Sitemap (/weblog/sitemap/) RSS Feed (/weblog/feeds/)

Keywords... OK

# Computing Large Binomial Coefficients Modulo Prime / Non-Prime (/weblog/2015/06/25/computing-large-binomial-coefficients-modulo-prime-non-prime/)

*Written on June 25, 2015 (2015-06-25T22:15:53+00:00)*
*Last update on June 28, 2015 (2015-06-28T21:41:03+00:00).*

Not rarely, in combinatoric problems it comes down to calculating the binomial coefficient (https://en.wikipedia.org/wiki/Binomial_coefficient) $\binom{n}{k}$ for very large $n$ and/or $k$ modulo a number $m$. In general, the binomial coefficient can be formulated with factorials (https://en.wikipedia.org/wiki/Factorial) as $\binom{n}{k} = \frac{n!}{k!(n-k)!}, 0 \leq k \leq n$. The problem here is that factorials grow extremely fast which makes this formula computationally unsuitable because of quick overflows. For that reason, many problems in that category require the calculation of $\binom{n}{k} \mod m$. In this post I want to discuss ways to calculate the binomial coefficients for cases in which $m$ is prime and when $m$ is non-prime.

## 1. First simple approaches for any $m$

The good news is that there are easy ways to compute the binomial coefficient for any modulo $m$ - the bad news is that they are not feasible for very large numbers. However, we will see later that we can break down very large numbers into sub-problems of much smaller numbers, in which case we can re-use techniques that we discuss now.

### Using Pascal's Triangle

The easiest (but also most limited) way to tackle this problem is to use a bottom-up dynamic programming approach by pre-computing Pascal's Triangle (https://en.wikipedia.org/?title=Pascal%27s_triangle). Pascal's Triangle can be seen as a table, where $\binom{n}{k}$ is the value at the $n$th row in the $k$th column. Pre-computing Pascal's Triangle is very easy, since the current value is simply the sum of two neighboring cells from the previous row. We know that $(a+b) \mod m = (a \mod m) + (b \mod m) \mod m$, thus we can calculate each cell of Pascal's Triangle with the modulo to avoid overflows.

**Code: Computing Pascal's Triangle, for any $m$**

```
def pascal_triangle(n,mod):
    # prepare first row in table
    pascal = []
    pascal.append([1,0])

    for i in range(1,n+1):
        # initialize new row in table
        pascal.append([])
        pascal[i] = [0]*(i+2)
        pascal[i][0] = 1

        for j in range(1,i+1):
            # calculate current value based on neighbor values from previous row in triangle
            pascal[i][j] = (pascal[i-1][j] + pascal[i-1][j-1])%mod

    return pascal

if __name__ == '__main__':
    n = 1009 # number of rows in pascal table
    mod = 123456 # not a prime

    # calculate pascal table with mod 123456
    pascal = pascal_triangle(n,mod)

    # (950 choose 100) mod 123456
    print(pascal[950][100]) # should be 24942
```

**pascal_triangle.py** **view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/pascal_triangle.py)** **(https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-pascal_triangle-py)** hosted with ❤ by **GitHub (https://github.com)**

Computing the triangle takes quadratic time $O(n^2)$, since we compute each possible $k$ for each row ($n$ rows in total). Once the triangle is computed we can simply lookup the binomial coefficient in constant time $O(1)$for different $n$ and $k$.
The big problem arises when looking at the space complexity of $O(n^2)$. For some problems $n$ might be small enough, so there

won't be an issue, but for very large $n$ we will not have enough memory to store the table, which means we have to find a better approach for those cases.

There is an easy upgrade in terms of space complexity. Each row from Pascal's Triangle can be calculated by only knowing the previous row. This means we only have to keep track of the current row, thus the space complexity is linear $O(n)$.

