**Binomial Coefficients**

**Definition**: The binomial coefficient  for all *n* ≥ *k* ≥ 0 (read “*n* choose *k*”)

where *n*! = *n* ⋅ (*n* – 1)⋅ (*n* – 2) ⋅ … ⋅ 2⋅ 1 and, by definition, 0! = 1.

This counts the number of *k*-element subsets of an *n*-element set, or equivalently, the number of ways to choose a set of *k* elements, taken from a set of *n* elements.

**Example 1**: Calculate:  =  =

**Example 2**: How many 8-bit strings contain exactly 3 zeros?

**Example 3**: Use the definition to simplify each of the following:

 =  =

**Example 4**: Suppose a string of *n* characters contains only two possible characters: \* and | Write an expression for the number of *n*-bit strings that contain exactly 2 \*’s. Simplify your expression to be written as a polynomial.

**Pascal’s Identity**: For all *n*  *k*  1, 

**Pascal’s Triangle**

(see <https://brilliant.org/wiki/pascals-triangle/>)

**Logarithms**

logs and exponents are inverse operations

* To “undo” multiplication, you divide.
* To “undo” addition, you subtract.
* To “undo” exponentiation, you hit it with a log.

**Definition** y = logbx ⬄ by = x Notation: We use lg *x* to represent log2*x*.

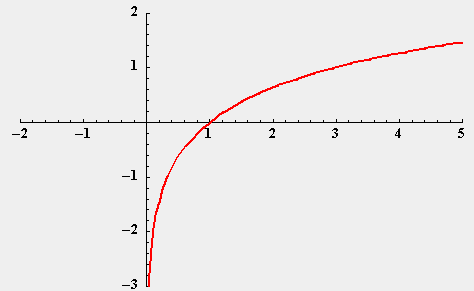
**Powers of 2**: 20 = 1, 21 = 2, 22 = 4,

**Example 1**: Use the definition to determine each of the following:

lg 32 = lg 1024 = lg 1 =

log381 = log 100,000 =

Graph of *y* = logb*x*



Note: logb1 = 0 for any base b > 0 (since b0 = 1).

The logarithm function has domain {*x* | *x* > 0}. The *x*-intercept is at *x* = 1.

The logarithm function is a strictly increasing and very slowly growing function.

**Example 2:** Use powers of 2 to calculate:

** **

**Change of Base Formula**: 

**Example 3**: Use a calculator and the change of base formula to calculate each of the following:

log3687 = log5791 =

**Properties of Logs:**

**Example 4**: Use properties of logs to simplify each of the following:

lg(8*n*)

lg(*n*2)

lg(*n*10)

lg()

lg(*nn*)

**Example 5:** The algorithm binarySearch is used to find the index for a key value in a sorted array a of n items. Determine the number of times the while loop is executed in the worst case.

binarySearch(a, key)

{

left = 1

right = a.last

while (left <= right)

{

mid = (left + right) / 2

if (key < a[mid])

right = mid - 1

else if (key > a[mid])

left = mid + 1

else

return mid // found

}

return -1 // not found

}

search 3770 student records => lg(3770) ≈ 11.8

search 332,000,000 people in US => lg (332,000,000) ≈ 28.3