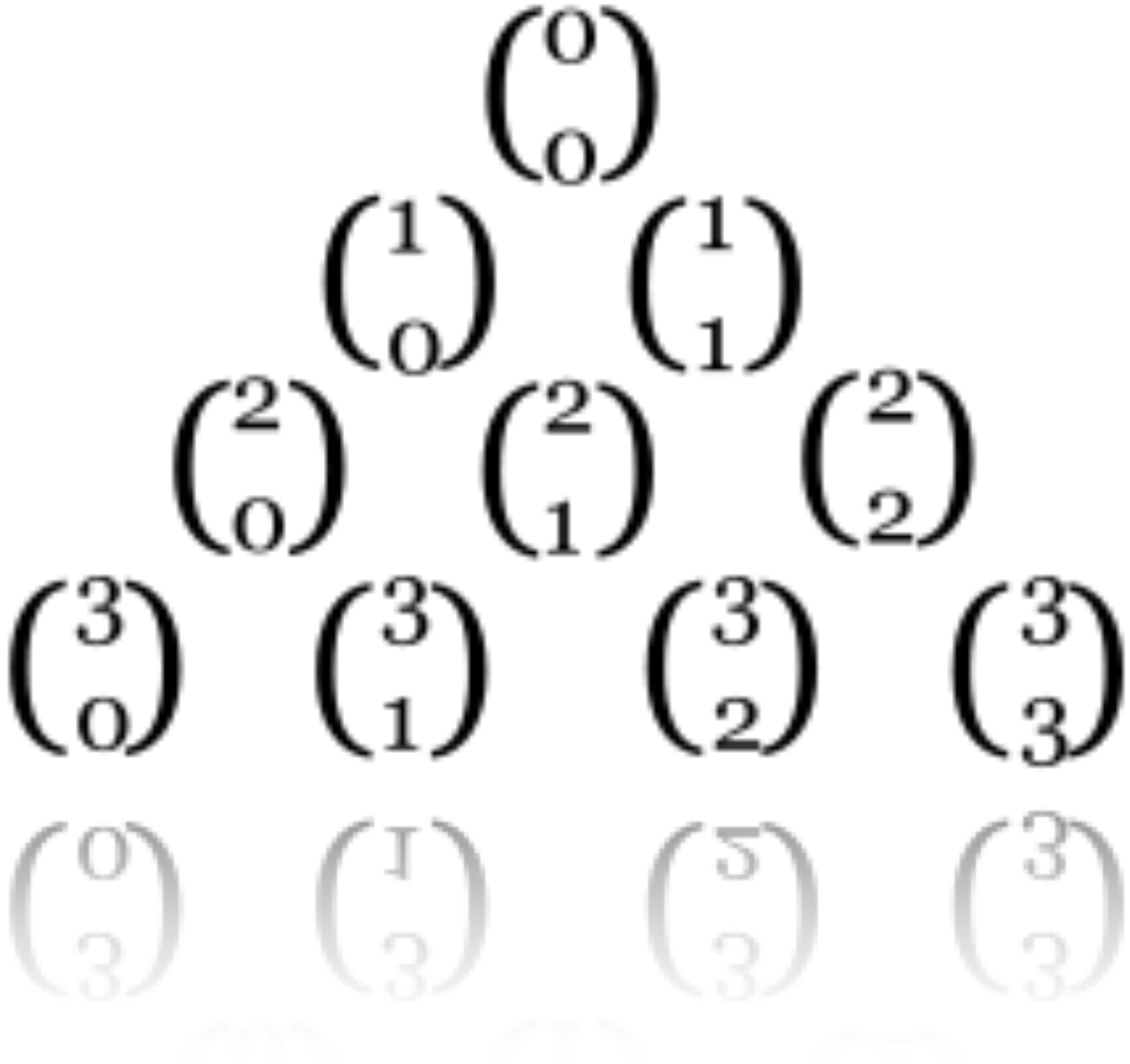
# Cálculo del Coeficiente Binomial

**Néstor Ibrahim Hernández Jorge**
**Nicolás Hernández González**
**David Pérez Rivero**

Diseño y Análisis de Algoritmos. ULL - Curso 2016/2017



# Introducción

Los coeficientes binomiales o números combinatorios hacen referencia al número de formas en que se pueden extraer subconjuntos a partir de un conjunto dado. Por ejemplo, se tiene un conjunto con 6 objetos diferentes {A,B,C,D,E,F}, de los cuales se desea escoger 2 (sin importar el orden de elección). Tendríamos las formas AB, AC, AD, AE, AF, BC, BD, BE, BF, CD, CE, CF, DE, DF, EF. Es decir, 15 subconjuntos de dos elementos.