**Code: Computing single row in Pascal's Triangle, for any $m$**

```python
#!/usr/bin/env python3

"""
Calculating a row in Pascal's Triangle to determine Binomial Coefficients
"""

# Calculating row 'n' of Pascal's Triangle with modulo 'mod' only remembering last calculated row
# bottom-up dynamic programming approach
# runtime:      O(n**2)
# space:        O(n)
def pascal_triangle_row(n,mod):
    # initialize necessary space for row 'n'
    pascal = [0]*(n+1)
    pascal[0] = 1

    for i in range(1,n+1):
        # initialize auxiliary array
        tmp = [0]*(n+1)
        tmp[0] = 1

        for j in range(1,i+1):
            # calculate current value based on neighbor values from previous row in triangle
            tmp[j] = (pascal[j] + pascal[j-1])%mod

        pascal = tmp

```

**view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/pascal_triangle_row.py)**
**pascal_triangle_row.py (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-pascal_triangle_row-py)** hosted with ❤ by **GitHub (https://github.com)**

Even though this approach allows to calculate larger binomial coefficients due to better space complexity, it still comes with a downside. In the first approach we can calculate the binomial coefficient in $O(1)$, after we have pre-computed Pascal's Triangle in $O(n^2)$. However in this approach, we need to recalculate the triangle or compute further rows each time a smaller or bigger row $n$ is asked than the currently computed one. Consequently, this approach is expensive when we expect to calculate many different binomial coefficients that are not in the same row of Pascal's Triangle.

## 2. Advanced approaches when $m$ is prime

In most related problems we are asked to calculate $\binom{n}{k} \mod m$, with $m$ being a prime number. The choice of $m$ being a prime number is by far no coincidence, since prime number modulus create a field in which certain Theorems hold.

### Using Fermat's Little Theorem

We remember that the binomial coefficient can be written as $\frac{n!}{k!(n-k)!} = \frac{n\cdot(n-1)\cdot\cdots\cdot(n-k+1)}{k!}$. We can easily calculate the numerator of that equation, since $((a \mod m)\cdot(b \mod m)) \mod m = (a\cdot b) \mod m$. In order to divide the numerator by the denominator, we have to multiply the numerator with the multiplicative inverse of the denominator. Lucky for us, if $m$ is prime, we can easily apply Fermat's little theorem (https://en.wikipedia.org/wiki/Fermat's_little_theorem). Through this theorem we know that $a^{-1} \equiv a^{p-2} \mod p$, with $p$ being a prime number. Consequently, we can easily find the multiplicative inverse through modular exponentiation of $(k!)^{m-2}$.

**Code: Using Fermat's Theorem, for $m$ prime and $k < m$**

```python
#!/usr/bin/env python3

"""
Using Fermat's little theorem to calculate nCk mod m, for k < m and m is prime

Two versions:
1. Pre-Compute factorials and multiplicative inverses in O(n*logn) --> later lookup in O(1)
2. Compute directly --> no lookup --> each time O(n)
"""

# modular exponentiation: b^e % mod
def mod_exp(b,e,mod):
    r = 1
    while e > 0:
        if (e&1) == 1:
            r = (r*b)%mod
        b = (b*b)%mod
        e >>= 1

    return r

# Using Fermat's little theorem to compute nCk mod p
# Note: p must be prime and k < p
def fermat_binom(n,k,p):
    if k > n:
        return 0
```

**fermat_binom.py** **view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/fermat_binom.py)** **(https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-fermat_binom-py)** hosted with ❤ by **GitHub (https://github.com)**

Computing the factorial $n!$ takes $O(n)$ and computing one multiplicative inverse takes $O(\log_2 n)$. Note, that we implemented two versions here:

In the first version, we pre-computed the factorials and the corresponding inverses in an array at the cost of $O(n)$ memory. This pre-computation takes $O(n \log_2 n)$ time and gives us the advantage of further lookups of binomial coefficients within the same range at constant time $O(1)$.

In the second version, without pre-computed lookups, we re-compute the values each time and only calculate the multiplicative inverse of one denominator, thus having a linear time $O(n)$ in total each time we want to calculate a binomial coefficient.

Whether we use pre-computed tables or not highly depends on $n$'s expected size. In case we have very large $n$, pre-computing the factorial and inverse tables with Fermat's theorem (space complexity $O(n)$ and runtime of pre-computation $O(n \log_2 n)$) might still be too expensive. In all other cases, pre-computing is desired since it will save us a lot of time later on. In the following I will only show algorithms without pre-computation, since those are also working for very large $n$, but changing those algorithms to pre-computed versions can be done easily (analogous to the current example), since the core idea is still the same.

