469 190 320 706 288 463 849 981 580 653 675 865 750 753

Out[238]=
$Aborted

Out[239]=
2

Out[242]=
100
```

Surprisingly often, all tested numbers are witnesses.

Exercise 5

The ElGamal public key cryptosystem is set up in the following way:
p=46935289616461321113779007835893780311135345411
g=2
A=3

You get the following ciphertext (which you know is encoded using the function “TextToCode” given in the Lab introduction file):
c1=8778664806307078139973418878384520848855131312
c2=16291448994917795033553964049510036051216903012

Find the plaintext using the Pohlig-Hellman algorithm and decode it using the function “CodeToText” given in the Lab introduction file.

We will implement PohligHellman and then solve the DLP with it.

```
In[73]:= CodeToText = Function[x, Module[{nmb, lst, tmp}, nmb = x;
             lst = {}; tmp = 0;
             While[nmb ≠ 0, tmp = Mod[nmb, 2^8];
              lst = Append[lst, tmp];
              nmb = (nmb - tmp) / 2^8];
             FromCharacterCode[lst]]];
```

```
In[68]:= PohligHellman = Function[{g, p, h},
           Module[{N, rlist, mlist, factorlist, llength, res, pow, sh, i},
             N = MultiplicativeOrder[g, p];
             factorlist = FactorInteger[N];
             rlist = {}; mlist = {};
             llength = Length[factorlist];
             For[i = 1, i < llength + 1, i++,
              pow = PowerMod[factorlist[[i]][[1]], factorlist[[i]][[2]], p];
              mlist = Append[mlist, pow]];
             For[i = 1, i < llength + 1, i++,
              sh = PohligE[PowerMod[g, N / mlist[[i]], p], factorlist[[i]][[1]],
                 PowerMod[h, N / mlist[[i]], p], factorlist[[i]][[2]], p];
              rlist = Append[rlist, sh]];
             res = ChineseRemainder[rlist, mlist];
             res]];
```

```
In[61]:= PohligE = Function[{g, q, h, e, p}, Module[{L, M, i, a, x, hgqin, gpow, res}, i = 0;
             L = {};
             M = {};
             gpow = 1;
             hgqin = h;
             a = PowerMod[g, q ^ (e - 1), p];
             x = Shanks[a, p, PowerMod[h, q ^ (e - 1), p]];
             L = Append[L, x];
             M = Append[M, 1];
             For[i = 1, i < e, i++,
              gpow = PowerMod[g, x * M[[i]], p];
              hgqin = Mod[hgqin * PowerMod[gpow, -1, p], p];
              x = Shanks[a, p, PowerMod[hgqin, q ^ (e - i - 1), p]];
              L = Append[L, x];
              M = Append[M, q ^ i]
             ];
             res = L.M;
             res
            ]];
```

Small test.

```
In[62]:= PohligE[5448, 5, 6909, 4, 11 251]
Out[62]=
         511

In[69]:= {t, a} = Timing[
           PohligHellman[2, 46 935 289 616 461 321 113 779 007 835 893 780 311 135 345 411, 3]]
Out[69]=
         {36.0495, 45 349 414 319 770 996 556 255 505 100 816 573 064 904 553 782}

In[70]:= c1 = 8 778 664 806 307 078 139 973 418 878 384 520 848 855 131 312;
         c2 = 16 291 448 994 917 795 033 553 964 049 510 036 051 216 903 012;
         p = 46 935 289 616 461 321 113 779 007 835 893 780 311 135 345 411;
         m = Mod[PowerMod[PowerMod[c1, a, p], -1, p] * c2, p]
Out[72]=
         11 377 403 073 054 103 083 557 395 451 482 074 081 367

In[74]:= CodeToText[m]
Out[74]=
         What ho! what ho!
```

Exercise 6

Let us set up RSA as in the textbook based on the following :

p = 1019837180925691653494215912076484435397939873560 13

q = 1432224163789235372128366372536713115610827824790 67

e = 65537

You know that Bob is going to send you a message which is either "Yes" or "No" . Say that Bob sent the ciphertext
c =
14107168815324318090835317276893095047469496348068905430075293831119265721969070956633661344299377669
where Bob has used RSA, as described in the textbook, and the encoding function "TextToCode" given in the Lab introduction file .

a) This is not secure. Show how Eve can find which message it is without knowing our private decryption key.

We will simply try both messages and raise them to the power e.

