- Timing Tests
- Expression Disassembly
  - Multiplication
  - math.pow()
  - Exponentiation
- BINARY_MULTIPLY versus BINARY_POWER
  - BINARY_MULTIPLY
  - BINARY_POWER
- Charting Performance Differences
  - Generating Functions
    - math.pow() and Exponentiation
    - Chained Multiplication
  - Finding the Crossover
  - Charting the Performance
- More Performance Testing
- Conclusions

Recently, I was writing an algorithm to solve a coding challenge that involved finding a point in a Cartesian plane that had the minimum distance from all of the other points. In Python, the distance function would be expressed as `math.sqrt(dx ** 2 + dy ** 2)`. However, there are several different ways to express each term: `dx ** 2`, `math.pow(dx, 2)`, and `dx * dx`. Interestingly, these all perform differently, and I wanted to understand how and why.

Timing Tests

Python provides a module called `timeit` to test performance, which makes testing these timings rather simple. With `x` set to `2`, we can run timing tests on all three of our options above:

Expression	**Timing (100k iterations)**
`x * x`	3.87 ms
`x ** 2`	80.97 ms

`math.pow(x, 2)`	83.60 ms

Expression Disassembly

Python also provides a model called `dis` that disassembles code so we can see what each of these expressions are doing under the hood, which helps in understanding the performance differences.

Multiplication

Using `dis.dis(lambda x: x * x)`, we can see that the following code gets executed:

```
0 LOAD_FAST                0 (x)
2 LOAD_FAST                0 (x)
4 BINARY_MULTIPLY
6 RETURN_VALUE
```

The program loads `x` twice, runs `BINARY_MULTIPLY`, and `return`s the value.

math.pow()

Using `dis.dis(lambda x: math.pow(x, 2))`, we can see the following code gets executed:

```
0 LOAD_GLOBAL              0 (math)
2 LOAD_ATTR                1 (pow)
4 LOAD_FAST                0 (x)
6 LOAD_CONST               1 (2)
8 CALL_FUNCTION            2
10 RETURN_VALUE
```

The `math` module loads from the global space, and then the `pow` attribute loads. Next, both arguments are loaded and the `pow` function is called, which `return`s the value.

Exponentiation

Using `dis.dis(lambda x: x ** 2)`, we can see that the following code gets executed:

```
0 LOAD_FAST                0 (x)
2 LOAD_CONST               1 (2)
4 BINARY_POWER
6 RETURN_VALUE
```

The program loads `x`, loads `2`, runs `BINARY_POWER`, and `return`s the value.

BINARY_MULTIPLY versus BINARY_POWER

Using the `math.pow()` functions as a point of comparison, both multiplication and exponentiation differ in only one part of their bytecode: calling `BINARY_MULTIPLY` versus calling `BINARY_POWER`.

BINARY_MULTIPLY

This function is located here in the Python source code. It does a few interesting things:

```
long_mul(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;

    CHECK_BINOP(a, b);

    /* fast path for single-digit multiplication */
    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        stwodigits v = (stwodigits)(MEDIUM_VALUE(a)) *
MEDIUM_VALUE(b);
        return PyLong_FromLongLong((long long)v);
    }

    z = k_mul(a, b);
    /* Negate if exactly one of the inputs is negative. */
```

```
    if (((Py_SIZE(a) ^ Py_SIZE(b)) < 0) && z) {
        _PyLong_Negate(&z);
        if (z == NULL)
            return NULL;
    }
    return (PyObject *)z;
}
```

For small numbers, this uses binary multiplication. For larger values, the function uses Karatsuba multiplication, which is a fast multiplication algorithm for larger numbers.

We can see how this function gets called in `ceval.c`:

```
case TARGET(BINARY_MULTIPLY): {
    PyObject *right = POP();
    PyObject *left = TOP();
    PyObject *res = PyNumber_Multiply(left, right);
    Py_DECREF(left);
    Py_DECREF(right);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

BINARY_POWER

This function is located here in the Python source code. It also does several interesting things:

The source code is too long to fully include, which partially explains the detrimental performance. Here are some interesting snippets:

```
if (Py_SIZE(b) < 0) {  /* if exponent is negative */
    if (c) {
        PyErr_SetString(PyExc_ValueError, "pow() 2nd argument "
                        "cannot be negative when 3rd argument
```