Note, that this code is only working as long as $k < m$, since otherwise $k! \mod m = 0$, which means there is no inverse defined. Luckily, with slight modifications we can work around this issue. First we compare the degree of $m$ in numerator and denominator, because the degree in the numerator has to be smaller or equal. Further, we cancel out any occurrence of $m$ in both, numerator and denominator.

**Code: Using Fermat's Theorem, for $m$ prime**

```
#!/usr/bin/env python3

"""
Using Fermat's little theorem to calculate nCk mod m, m is prime
Computation O(n)
"""

# modular exponentiation: b^e % mod
def mod_exp(b,e,mod):
    r = 1
    while e > 0:
        if (e&1) == 1:
            r = (r*b)%mod
        b = (b*b)%mod
        e >>= 1

    return r

# get degree of p in n! (exponent of p in the factorization of n!)
def fact_exp(n,p):
    e = 0
    u = p
    t = n
    while u <= t:
        e += t//u
        u *= p
```

**view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/fermat_binom_advanced.py)** **fermat_binom_advanced.py (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-fermat_binom_advanced-py)** hosted with ❤ by **GitHub (https://github.com)**

We are now able to determine the binomial coefficient in linear time $O(n)$ as long as the modulo is a prime number. We will see in the next sections how we can enhance the computation in terms of time complexity in some scenarios. Further, we will see how we can determine the binomial coefficients in case the modulo is not a prime number. The algorithms involving Fermat and Pascal are very crucial, since all further approaches that we are going to discuss in this post are relying on them to solve the core computation.

## Using Lucas' Theorem

Lucas' theorem (https://en.wikipedia.org/wiki/Lucas%27_theorem) gives us a great way to break down the computation of one large binomial coefficient into the computation of smaller binomial coefficients. In order to do so, we first have to convert the digits of $n$ and $k$ into their base $m$ representation $n = n_0 + n_1 \cdot m^1 + n_2 \cdot m^2 \ldots n_x \cdot m^x$ and $k = k_0 + k_1 \cdot m^1 + k_2 \cdot m^2 \ldots k_x \cdot m^x$. We can then define sub problems: $\binom{n}{k} = \prod_{i=0}^{x} \binom{n_i}{k_i} \pmod{m}$. These problems can then be solved using Fermat's theorem like we did before.

### Code: Using Lucas' + Fermat's Theorem, for $m$ prime

```
#!/usr/bin/env python3

"""
Using Lucas' and Fermat's little theorem to calculate nCk mod m, m is prime
"""

# modular exponentiation: b^e % mod
def mod_exp(b,e,mod):
    r = 1
    while e > 0:
        if (e&1) == 1:
            r = (r*b)%mod
        b = (b*b)%mod
        e >>= 1

    return r

# get degree of p in n! (exponent of p in the factorization of n!)
def fact_exp(n,p):
    e = 0
    u = p
    t = n
    while u <= t:
        e += t//u
        u *= p

```

**lucas_binom.py** **view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/lucas_binom.py)**

**(https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-lucas_binom-py)** hosted with ❤ by **GitHub (https://github.com)**

> A little side fact: Lucas' theorem gives an efficient way of determining whether our prime $m$ is a divisor of the binomial coefficient, without calculating the coefficient itself. All we have to do is compare the digits of $n$ and $k$ in their base $m$ representation $n = n_0 + n_1 \cdot m^1 + n_2 \cdot m^2 \ldots$ and $k = k_0 + k_1 \cdot m^1 + k_2 \cdot m^2 \ldots$ and if in at least one case $k_i > n_i$, then the product of all sub binomial coefficients would be 0, since $\binom{n}{k} = 0$, when $k > n$.

We used Lucas' theorem to first split the problem into smaller binomial coefficients that we could then use to find our solution. In order to solve the small coefficients, we used Fermat's theorem like in the previous section of this post. We could have also used Pascal's Triangle in order to compute these small coefficients. Also, in case $n$ and $k$ are smaller than $m$, Lucas' theorem would not have brought any benefits, because the smallest sub problem would then be the original problem itself. However, in that case Lucas' theorem does only bear the small overhead of bringing $n$ and $k$ in base $m$ representation, which is why in general it is a good approach to first try splitting the problem into smaller sub problems with Lucas' theorem, then using Pascal's Triangle or Fermat's theorem to solve these sub problems.