```
In[ ]:= TextToCode = Function[x, Module[{lst, res, i}, lst = ToCharacterCode[x];
            res = 0;
            For[i = 0, i < Length[lst], i++, res = res + 2^(8 * i) * lst[[i + 1]]];
            res]];

In[ ]:= a = TextToCode["Yes"]
Out[ ]=
        7 562 585

In[ ]:= b = TextToCode["No"]
Out[ ]=
        28 494

In[ ]:= e = 65 537
Out[ ]=
        65 537

In[ ]:= p = 101 983 718 092 569 165 349 421 591 207 648 443 539 793 987 356 013
        q = 143 222 416 378 923 537 212 836 637 253 671 311 561 082 782 479 067
        n = p * q
Out[ ]=
        101 983 718 092 569 165 349 421 591 207 648 443 539 793 987 356 013
Out[ ]=
        143 222 416 378 923 537 212 836 637 253 671 311 561 082 782 479 067
Out[ ]=
        14 606 354 536 524 698 706 200 717 660 113 257 376 264 705 905 172 858 159 450 406 841 609 831 
          905 150 296 476 906 084 608 849 079 871

In[ ]:= PowerMod[a, e, n]
Out[ ]=
        14 107 168 815 324 318 090 835 317 276 893 095 047 469 496 348 068 905 430 075 293 831 119 265 
          721 969 070 956 633 661 344 299 377 669

In[ ]:= PowerMod[b, e, n]
Out[ ]=
        11 774 244 636 341 304 750 761 336 269 690 742 471 400 677 163 271 259 631 947 941 872 681 238 
          267 252 125 222 308 976 903 173 383 755
```

Eve can simply test both messages by computing the e-th power mod N. The answer is thus “Yes”.

One needs to make the encryption probabilistic. So, let us set up RSA together with a so-called padding as described in steps 1 to 4 in Exercise 3.43 in [HPS], which is one way to make it into a probabilistic encryption. Say that b=24 (following the notation in Exercise 3.43) and that you receive the following ciphertext.

c=9681327024234897251183230455720571191065824040151988283717305498762980018949294418542546047469988823

b) Give a short explanation why the padding protects against attacks such as the one Eve can use in a).

Since Alice chooses a random string of b bits and concatenates it with m’ xor r, Eve can no longer just go through the possible messages since they will not as easily match. Alice essentially greatly increases the number of possible messages to try and power. The chance to hit the exact message that was sent thus decreases greatly.

c) Find the plaintext m’ (following the notation in Exercise 3.43) using our private key, and decode it using the function “CodeToText” given in the Lab introduction file.

We start as if we would normally decrypt.

```
In[ ]:= d = PowerMod[e, -1, (p - 1) * (q - 1)]
Out[ ]=
7 924 209 767 705 809 886 002 815 453 947 035 506 860 118 993 216 232 869 297 466 794 793 749 ⋱
  672 647 601 339 892 018 655 020 485 353
```

```
In[ ]:= m = PowerMod[
    9 681 327 024 234 897 251 183 230 455 720 571 191 065 824 040 151 988 283 717 305 498 762 ⋱
      980 018 949 294 418 542 546 047 469 988 823, d, n]
Out[ ]=
272 822 270 903 355
```

Now we convert m to binary.

```
In[ ]:= L = IntegerDigits[m, 2]
Out[ ]=
{1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1,
 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1}
```

```
In[ ]:= Length[L]
Out[ ]=
48
```

```
In[ ]:= M = L[[1 ;; 24]]
        P = L[[25 ;; 48]]
Out[ ]=
        {1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0}
Out[ ]=
        {1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1}
```

Now we invert XOR. The inverse of XOR is XOR.

