The combination formula has immediate applications to a wide range of disciplines, including statistics, project management, computer science, and finance. The wide reach of the combination formula gives it an importance that demands greater study. In this paper we will study the combination formula in some depth to encourage an intuitive understanding of its behavior and computation.

A *combination* is an unordered subset of some set . There are several common notations for combinations, including:

 can be read “ combinations taken at a time,” “ choose ,” or “ take .” There is a simple formula to compute , known as the *combination formula* or *binomial coefficient*, that we will introduce at the end of this chapter.

Consider a set . The *cardinality* (size) of this set is , so for use in the combination formula. If we wish to divide into subsets into unit size then the set containing all possible subsets is:

We can easily see that if we want subsets of size 1 then there are possible ways to do this. We can generalize this to

for all positive integers . In the special case of we say because it is never possible to take more elements than are in a set. That is, .

The case is worth investigating. If then , an empty set. Given an input set with no elements, then there is only one way to combine the empty elements of this set into another set – also an empty set. So . In fact, the same logic applies for all . It also works for the case ; there is only one combination of a set that contains of its elements. Therefore

A string of bits can be used as a convenient method to represent the presence or absence of an element in a set. For we need only six bits. We can easily iterate through all possible bit combinations to see that there are possible combinations.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 000000 | 001000 | 010000 | 011000 | 100000 | 101000 | 110000 | 111000 |
| 000001 | 001001 | 010001 | 011001 | 100001 | 101001 | 110001 | 111001 |
| 000010 | 001010 | 010010 | 011010 | 100010 | 101010 | 110010 | 111010 |
| 000011 | 001011 | 010011 | 011011 | 100011 | 101011 | 110011 | 111011 |
| 000100 | 001100 | 010100 | 011100 | 100100 | 101100 | 110100 | 111100 |
| 000101 | 001101 | 010101 | 011101 | 100101 | 101101 | 110101 | 111101 |
| 000110 | 001110 | 010110 | 011110 | 100110 | 101110 | 110110 | 111110 |
| 000111 | 001111 | 010111 | 011111 | 100111 | 101111 | 110111 | 111111 |

If and then there are 15 possible combinations of bit strings with a *Hamming weight* of two:

Notice that inverting all of these bits give the opposite combination.

With two bits set we represented all combinations with two elements chosen. By inverting each bit, we get all combinations with two elements *not* chosen. This generalizes to the identity

The sum of all possible combinations for a set of size is .

The case has an intriguing relationship to the triangular numbers. If we start with and select then we must choose one of the remaining five elements. If we do not choose and choose then we must select one of the remaining four elements. If we choose neither nor but choose then we must select one of the remaining three elements. This process continues until we choose and have only available to select. If we select no element before then it is not possible to take two elements from a subset of size one (recall ). In general,

The final case we will consider before moving on to a general formula is for . For we choose and now must take two of the remaining five elements , which is given by . Next we eliminate and choose and take two of the remaining four elements , which is given by . We select and are forced to take each element in , at which point no further combinations are possible. (Recall that in cases we define ). We therefore realize is actually the sum of triangular numbers.

We will construct a closed-form solution for but first let us develop an intuition for its structure. Visualize each triangular sum as a triangle of bricks with unit height. A stack of these bricks will form half of a pyramid. We can easily find the volume of a pyramid using calculus. Assuming the height of a pyramid relates to its width linearly by some constant then the area . Then the volume of the pyramid with a square base is given by

If the pyramid has a triangular base instead of a square then and the volume becomes

We expect to see a constant factor of in any closed-form solution to . For finding a closed form sum of triangular numbers we need the identity

We prove this identity by induction. Given a basis of

Assume the induction hypothesis is true for some and the sum is . Then at the sum must be larger by .

We see that this sum is equal to the following which proves the induction hypothesis is true for all .

Armed with the identity for summing a series of squares we can now find

We now have identities for combinations with . We could continue creating additional identities but we now have enough information to recognize the behavior of the combination formula. Again, for the purposes of building intuition, consider the hypervolume of a series of half pyramids arranged in ascending size. Though impossible to visualize, we can calculate the hypervolume using the repeated integral

Notice the denominator of the hypervolume is in four dimensions. It was in three dimensions, 2 in two dimensions, and unit in a single dimension. Further integration into increasingly high-dimensional space would increase the denominator in the hypervolume equation according to the power rule. We can also see a pattern in the numerator. At the numerator was , for the numerator was , and at the numerator was .

We can also understand this behavior intuitively. At we found combinations. because we don’t combine any element with itself (otherwise it would have been ) and to deduplicate combinations with different orderings. In the combination formula we consider sets like , , , , , and to be the same, hence we would divide by in the case of . If we wanted to count all of these orderings then we simply would not divide by this factor. A counting of subsets where ordering matters is known as the *permutation*. In general, we can recursively define any combination by the sum of combinations taken at a time.

A recursive definition to calculate arbitrary combinations is equal to the canonical definition using factorials, known as the *binomial coefficient identity*.

Another recursive definition for finding combinations views the problem as a decision tree. Consider again and the problem . We construct a binary tree rooted at the original problem , where we branch based on whether we choose or do not choose . If we choose then we are left with elements from which we must take . If we do not choose then we must choose elements from the remaining . Thus, .

