# Factorization algorithms

We consider the problem of getting a nontrivial factor of a composite number. Factorization algorithms appeal recursively to the solution of this problem and combined with primality tests give the full prime factorization of a number.

**Fermat's factorization.** It is an extremely simple method essentially based on the relation $x^2 - y^2 = (x-y)(x+y)$. If we can find $y$ such that $n + y^2 = x^2$ then $(x-y)|n$.

The following program uses this technique to obtain a nontrivial factor. Some lines at the beginning are included to detect primes and even numbers.

```
def fermat_factor(n):
    if Mod(n,2)==0:
        return 2
    if is_prime(n):
        return n
    y = 0
    while( is_square(n+y^2)==False ):
        y += 1
    return sqrt(n+y^2)-y
```

If $n = pq$ with $p$ and $q$ odd primes, as in RSA, then $n = x^2 - y^2$ with $x = (p+q)/2$, $y = (p-q)/2$. If $p$ and $q$ are very close then $y$ is small and this simple method gives the factorization even for gigantic numbers.

For instance

```
p = random_prime(10^101,True)
q = next_prime(p)
n = p*q
print 'The number is', n
print 'It has', n.ndigits(), 'digits'

print '\nA factor is ', fermat_factor(p*q)
```

can factor in no time a 200 digits number of this form. The moral of the story is that in RSA close prime numbers has to be avoided.

```
The number is
659230925587256723667156710893739926206106725832901190192513650209261568\
920507606065172489394540316660737278499039707243803129856565681278595584\
2884851162188556539495554272669866793569293859644799976733
It has 202 digits

A factor is
811930369913120520211362870245508687326172438342983987822130172511668081\
44382954024225441452897752819
```

**Pollard's $p-1$ algorithm.** It is not useful for all numbers but it allows to factorize some extremely large special numbers. It computes $P(B) = \gcd(a^{B!} - 1, n)$ for increasing values of $B$. Of course, if $1 < P(B) < n$ for some $B$, we have got a nontrivial factor. Actually $B!$ is a simplification of the original algorithm, a slightly better choice of the exponent is $\operatorname{lcm}(1, 2, 3, \ldots, B)$. We shall take initially $a = 2$.

The theory suggests that this is a good algorithm if there are prime factors $p$ such that the prime factorization of $p-1$ only contains small prime powers. Here 'good' means that the value of $B$ is reasonably small.

This function computes the values of $P(B)$ for $B <$ `b` and return a nontrivial factor if it finds it.

```
def pollard_p(n,b):
    a = 2
    for i in range(1,b+1):
        a = Mod(a,n)^i
        d = gcd (a-1,n)
        if (d!=1) and (d!=n):
            return d
```

Note that $a_B = a^{B!}$ is computed by the recurrence $a_B = (a_{B-1})^B$ and, of course, we work modulo $n$, otherwise the size of $a_B$ would be unmanageable for a computer.

With `pollard_p(10403,10)` we get the factor `101` and `None` if `10` is substituted by a smaller number.

A slight variation tries bigger and bigger values of $B$ up to getting a nontrivial factor

```
def pollard_p_auto(n):
    a = 2
    i = 0
    d = n
    if is_prime(n):
        return d
    while (d==1) or (d==n):
        i += 1
        a = Mod(a,n)^i
        d = gcd (a-1,n)
    return d
```

One has to be careful with this program because for instance `pollard_p_auto(65)` enters into an infinite loop because $2^{B!} = 2^{12k}$ for $B > 3$ and $2^{12} \equiv 1 \pmod{65}$.

We avoid this problem changing the basis and starting up if at some point $a^{B!}$ becomes 1 modulo $n$.

```
def pollard_p_auto2(n):
    aa =2
    a = aa
    i = 0
    d = n
    if is_prime(n):
        return d
    while (d==1) or (d==n):
        i += 1
        a = Mod(a,n)^i
        d = gcd (a-1,n)
        if a == 1:
            aa += 1
            a = aa
            i = 0
    return d
```

It is interesting to check numerically the performance of the algorithm for $n = pq$ in terms of the factorization of the $p-1$ where $p$ is the output of `pollard_p_auto2(n)`. To this end we consider

```
k = 7
for i in range(20):
    p = random_prime(10^k, True)
    q = random_prime(10^k, True)
    t = cputime()
    f = pollard_p_auto2(p*q)
    dt = cputime(t)
    print factor(f-1), ' Time:', dt
```

that prints the factorization of $p-1$ and the interval of time `dt` required by `pollard_p_auto2(n)` to get $p$.

```
2^2 * 3^2 * 139 * 347  Time: 0.043994
2^2 * 3 * 245261  Time: 30.766322
2^2 * 3^7 * 5 * 109  Time: 0.014998
2 * 17 * 19 * 8923  Time: 1.134828
2^4 * 3^2 * 23 * 1423  Time: 0.173974
2^4 * 5 * 157 * 503  Time: 0.061991
2^2 * 5 * 13 * 43 * 401  Time: 0.049991
2 * 7 * 11 * 4597  Time: 0.555916
2 * 3^2 * 31 * 4451  Time: 0.563914
2 * 3^2 * 13^2 * 19 * 79  Time: 0.010998
2 * 17^2 * 17159  Time: 2.112679
2^2 * 3 * 7 * 101149  Time: 12.303129
2^3 * 3^2 * 19 * 37 * 41  Time: 0.00699899999995
2^3 * 11 * 19 * 1993  Time: 0.241964
2 * 3 * 7 * 11 * 15199  Time: 1.845719
2 * 3^3 * 5 * 27743  Time: 3.366488
```

```
2 * 3 * 11 * 151 * 647  Time: 0.077988
2^2 * 31 * 157 * 199  Time: 0.024996
2^3 * 17 * 19 * 29 * 113  Time: 0.0149980000001
2 * 5 * 131113  Time: 16.300522
```

Note that the biggest number in this list corresponds to $p-1 = 2^2 \cdot 3 \cdot 245261$ having the unbalanced prime factor 245261. On the other hand, the best performance is for $p-1 = 2 \cdot 3^2 \cdot 13^2 \cdot 19 \cdot 79$ with many small small prime power factors.

Running the program with higher values of `k` (this is typically like one half of the number of digits) we realize that Pollard's $p-1$ algorithm is not convenient as a single method for general numbers.

For instance a table for `k=10` included some extreme values like

```
2^2 * 5 * 17 * 27685279  Time: 3471.164302
2 * 347 * 6327889  Time: 793.400386
```