El coeficiente binomial $\binom{n}{k}$ está dado por la fórmula $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

En el caso anterior, para el conjunto de 6 objetos diferentes, cogidos por pares sin importar el orden, la fórmula sería de la siguiente forma:

$$\binom{n}{k} = \binom{6}{2} = 15$$

La forma recursiva de calcularlo es:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{para todos los números enteros } n, k > 0,$$

Los casos iniciales son:

$$\binom{n}{0} = 1 \quad \text{para todos los números enteros } n \geq 0,$$
$$\binom{0}{k} = 0 \quad \text{para todos los números enteros } k > 0.$$

Además, para cualquier entero n se cumple que:

$$\binom{n}{0} = 1 = \binom{n}{n}$$

Por lo tanto, para la forma recursiva tenemos los siguientes casos:

$$\binom{n}{k} = \begin{cases} 1 & si\ k = 0 \\ 1 & si\ k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & si\ 0 < k < n \\ 0 & eoc \end{cases}$$

Siendo los dos primeros y el último los casos bases de parada del algoritmo recursivo.

## Algoritmo Recursivo

Mediante el algoritmo recursivo que calcula los valores, la complejidad resulta ser exponencial, debido a la repetición de los cálculos que realiza. En concreto, la complejidad es $O(2^n)$. Ya que básicamente el problema se divide en 2 subproblemas, y a su vez, cada uno de estos subproblemas se dividen en otros 2 subproblemas, así hasta llegar a los casos iniciales. El pseudocódigo sería el siguiente:

```
funcion C(n,k)
{
        Si k = 0 o k = n entonces
                devolver 1
        sino si 0 < k< n
                devolver  C(n-1,k-1) + C(n-1,k)
        sino
                devolver 0
}
```

El principal problema de esta técnica es que realizamos los mismos cálculos varias veces durante la ejecución del problema.

# Programación Dinámica

Para resolver el problema del algoritmo recursivo podemos recurrir a la programación dinámica. Para ello, podemos basarnos en el triángulo de Pascal.

### Triángulo de Pascal

El triángulo de Pascal fue uno de los trabajos pioneros en el estudio moderno de la probabilidad, establecido por Blaise Pascal en 1654.

Se tiene una cuadrícula rectangular en la cual se escribe el número 1 en las casillas del borde superior y el borde izquierdo de tal manera:

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | ... |
| 1 | | | | |
| 1 | | | | |
| 1 | | | | |
| ... | | | | |

Los números de las demás casillas se obtienen con la siguiente regla: en cada casilla se escribe la suma de los valores de las dos casillas contiguas situadas a su izquierda y en la parte superior:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | ... |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
| 1 | 3 | 6 | 10 | 15 | 21 | 28 | ... |
| 1 | 4 | 10 | 20 | 35 | 56 | 84 | ... |
| ... | ... | ... | ... | ... | ... | ... | |

Pascal noto que para cualquier coeficiente binomial de n en 0, o de n en n, da como resultado 1, como hemos explicado anteriormente. Estos, corresponden con los bordes de la matriz, y posteriormente también notó, que las demás casillas de su matriz son también coeficientes binomiales, quedando el triángulo de esta manera:

$$\begin{array}{ccccccc} & & & \binom{0}{0} & & & \\ & & \binom{1}{0} & & \binom{1}{1} & & \\ & \binom{2}{0} & & \binom{2}{1} & & \binom{2}{2} & \\ \binom{3}{0} & & \binom{3}{1} & & \binom{3}{2} & & \binom{3}{3} \end{array}$$

La afirmación de que las entradas del triángulo de Pascal son precisamente los coeficientes binomiales, se basa en la siguiente identidad, conocida ahora como *identidad de Pascal* o *teorema de Pascal*:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Entonces, sabiendo esto, creamos una tabla para ir almacenando los valores del triángulo. El array se irá construyendo por filas de arriba hacia abajo y de izquierda a derecha mediante el siguiente algoritmo (**bottom-up**):

```
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;

    // Caculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previosly stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}
```

Con esto, claramente la complejidad del algoritmo pasará a ser de O(nk), ya que son dos bucles for que van desde 0 hasta n y otro interno que va desde 0 hasta k.