```
In[ ]:= R = {};
        For[i = 1, i < 25, i++, R = Append[R, Mod[M[[i]] + P[[i]], 2] ]]

In[ ]:= R
Out[ ]=
        {0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1}

In[ ]:= mm = FromDigits[R, 2]
Out[ ]=
        7 562 585

In[17]:= CodeToText = Function[x, Module[{nmb, lst, tmp}, nmb = x;
              lst = {}; tmp = 0;
              While[nmb ≠ 0, tmp = Mod[nmb, 2^8];
               lst = Append[lst, tmp];
               nmb = (nmb - tmp) / 2^8];
              FromCharacterCode[lst]]];

In[ ]:= CodeToText[mm]
Out[ ]=
        Yes
```

Exercise 7

The RSA public key cryptosystem is set up by Alice in the following way:
n=193990039248117510110146289354346078444703425399126390896263687640955720804305868
251686436594206833697384507892252298116836513373204804988523790747917506684070650077
877388271652545753152731863768342998535177612777139380513550161863220306035637289497
971286889223191828141474758368917643301805545585847158202870859030856164655492881248
882185901425646755788062676650156864146153235735606795622976397068676400823051883389
6971726760554427
e=65537

The following ciphertext is sent by Bob to Alice:
c=829104585720293278626470138282411210590533811158703130580756826705756640524672985
376810471616350668463680489398108960669984158688828010844108708408493717783601465735
765204176314915216787694438820518397209550097224443643191484791651143362192151658048
948204027173310314513331591611598291620931752367747278198047722865730004108846746722
796203320711033035573007658737469411857835505566554339107332759317185845826056144
437100913389492

Try if "FactorInteger" can factor "n" (do not wait too long).

Use Pollard’s p-1 method to find the plaintext m (there is no padding in this exercise) and decode it using “CodeToText” given in the Lab introduction file.

```
In[1]:= n =
   1 939 900 392 481 175 101 101 462 893 543 460 784 447 034 253 991 263 908 962 636 876 409 ⋱
    557 208 043 058 682 516 864 365 942 068 336 973 845 078 922 522 981 168 365 133 732 048 ⋱
    049 885 237 907 479 175 066 840 706 500 787 738 827 165 254 575 315 273 186 376 834 299 ⋱
    853 517 761 277 713 938 051 355 016 186 322 030 603 563 728 949 971 286 889 223 191 828 ⋱
    141 474 758 368 917 643 301 805 545 585 847 158 202 870 859 030 856 164 655 492 881 248 ⋱
    821 859 014 256 467 557 880 626 766 501 568 641 461 532 357 356 067 956 229 763 970 686 ⋱
    764 008 230 518 833 896 971 726 760 554 427;

In[∘]:= FactorInteger[n]
Out[∘]=
        $Aborted
```

As expected the computation does not finish in reasonable time.
We therefore implement Pollard’s p-1 method.

```
In[8]:= Pollard = Function[N, Module[{a, j, d, success, save}, success = 0;
          a = 1;
          While[success == 0,
           a = a + 1;
           Print[a];
           save = a;
           For[j = 2, j < 4 000 000, j ++,
            a = PowerMod[a, j, N];
            d = GCD[a - 1, N];
            If[1 < d && d < N, success = 1; Return[d]; Break]
           ];
           a  =  save]]];

In[9]:= Pollard[n]

        2

Out[9]= Return[
   5 702 390 391 884 151 697 170 166 055 017 703 875 988 089 402 732 681 466 062 961 247 597 338 ⋱
    555 541 330 278 273 203 350 004 828 709 560 664 021 773 139 845 230 558 750 182 503 365 405 ⋱
    245 690 138 838 075 398 557 115 665 510 418 749 762 874 798 929 497 190 248 597 214 269 740 ⋱
    839 823 929 694 745 316 711]

In[14]:= p =
   5 702 390 391 884 151 697 170 166 055 017 703 875 988 089 402 732 681 466 062 961 247 597 ⋱
    338 555 541 330 278 273 203 350 004 828 709 560 664 021 773 139 845 230 558 750 182 503 ⋱
    365 405 245 690 138 838 075 398 557 115 665 510 418 749 762 874 798 929 497 190 248 597 ⋱
    214 269 740 839 823 929 694 745 316 711;
        q = n / p;
```

Using Pollard, we were able to factorize n. The rest is a simple decryption process.

