The k-ary LR method

It turns out that the LR method of repeated squaring can be generalized. Instead of breaking the exponent into bits of its base-2 representation, we can break it into larger pieces, and save some computations this way.

I'll present the k-ary LR method that breaks the exponent into its "digits" in base $m = 2^k$ for some integer `k`. The exponent can be written as:

$$b = t_i m^i + t_{i-1} m^{i-1} + \cdots t_0 m^0$$

Where t_i are the digits of `b` in base `m`. a^b is then:

$$a^{t_i m^i} \cdot a^{t_{i-1} m^{i-1}} \cdot \ldots \cdot a^{t_0}$$

We compute this iteratively as follows [6]:

Raise a^{t_0} to the `m`-th power and multiply by a^{t_1}. We get $r_1 = a^{t_0 m + t_1}$. Next, raise r_1 to the `m`-th power and multiply by a^{t_2}, obtaining $r_2 = a^{t_0 m^2 + t_1 m + t_2}$. If we continue with this, we'll eventually get a^b.

This translates into the following code:

```
def modexp_lr_k_ary(a, b, n, k=5):
    """ Compute a ** b (mod n)

        K-ary LR method, with a customizable 'k'.
    """
    base = 2 << (k - 1)

    # Precompute the table of exponents
    table = [1] * base
    for i in xrange(1, base):
        table[i] = table[i - 1] * a % n

    # Just like the binary LR method, just with a
    # different base
    #
    r = 1
    for digit in reversed(_digits_of_n(b, base)):
        for i in xrange(k):
            r = r * r % n
        if digit:
            r = r * table[digit] % n

    return r
```

Note that we save some time by pre-computing the powers of `a` for exponents that can be digits in base `m`. Also, the `_digits_of_n` is the following generalization of `_bits_of_n`:

```python
def _digits_of_n(n, b):
    """ Return the list of the digits in the base 'b'
        representation of n, from LSB to MSB
    """
    digits = []

    while n:
        digits.append(n % b)
        n //= b

    return digits
```

Performance of the k-ary method

In my tests, the k-ary LR method with `k = 5` is about 25% faster than the binary LR method, and is within 20% of the built-in `pow` function.

Experimenting with the value of `k` affects these results, but 5 seems to be a good value that produce the best performance in most cases. This is probably why it's also used as the value of `k` in the implementation of `pow`.