Hemos pasado de una complejidad O(2n) a una complejidad polinomial O(nk). Aquí podemos ver la gran optimalidad que nos da la Programación Dinámica.

En este algoritmo, como podemos ver, el espacio que ocupa es también nk, es decir, estamos almacenando el triángulo completo. Realmente no necesitamos almacenarlo al completo, así que podemos introducir una mejora para así optimizar también el tamaño de almacenamiento de nuestro programa. Esta vez, en vez de usar un array bidimensional, usaremos uno convencional (1 dimensión). En este almacenaremos una línea del anterior array bidimensional, y con esta línea realizaremos los cálculos para obtener la siguiente línea. Es decir, sólo vamos almacenando la línea actual de la matriz.

```
int binomialCoeff(int n, int k)
{
    int C[k+1];
    memset(C, 0, sizeof(C));

    C[0] = 1;  // nC0 is 1

    for (int i = 1; i <= n; i++)
    {
        // Compute next row of pascal triangle using
        // the previous row
        for (int j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }
    return C[k];
}
```

Ahora, la complejidad sigue siendo de O(nk) pero hemos conseguido de pasar de un almacenamiento nk a uno k.

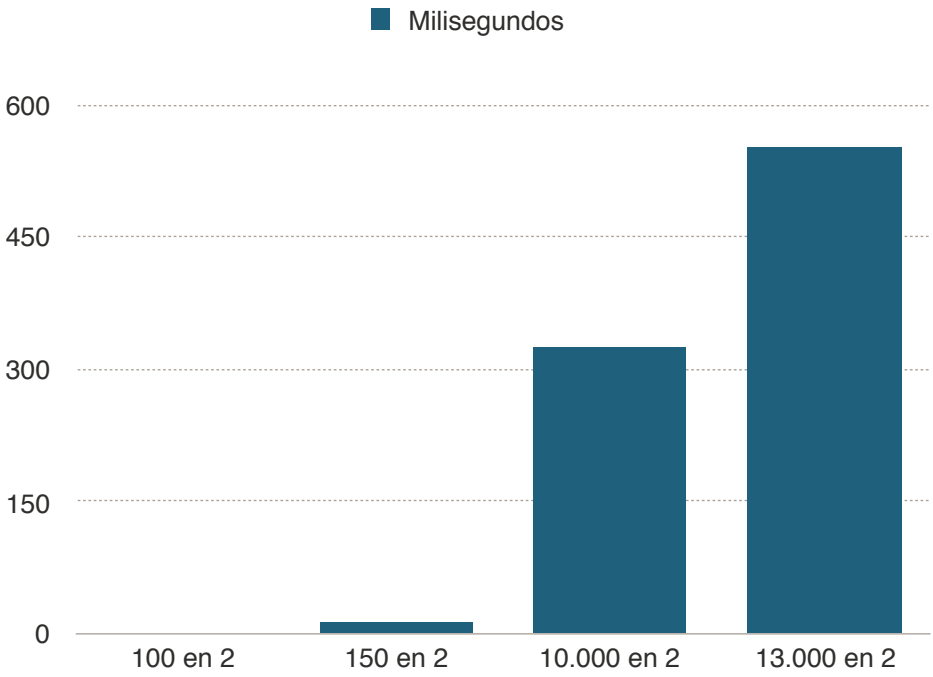
**Algoritmo Top-Down**:

La variante top-down del algoritmo, a diferencia del bottom-up, no rellena la tabla desde un principio, va realizando los cálculos, y si la posición de la tabla a la queremos acceder contiene un 0, entonces calculamos su valor, si no, simplemente retornamos su contenido, previamente calculado.