> Remember, in this example we did not use pre-computation, which makes it more efficient when computing only one single very large $n$. Whether we use pre-computed tables or not highly depends on the expected size of $n$ (which when using Lucas' theorem, is the biggest prime factor in $m$) and whether we expect to calculate many different coefficients. In case our sub problems have still very large $n$, pre-computing the factorial and inverse tables with Fermat's theorem (space complexity $O(n)$ and initial runtime $O(n \log_2 n)$) might still be too expensive to be feasible. In all other cases, pre-computing is desired since it will save us a lot of time later on.

All in all, now we have efficient ways of computing large binomial coefficients modulo a prime number.

## 3. $m$'s prime factorization is square-free (https://en.wikipedia.org/wiki/Square-free_integer)

What do we do in cases in which $m$ turns out not to be a prime? The answer lies in the prime factorization of $m$. The following method works when $m$'s prime factorization is square-free (https://en.wikipedia.org/wiki/Square-free_integer), e.g. $6 = 2 \cdot 3$, which means $6$ is square-free, whereas $24 = 2^3 \cdot 3$ is not square-free, since the prime $2$ occurs more than once.

Lets assume $m$'s prime factors are $m = p_0 \cdot p_1 \cdot \dots \cdot p_r$. For each prime factor $p_i$ of $m$, we can solve $\binom{n}{k} \mod p_i$, using the approaches we have discussed so far, which leaves us with $r$ results $a_0, a_1, \dots, a_r$. Through the Chinese Remainder Theorem (CRT) (https://en.wikipedia.org/wiki/Chinese_remainder_theorem) we know that we can find a number $x$, which solves the following congruences $x \equiv a_0 \mod p_0$, $x \equiv a_1 \mod p_1$, ..., $x \equiv a_r \mod p_r$.

Let's define $m_i = \frac{p_0 \cdot p_1 \cdot \dots \cdot p_r}{p_i}$. We know that $m_i$ and $p_i$ must be coprime, then $gcd(m_i, p_i) = 1$, meaning $1 = s_i \cdot m_i + t_i \cdot p_i$. Using the Extended Euclidean Algorithm (https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm), we can find such $s_i$ and $t_i$. Finally, we can find the solution that combines all the congruences to $x = \sum_{i=0}^{r}(a_i \cdot m_i \cdot s_i)(\mod \prod_{i=0}^{r} m_i)$

The CRT works as long as $p_0 \dots p_r$ are co-prime. We see, that by applying the CRT to those congruence systems, we can combine them into a single congruence modulo $m$, which solves our problem.

**Code: Using Lucas' + Fermat's + Chinese Remainder Theorem, for $m$ with square-free prime factors**

```
#!/usr/bin/env python3

"""
Using Lucas' and Fermat's little theorem to calculate nCk mod m, m's prime factorization is square-free
Also using Chinese Remainder Theorem to combine congruences of prime factors.
"""

# modular exponentiation: b^e % mod
def mod_exp(b,e,mod):
    r = 1
    while e > 0:
        if (e&1) == 1:
            r = (r*b)%mod
        b = (b*b)%mod
        e >>= 1

    return r

# get degree of p in n! (exponent of p in the factorization of n!)
def fact_exp(n,p):
    e = 0
    u = p
    t = n
    while u <= t:
        e += t//u
        u *= p
```

**lucas_crt_binom.py view raw (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c/raw/b0b2ab612779837adffc8b613e0a457c3507df37/lucas_crt_binom.py) (https://gist.github.com/fishi0x01/46ebb1d310df5f9b5d8c#file-lucas_crt_binom-py)** hosted with ❤ by **GitHub (https://github.com)**

> Finding the prime factorization of $m$ is non-trivial and no efficient algorithm is known so far that can solve this problem for large $m$. However, as long as $m$ has a reasonable size (and that is very likely in those combinatoric problems), finding the prime factorization is feasible. You can write your own algorithm or use some online calculators to determine the prime factors. Also remember that our current approach is only valid for $m$ whose prime factors are square-free.