```
In[16]:= d = PowerMod[65 537, -1, (p - 1) * (q - 1)]
Out[16]=
1 129 568 623 487 100 768 010 939 247 760 990 081 866 476 557 769 818 911 911 182 779 813 923 ⋱
  625 068 757 532 134 917 086 053 912 596 048 932 695 374 557 875 465 248 361 511 034 768 629 ⋱
  485 081 515 980 281 760 047 008 202 403 243 716 576 588 012 971 909 994 756 759 029 135 944 ⋱
  098 358 949 556 159 849 639 863 319 857 794 323 365 314 078 225 900 908 741 996 923 283 109 ⋱
  065 373 588 377 636 903 411 452 552 847 137 913 254 540 311 252 647 671 569 445 591 179 278 ⋱
  340 862 186 168 041 758 994 307 780 563 043 935 941 071 843 230 655 967 528 068 480 520 035 ⋱
  675 828 087 751 753
```

```
In[18]:= c =
    829 104 585 720 293 278 626 470 138 282 411 210 590 533 811 158 703 130 580 756 826 705 756 ⋱
      640 524 672 985 376 810 471 616 350 668 463 680 489 398 108 960 669 984 158 688 828 010 ⋱
      844 108 708 408 493 717 783 601 465 735 765 204 176 314 915 216 787 694 438 820 518 397 ⋱
      209 550 097 224 443 643 191 484 791 651 143 362 192 151 658 048 948 204 027 173 310 314 ⋱
      513 331 591 611 598 291 620 931 752 367 747 278 198 047 722 865 730 004 108 846 746 722 ⋱
      796 203 320 711 033 035 573 007 658 737 469 411 857 835 505 566 554 339 107 332 759 317 ⋱
      185 845 826 056 144 437 100 913 389 492;
  m = PowerMod[c, d, n]
  CodeToText[m]
Out[19]=
23 066 130 802 521 454 763 034 430 089 971 903 303 054 421 741 896 615 414 067 473 174 409 510 ⋱
  964 946 120 894 080 328
Out[20]=
He hath been bitten by the Tarantula.
```

Exercise 8

In this exercise you should use index calculus to solve the following DLP:
g^x=h (mod p)
where
p=348321033287
g=5
h=270836118207

The following simple function computes random powers g^i (mod p) and returns the first it finds that is B-smooth, where B is the b:th prime number.

```
In[76]:= GuessSmoothPowers = Function[{g, p, b},
           Module[{res, Lp, st, i, gi, git, j, gie, Lgie}, Lp = Map[Prime, Range[b]];
             st = 0;
             While[st == 0, i = RandomInteger[{2, p - 2}];
               gi = PowerMod[g, i, p];
               git = gi;
               Lgie = {};
               For[j = 0, j < Length[Lp], j++, gie = IntegerExponent[git, Lp[[j + 1]]];
                 Lgie = Append[Lgie, gie];
                 git = git / Lp[[j + 1]] ^ gie];
               If[git == 1, st = 1;
                 res = {Lgie, i}]];
             res]];

In[26]:= R1 = GuessSmoothPowers[2, 853, 4]
Out[26]=
         {{1, 2, 2, 0}, 79}

In[24]:= LinearSolve[{{1, 2}, {1, 3}}, {4, 7}, Modulus → 11]
Out[24]=
         {9, 3}

In[137]:=
         Smoothen = Function[{g, h, p, b},
           Module[{Lp, st, gh, k, ght, Lghe, j, ghe}, Lp = Map[Prime, Range[b]];
             st = 0;
             gh = h;
             For[k = 0, k < p & st == 0, ght = gh;
               Lghe = {};
               For[j = 0, j < Length[Lp], j++, ghe = IntegerExponent[ght, Lp[[j + 1]]];
                 Lghe = Append[Lghe, ghe];
                 ght = ght / Lp[[j + 1]] ^ ghe];
               If[ght == 1, st = 1, gh = Mod[gh * PowerMod[g, -1, p], p]; k++]];
             Print["k=", k];
             FactorInteger[gh]]];
```

(a) What is the theoretically suggested value for B that should make the hunt with GuessSmoothPowers the most efficient? Time how long it takes to find a “smooth power” for this value of B.

According to the book, one should take the following B.

```
In[27]:= p = 348 321 033 287
         B = Exp[Sqrt[Log[p] * Log[Log[p]]]]
Out[27]=
         348 321 033 287
Out[28]=
```

$$e^{\sqrt{\text{Log}[348\,321\,033\,287]\ \text{Log}[\text{Log}[348\,321\,033\,287]]}}$$

```
In[29]:= N[B]
Out[29]=
         11 345.2

In[31]:= {t, g} = Timing[GuessSmoothPowers[5, p, B]];

In[32]:= t
Out[32]=
         3.00284
```

It takes about 3 seconds.

(b) Solve the DLP using index calculus. (You might want to use a smaller B than the theoretical one suggested in (a) to decrease the number of times you need to use the function GuessSmoothPowers.)

We will use B=11.