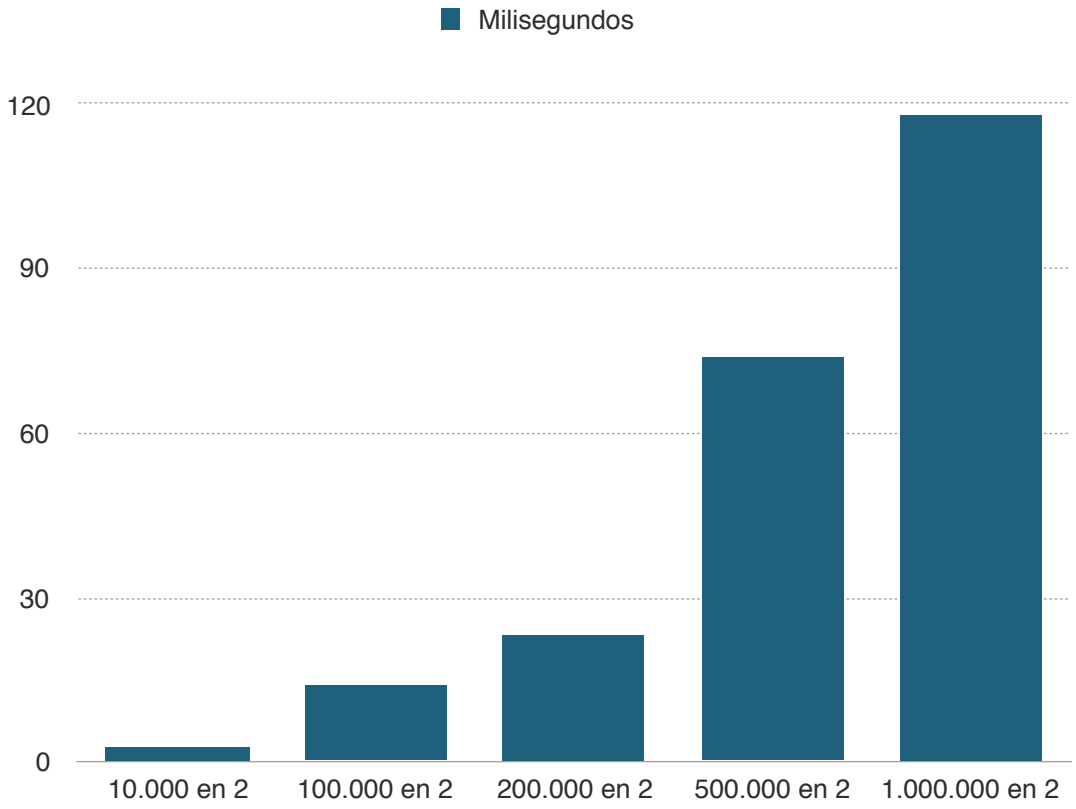
```
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
  // Base Cases
  if (k==0 || k==n)
    return 1;

  if (mem[k][n] != 0)
     return  mem[k][n];

  // Recur
  mem[k][n] = binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k)

  return mem[k][n];
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(r
    return 0;
}
```

## Análisis de Tiempo

Recursivo:




Programación Dinámica:




En las gráficas claramente podemos ver como en el recursivo, según vamos aumentando el número del conjunto de elementos el tiempo de ejecución se dispara, tanto que, solamente de pasar de 10.000 en 2 a 13.000 en 2 aumenta en casi el doble en el tiempo, además, a partir de estos datos el tiempo de ejecución se vuelve demasiado grande. En cambio, como vemos en la gráfica de la ejecución de programación dinámica podemos utilizar conjuntos mucho mayores y el tiempo de ejecución es insignificante comparado con el recursivo. También vemos como el tiempo aumenta de forma polinomial, pasamos de 500.000 en 2 a 1.000.000 en 2 y el tiempo no es ni siquiera el doble. La mejoría es claramente abrumadora.

## Conclusiones

Como podemos ver, gracias a la Programación Dinámica hemos pasado de una complejidad exponencial a una polinomial, la diferencia es bastante positiva si hablamos de optimización. Esto es algo común en casi todos los algoritmos de Programación Dinámica, pero una de las ventajas de este es que no hay que reconstruir una solución a partir de la tabla. Además, el tamaño de la tabla es menor que en otros algoritmos similares, sólo tenemos un vector de dimensión n. Con la programación dinámica podemos realizar cálculos de coeficientes mayores y además lo hace en un tiempo razonable, cosa que la recursividad no hace.

# Bibliografía

http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/

http://stackoverflow.com/questions/26228385/time-complexity-of-recursive-algorithm-for-calculating-binomial-coefficient

http://eafranco.com/docencia/analisisdealgoritmos/files/09/Diapositivas09.pdf

http://onlinejudge.inf.um.es/curso/sesion6.html

http://www.geeksforgeeks.org/space-and-time-efficient-binomial-coefficient/