```
specified");
        goto Error;
    }
    else {
        /* else return a float.  This works because we know
           that this calls float_pow() which converts its
           arguments to double. */
        Py_DECREF(a);
        Py_DECREF(b);
        return PyFloat_Type.tp_as_number->nb_power(v, w, x);
    }
}
```

After creating some pointers, the function checks if the `power` given is a float or is negative, where it either errors or calls a different function to handle exponentiation.

If neither cases hit, it checks for a third argument, which is always `None` according to `ceval.c`[1]:

```
case TARGET(BINARY_POWER): {
    PyObject *exp = POP();
    PyObject *base = TOP();
    PyObject *res = PyNumber_Power(base, exp, Py_None);
    Py_DECREF(base);
    Py_DECREF(exp);
    SET_TOP(res);
    if (res == NULL)
        goto error;
    DISPATCH();
}
```

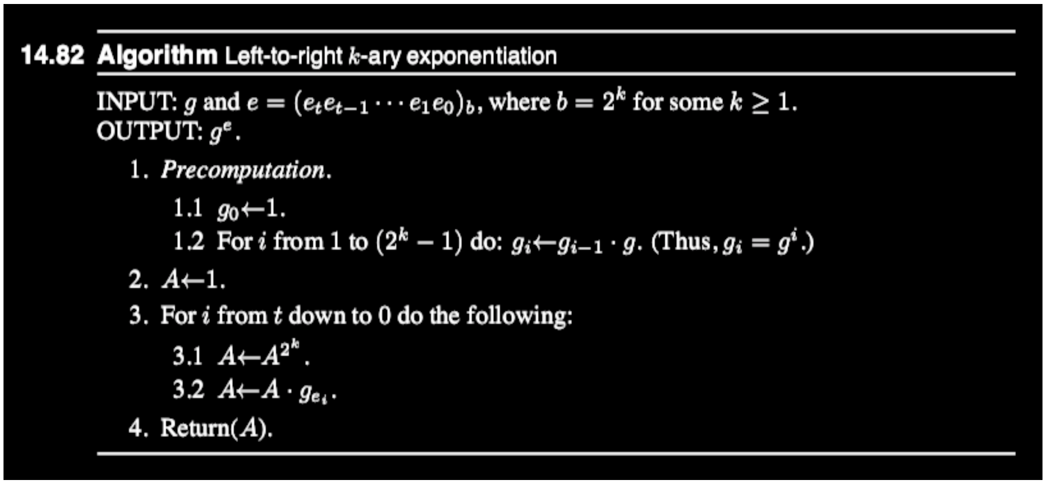
Finally, the function defines two routines: `REDUCE` for modular reduction and `MULT` for multiplication and reduction. The multiplication function uses `long_mul` for both values, which is the same function used in `BINARY_MULTIPLY`.

```
#define REDUCE(X)                                   \
```

```
do {                                                \
    if (c != NULL) {                                \
        if (l_divmod(X, c, NULL, &temp) < 0)        \
            goto Error;                             \
        Py_XDECREF(X);                              \
        X = temp;                                   \
        temp = NULL;                                \
    }                                               \
} while(0)

#define MULT(X, Y, result)                          \
do {                                                \
    temp = (PyLongObject *)long_mul(X, Y);          \
    if (temp == NULL)                               \
        goto Error;                                 \
    Py_XDECREF(result);                             \
    result = temp;                                  \
    temp = NULL;                                    \
    REDUCE(result);                                 \
} while(0)
```

After that, the function uses left-to-right k-ary exponentiation defined in chapter 14.6[2] of the Handbook of Applied Cryptography:



14.82 Algorithm Left-to-right k-ary exponentiation

```
INPUT: $g$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$, where $b = 2^k$ for some $k \geq 1$.
OUTPUT: $g^e$.
  1. Precomputation.
      1.1 $g_0 \leftarrow 1$.
      1.2 For $i$ from 1 to $(2^k - 1)$ do: $g_i \leftarrow g_{i-1} \cdot g$. (Thus, $g_i = g^i$.)
  2. $A \leftarrow 1$.
  3. For $i$ from $t$ down to 0 do the following:
      3.1 $A \leftarrow A^{2^k}$.
      3.2 $A \leftarrow A \cdot g_{e_i}$.
  4. Return($A$).
```

Charting Performance Differences

We can use the `timeit` library above to profile code at different values and see how the performance changes over time.

Generating Functions

To test the performance at different `power` values, we need to generate some functions.

math.pow() and Exponentiation

Since both of these are already in the Python source, all we need to do is define a function for exponentiation we can call from inside a `timeit` call:

```
exponent = lambda base, power: base ** power
```

Chained Multiplication

Since this changes each time the `power` changes[3], we need to generate a new multiplication function each time the base changes. To do this, we can generate a string like `x*x*x` and call `eval()` on it to return a function:

```
def generate_mult_func(n):
    mult_steps = '*'.join(['q'] * n)
    func_string = f'lambda q: {mult_steps}'  # Keep this so we can
print later
    return eval(func_string), func_string
```

Thus, we can make a `multiply` function like so:

```
multiply, func_string = generate_mult_func(power)
```

If we call `generate_mult_func(4)`, `multiply` will be a lambda function that looks like this:

```
lambda q: q*q*q*q
```

Finding the Crossover

Using the code posted here, we can determine at what point `multiply` becomes less efficient than `exponent`.

Staring with these values:

```
base = 2
power = 2
```

We loop until the time it takes to execute `100,000` iterations of `multiply` is slower than executing `100,000` iterations of `exponent`. Initially, here are the timings, with `math.pow()` serving as a point of comparison:

```
Starting speeds:
Multiply time   11.83 ms
Exponent time   86.52 ms
math.pow time   73.90 ms
```

When running on repl.it, Python finds the crossover in 1.2s:

```
Crossover found in 1.2 s:
Base, power     2, 15
Multiply time   110.09 ms
Exponent time   108.20 ms
math.pow time   79.82 ms
Multiply func   lambda q: q*q*q*q*q*q*q*q*q*q*q*q*q*q*q
```

Thus, chaining multiplication together is faster until our expression gets to `2^14`; at `2^15` exponentiation becomes faster.

Charting the Performance

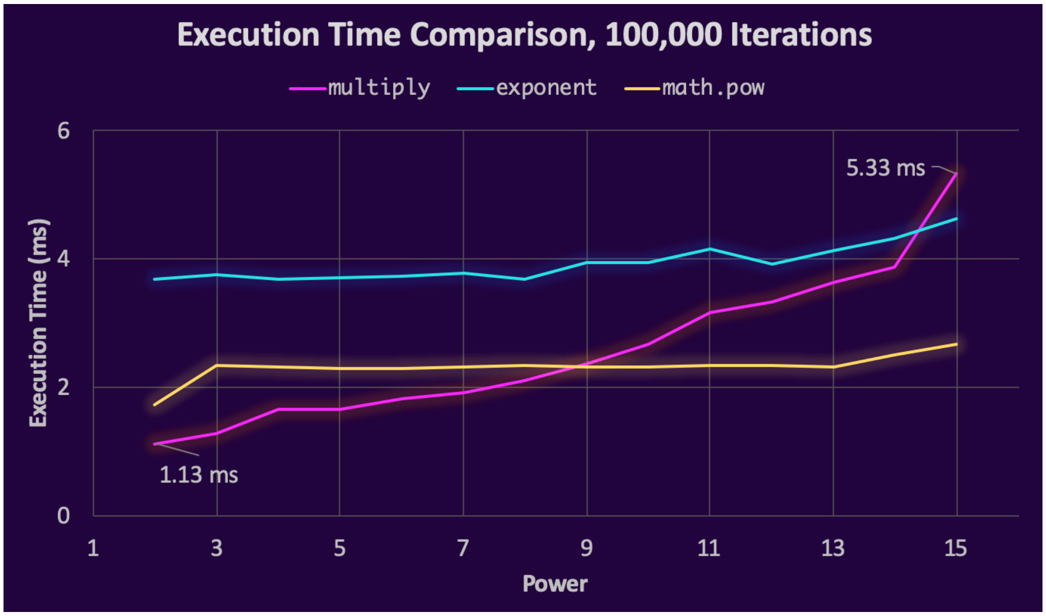
Using Pandas, we can keep track of the timing at each power:

```
     Power  multiply  exponent  math.pow
```

```
  2  0.011825  0.086524  0.073895
  3  0.022987  0.097911  0.075673
  4  0.016409  0.090745  0.025436
  5  0.068577  0.090413  0.023301
  6  0.019716  0.104905  0.107520
  7  0.072666  0.093392  0.084163
  8  0.031971  0.087344  0.072766
  9  0.034182  0.162763  0.042760
 10  0.076582  0.087033  0.090269
 11  0.105528  0.116346  0.024251
 12  0.087499  0.094689  0.078410
 13  0.040243  0.102694  0.029103
 14  0.098822  0.106432  0.080152
 15  0.110085  0.108199  0.079816
```

From here, it is very simple to generate a line graph:

```
plot = df.plot().get_figure()
plot.savefig('viz.png')
```

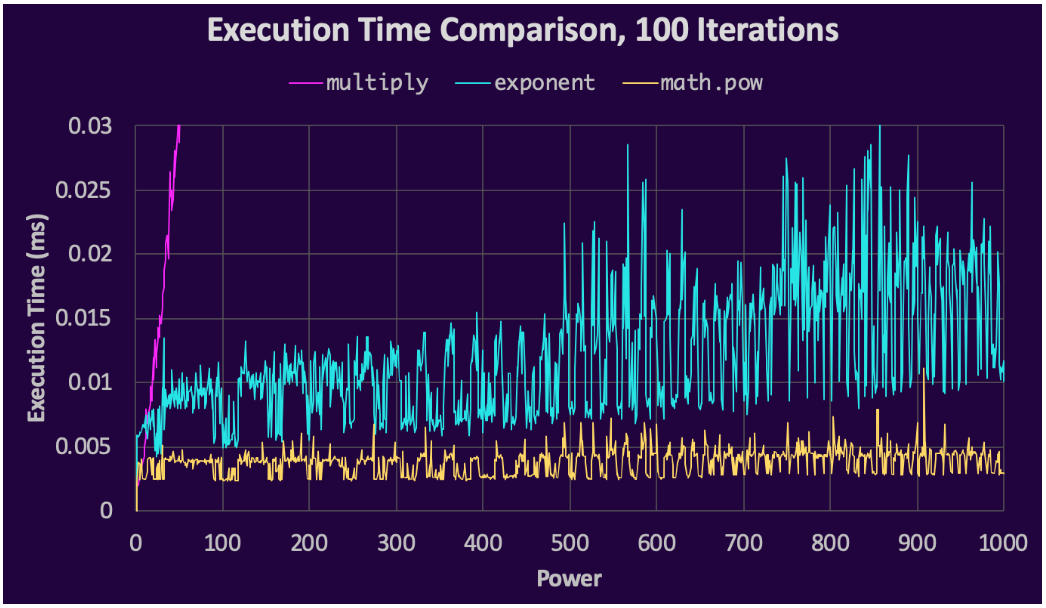


Interestingly, `math.pow()` and `exponent` mostly perform at the same rate, while our `multiply` functions vary wildly. Unsurprisingly, the longer the multiplication chain, the longer it takes to execute.

More Performance Testing

While the crossover is interesting, this doesn't show what happens at powers larger than 15. Going up through `1000`, we get the following trend:

When we zoom in so that `math.pow()` and `exponent` are more pronounced, we see the same performance trend continue:



While using `**` the time gradually increases, `math.pow()` generally has executes at around the same speed.

Conclusions

When writing algorithms that use small exponents, here proved less than `15`, it is faster to chain multiplication together than to use the `**` exponentiation operator. Additionally, `math.pow()` is more efficient than chained multiplication at powers larger than `10` and always more efficient than the `**` operator, so there is never a reason to use `**`.

Additionally, this is also true in JavaScript[4]. Thanks @julaincoleman for this comparison!

Discussion: <a href="https://www.reddit.com/r/Python/comments/bv1ez2/performance_of_various_python_exponentiation/“>r/Python, Hacker News | View as: PDF, Markdown

---

1. This is used as the `modulus` parameter in the sodlib `pow()` and `math.pow()` functions: `pow(2, 8, 10)` results in `2^8 % 10`, or `6`
2. According to the Python source, specifically section 14.82.
3. `x ** 2 == x * x`, `x ** 3 == x * x * x` and so on.
4. Except in Safari, where `Math.pow()` is the slowest.