```
In[116]:=
         B = 11; g = 5; h = 270 836 118 207; p = 348 321 033 287; L = {};
         For[i = 0, i < 11, i++, L = Append[L, GuessSmoothPowers[g, p, B]]];

In[118]:=
         L
Out[118]=
         {{{11, 3, 0, 0, 1, 0, 1, 0, 1, 0, 0}, 23 894 585 919},
          {{0, 2, 2, 0, 0, 1, 2, 2, 0, 0, 0}, 29 489 495 337},
          {{0, 0, 0, 0, 0, 1, 4, 0, 0, 2, 0}, 306 934 516 471},
          {{0, 5, 0, 1, 0, 0, 2, 1, 2, 1, 0}, 87 677 136 338},
          {{5, 1, 7, 1, 1, 0, 1, 0, 0, 1, 0}, 322 983 501 315},
          {{4, 1, 1, 0, 1, 2, 1, 0, 1, 1, 1}, 115 086 362 691},
          {{2, 2, 0, 1, 0, 0, 0, 0, 0, 6, 0}, 313 663 473 588},
          {{0, 1, 2, 0, 1, 1, 4, 2, 0, 0, 0}, 140 292 197 647},
          {{0, 1, 3, 2, 2, 0, 0, 0, 1, 2, 0}, 172 917 914 743},
          {{0, 5, 1, 0, 0, 1, 5, 0, 0, 0, 0}, 109 439 509 201},
          {{5, 2, 1, 0, 0, 0, 0, 2, 2, 0, 1}, 72 878 891 123}}
```

P gives us the first 11 primes.

```
In[90]:= P = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31};
         FactorInteger[p - 1]
Out[91]=
         {{2, 1}, {271, 1}, {642 658 733, 1}}
```

We now construct the system of linear equations.

```
In[119]:=
         M = {}; K = {};
         For[i = 1, i < 12, i++, M = Append[M, L[[i]][[1]]];
           K = Append[K, L[[i]][[2]]]];
```

We solve for in every prime factor modulus.

```
In[126]:=
        a = LinearSolve[M, K, Modulus → 2]
        b = LinearSolve[M, K, Modulus → 271]
        c = LinearSolve[M, K, Modulus → 642 658 733]
Out[126]=
        {0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0}
Out[127]=
        {52, 154, 1, 116, 121, 54, 177, 43, 229, 266, 107}
Out[128]=
        {296 065 870, 397 418 791, 1, 296 517 738, 556 500 513, 505 269 287,
         53 159 256, 545 701 584, 198 841 950, 155 526 626, 350 152 691}
In[131]:=
        Logs = {};
        For[i = 1, i < 12, i++, Logs =
           Append[Logs, ChineseRemainder[{a⟦i⟧, b⟦i⟧, c⟦i⟧}, {2, 271, 642 658 733}]]];
```

We test quickly whether everything worked.

```
In[154]:=
        Logs
        A = {};
        For[i = 1, i < 12, i++, A = Append[A, PowerMod[g, Logs⟦i⟧, p]]];
        A
Out[154]=
        {82 556 383 694, 200 264 284 754, 1, 63 277 073 572, 132 301 540 778, 54 488 602 859,
         89 382 723 143, 46 174 471 627, 34 902 413 532, 121 618 027 163, 142 377 732 684}
Out[157]=
        {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31}
In[178]:=
        R = Smoothen[g, h, p, B]
        k=323 648
Out[178]=
        {{2, 5}, {3, 6}, {7, 1}, {11, 3}, {19, 2}}
```

Now we can finally find x by using the relevant prime factors.

```
In[179]:=
        k = 323 648;
        S = Logs⟦{1, 2, 4, 5, 8}⟧;
        For[i = 1, i < 6, i++, R⟦i⟧ = R⟦i⟧⟦2⟧];
In[184]:=
        x = Mod[k + R.S, p - 1];
        x
Out[185]=
        76 972 390 086
```

One last quick test.

```
In[186]:=
        PowerMod[g, x, p]
Out[186]=
        270 836 118 207
```