Now we are also able to compute large binomial coefficients modulo numbers whose prime factorization is square-free.

## 4. $m$ has prime factors with multiplicity (https://en.wikipedia.org/wiki/Multiplicity_(mathematics)) higher than 1

Finally, lets have a look at all the remaining possible modulo numbers, namely those containing prime factors with a multiplicity (https://en.wikipedia.org/wiki/Multiplicity_(mathematics)) higher than 1. For that purpose we can use Andrew Granville's generalization of Lucas' Theorem (http://www.dms.umontreal.ca/~andrew/PDF/BinCoeff.pdf). This theorem also allows prime powers in the factorization of $m$. After applying the theorem, we could then use our other methods to calculate the sub problems and then like before use the Chinese Remainder Theorem to combine the sub solutions into one solution. With prime powers allowed, we can then factorize any modulus $m$, meaning we can find the binomial coefficients modulo any number $m$. I don't have an implementation ready yet for the generalization of Lucas' Theorem, but once I have some time I will try to add it to this post.

## 5. Quick Summary

We should do the following steps in order to compute large binomial coefficients $\binom{n}{k} \mod m$:

1. Find prime factors (and multiplicities) $p_0^{e_0}, \ldots p_r^{e_r}$ of $m$
2. Use (generalized) Lucas' Theorem to find all sub problems for each $\binom{n}{k} \mod p_i^{e_i}$
3. Solve sub problems $\binom{n}{k} \mod p_i^{e_i}$ with Fermat's little theorem or Pascal's Triangle
4. Use Chinese Remainder Theorem to combine sub results

Remember, if $p_i > n$, then there are no sub problems and we basically just solve the problem at hand using Fermat's little theorem or Pascal's Triangle.

That's it! Now we can compute the binomial coefficients for very large numbers for any modulo $m$.

**Tags** : algorithm (/weblog/tags/algorithm/) python (/weblog/tags/python/)

**Short url** : http://fishi.devtail.com/weblog/B/ (http://fishi.devtail.com/weblog/B/)

## Previous entry

[Another Python BrainF*** Interpreter (/weblog/2015/06/15/another-python-bf-interpreter/)]

## Next entry

[Testing ansible playbooks with ansible-vault encrypted data using Travis CI (/weblog/2016/04/02/testing-ansible-ansible-vault-travis-ci/)]

## Tags (/weblog/tags/)

algorithm (/weblog/tags/algorithm/) ansible (/weblog/tags/ansible/) c (/weblog/tags/c/) c# (/weblog/tags/c%23/) ci (/weblog/tags/ci/) dcom (/weblog/tags/dcom/) django (/weblog/tags/django/) git (/weblog/tags/git/) java

(/weblog/tags/java/) linux (/weblog/tags/linux/) network (/weblog/tags/network/) powershell (/weblog/tags/powershell/) python (/weblog/tags/python/) sysadmin (/weblog/tags/sysadmin/) windows (/weblog/tags/windows/)

## Archives

- 2015 (/weblog/2015/)
  - January (/weblog/2015/01/)
    - January 16 (/weblog/2015/01/16/)
    - January 21 (/weblog/2015/01/21/)
    - January 25 (/weblog/2015/01/25/)
  - February (/weblog/2015/02/)
    - February 6 (/weblog/2015/02/06/)
    - February 22 (/weblog/2015/02/22/)
  - March (/weblog/2015/03/)
    - March 2 (/weblog/2015/03/02/)
  - April (/weblog/2015/04/)
    - April 12 (/weblog/2015/04/12/)
  - May (/weblog/2015/05/)
    - May 18 (/weblog/2015/05/18/)
  - June (/weblog/2015/06/)
    - June 15 (/weblog/2015/06/15/)
    - June 25 (/weblog/2015/06/25/)
- 2016 (/weblog/2016/)
  - April (/weblog/2016/04/)
    - April 2 (/weblog/2016/04/02/)
  - June (/weblog/2016/06/)
    - June 2 (/weblog/2016/06/02/)