At the next level of the decision tree we choose or do not choose . On one branch we had to choose two elements we now recurse to . On the other branch, where we had to choose three elements, we find . Recursion continues to base cases, such as and . The combination formula constructed as a decision tree generalizes to an identity known as *Pascal’s Formula*.

Now let us consider the computational complexity on a computer.

First, we compute directly.

def binomial(n, r, f):

 return f(n) // (f(r) \* f(n - r))

The parameter f, in this context, refers to a factorial function that is specified by the caller. Using Python’s built in math.factorial function we easily construct Pascal’s Triangle.

import math

for n in range(10):

 print(\*[binomial(n, r, math.factorial) for r in range(n+1)])

When executed, the program outputs Pascal’s Triangle:

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

1 5 10 10 5 1

1 6 15 20 15 6 1

1 7 21 35 35 21 7 1

1 8 28 56 70 56 28 8 1

1 9 36 84 126 126 84 36 9 1

It is simple to implement our own naïve factorial function:

def \_factorial(n):

 return 1 if n == 0 else n \* \_factorial(n - 1)

We can compare the performance of the built in math.factorial function to our own implementation.

def benchmark(limit, f):

 start = timer()

 for n in range(limit):

 [binomial(n, r, f) for r in range(n+1)]

 end = timer()

 return end – start

When executed we find the built in math.factorial function is considerably faster than the naïve \_factorial function we have built. This is actually quite common in programming. Developers of popular frameworks often invest in finding optimal approaches to computation that might not seem obvious. For example, it is faster in practice to compute factorials by divide and conquer rather than a straightforward multiplication.

>>> benchmark(100, \_factorial)

0.19996605199412443

>>> benchmark(100, math.factorial)

0.02862050899420865

Interestingly, we can beat Python’s built in math.factorial easily with dynamic programming by memozing factorial calculations for later. Python has a very convenient syntax for caching results of pure functions.

from functools import lru\_cache

@lru\_cache(maxsize=1024)

def \_factorial\_dp(n):

 return 1 if n == 0 else n \* \_factorial\_dp(n - 1)

This single directive instructs Python to check if an identical invocation to \_factorial\_dp has been cached and, if so, return this result immediately.

>>> \_factorial\_dp.cache\_clear()

>>> \_factorial\_dp.cache\_info()

CacheInfo(hits=0, misses=0, maxsize=1024, currsize=0)

>>> benchmark(100, \_factorial\_dp)

0.009150605008471757

>>> \_factorial\_dp.cache\_info()

CacheInfo(hits=15149, misses=100, maxsize=1024, currsize=100)

Here we see that the use of dynamic programming saved 15149 invocations of the ­\_factorial\_dp function when calculating the first 100 rows of Pascal’s Triangle. The benefits of the dynamic programming approach become increasingly pronounced as the size of the input grows.

>>> \_factorial\_dp.cache\_clear()

>>> benchmark(500, \_factorial\_dp)

1.2330169359920546

>>> benchmark(500, math.factorial)

5.665788909012917

>>> \_factorial\_dp.cache\_clear()

>>> benchmark(600, \_factorial\_dp)

2.3750273040204775

>>> benchmark(600, math.factorial)

11.008371475007152

>>> \_factorial\_dp.cache\_clear()

>>> benchmark(700, \_factorial\_dp)

4.588109176023863

>>> benchmark(700, math.factorial)

20.517555702012032

The dynamic programming approach can also be beneficial if we implement Pascal’s Formula. Again, we define a benchmarking procedure to measure the time necessary to complete the calculation. The benchmark accepts a “combiner” that will be used to calculate the first rows of Pascal’s Triangle.

def benchmark\_combination(limit, combiner):

 start = timer()

 for n in range(limit):

 [combiner(n, r) for r in range(n+1)]

 end = timer()

 return end – start

def combination(n, r):

 if r == 1:

 return n

 elif r > n:

 return 0

 elif r == 0 or r == n:

 return 1

 else:

 return combination(n-1, r-1) + combination(n-1, r)

To our horror we heuristically assess that the doubly-recursive combination function has exponential computational complexity.

>>> benchmark\_combination(10, combination)

0.00042666701483540237

>>> benchmark\_combination(15, combination)

0.01228521199664101

>>> benchmark\_combination(20, combination)

0.2757053799869027

>>> benchmark\_combination(25, combination)

8.162856736016693

Dynamic programming to the rescue!

@lru\_cache(maxsize=65536)

def combination\_dp(n, r):

 if r == 1:

 return n

 elif r > n:

 return 0

 elif r == 0 or r == n:

 return 1

 else:

 return combination\_dp(n-1, r-1) + combination\_dp(n-1, r)

We change nothing in the code but enable caching of results. The maxsize parameter controls how large the dictionary can grow. A larger dictionary is not strictly better in practice.

>>> benchmark\_combination(1000, combination\_dp)

0.6813005529984366

>>> combination\_dp.cache\_info()

CacheInfo(hits=995006, misses=500500, maxsize=65536, currsize=65536)

The memoized version of Pascal’s Formula outperforms even our memoized factorial function used in the binomial formula. Why? Pascal’s Formula consists only of addition, whereas the binomial formula requires a large factorial divided by another large factorial. Multiplication is generally quadratic, in complexity (slightly faster with Karatsuba’s method) whereas addition is an linear time operation. With dynamic programming we trade calculation for memory and reduce an exponential time algorithm to linear time.