COMBINATORIA
CON LA EXPRESIÓN
EN PRIMOS DEL FACTORIAL

- Simplificación de cálculos con factoriales -

https://github.com/GuillermoArriaga , *https://www.hackerrank.com/GuillermoArriaga*

por Guillermo Arriaga García *guillermoarriagag@gmail.com*

Creación del algoritmo y pruebas exitosas: marzo de 2017

Demostración matemática: noviembre de 2017

Redacción y publicación: abril de 2019

Que las matemáticas sigan emocionando la vida de muchas personas, sigan fomentando la creatividad y el desarrollo de herramientas útiles para una gran cantidad de las labores humanas y que, con la computación, una de sus principales aplicaciones y herramientas, se siga desarrollando el conocimiento y bienestar humano.

Contenido

I. Presentación 1
II. Explicación Intuitiva Introductoria 3
III. Demostraciones Matemáticas 5
3.1. Fórmula Base 5
3.1.1. Primer Modo 5
3.1.2. Segundo Modo: 11
3.2. Fórmula Complementaria 17
3.3. Fórmula Base en Presentaciones Prácticas 19
IV. Algoritmos, Complejidades y Códigos en C# 23
4.1. Procedimientos Generales 23
4.1.1. Potencia de un Factor Primo de un Factorial 23
4.1.2. Primos Hasta Algún Natural Mayor a Uno 24
4.1.3. Factorial en Factores Primos 30
4.1.4. Valor Numérico de Factores Primos 31
4.2. Procedimientos Particulares de Combinatoria 33
4.2.1. Combinaciones sin Repetición 33
4.2.2. Combinaciones con Repetición 35
4.2.3. Permutaciones sin Repetición 37
4.2.4. Permutaciones con Repetición 38
4.2.5. Variaciones sin Repetición 40
4.2.6. Variaciones con Repetición 42
4.3. Otros Procedimientos 44
4.3.1. Factores Primos de un Natural Mayor a Uno 44
4.3.2. Algunas Operaciones con Factores Primos 44
V. Código en C# 48

I. Presentación

En este documento se presenta un algoritmo justificado matemáticamente para la simplificación de operaciones de multiplicación y división de factoriales. Consiste en trabajar con factoriales en su factorización en primos.

- El cálculo de combinaciones se simplifica volviendo las divisiones de factoriales en restas de exponentes y las multiplicaciones de factoriales en sumas de sus exponentes.
- Se puede almacenar o mantener fácilmente el valor de un factorial, por su expresión en factores primos.
- Computacionalmente se agiliza el cálculo de una combinación y se puede calcular ésta para valores grandes que, con los mismos recursos computacionales, o tardaría notablemente o no sería posible.

Fórmula base:

Para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo menor o igual a n más cercano a n, sucede que:

$$n! = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{{p_n}^i}\right\rfloor}$$

La potencia total de cada primo menor a n o igual a él, es la suma de las partes enteras de las divisiones de n entre cada una de las potencias de cada primo.

Fórmula complementaria:

Sea **N** el conjunto de los enteros positivos.

$$\forall\, a, b, c \in N \left(\left\lfloor \frac{a}{b^{c+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b^c} \right\rfloor}{b} \right\rfloor \right)$$

Presentaciones Prácticas de la Fórmula Base

Para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo mayor de los menores a n+1, sucede que:

$$(1)\ n! = 2^{\sum_{i=1}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\lfloor \log_3 n \rfloor} \left\lfloor \frac{n}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\lfloor \log_5 n \rfloor} \left\lfloor \frac{n}{5^i} \right\rfloor} \cdot \ldots \cdot p_n^{\sum_{i=1}^{\lfloor \log_{p_n} n \rfloor} \left\lfloor \frac{n}{p_n{}^i} \right\rfloor}$$

$$(2)\ n! = 2^{\sum_{i=1}^{\lfloor \log_2 n \rfloor} k_{2,i}} \cdot 3^{\sum_{i=1}^{\lfloor \log_3 n \rfloor} k_{3,i}} \cdot 5^{\sum_{i=1}^{\lfloor \log_5 n \rfloor} k_{5,i}} \cdot \ldots \cdot p_n^{\sum_{i=1}^{\lfloor \log_{p_n} n \rfloor} k_{p_n,i}}$$

$$siendo\ k_{q_n,0} = n\ ,k_{q_n,i} = \left\lfloor \frac{k_{q_n,i-1}}{q_n} \right\rfloor \forall i \in N\ ,\forall q_n \in \{primos \leq n\}$$

El modo de operar con los $k_{q,i}$ es el programado para el cálculo de potencias de los factores primos.

```
public uint PotenciaPrimoDeFactorial(uint n, uint primo)
{
    uint potencia = 0;
    uint k = n / primo;

    while (k > 0)
    {
        potencia += k;
        k /= primo;
    }
    return potencia;
}
```

Algoritmo	Complejidad
Potencia de un factor primo de un factorial n	O(log n)
Primos hasta el natural n > 1	En factores primos y en valor numérico: O(n log n)
Primos del natural n > 1	
Factorial de n > 1 expresado en factores primos	
Valor numérico de factores primos provenientes de factoriales	
Combinaciones sin repetición	
Combinaciones con repetición	
Permutaciones sin repetición	
Permutaciones con repetición	
Variaciones sin repetición	
Variaciones con repetición	En factores p.: O(n log n) En valor numérico: O(n^2 log n)

II. Explicación Intuitiva Introductoria

PROBLEMA: Al diseñar una función computacional que calcule una combinación, aunque se disminuyan al mínimo las fracciones que intervienen en su cómputo final, hay problemas de tiempo que se pueden evitar si se convierten, en cierto modo, las multiplicaciones en sumas y las divisiones en restas, sin arriesgar decimales por el uso de la función logarítmica.

SOLUCIÓN: Como un factorial es un entero positivo, tiene una factorización única en primos (salvo acomodos diversos de los factores), y como para $nCr = \frac{n!}{r!(n-r)!}$ con r < n+1, entonces tienen esos tres factoriales una notable cantidad de factores primos comunes, de modo que factorizándolos en primos se logra operar las divisiones como restas de sus potencias y la multiplicación como suma de ellas. Con el algoritmo propuesto en el código para C# logré pasar las pruebas en el rango de velocidad deseada del reto después del código.

Ha sido una gran experiencia buscar una optimización del cálculo de combinatorias, después de intentar varias otras, ha sido muy gratificante construir un algoritmo simple y tengo gran gusto en compartirlo pues puede ser muy útil y es poco conocida esta solución. Pensé en publicarlo, pero indagando en internet vi que no era algo desconocido por todos en el mundo: https://janmr.com/blog/2010/10/prime-factors-of-factorial-numbers/

A continuación, te comparto algo de esta experiencia, deseo que te sirva en gran medida y aumente el interés por la investigación en computación y matemáticas.

Para un natural n>1, los factores primos de su factorial sólo pueden ser los menores a n+1, así que traté de ver cómo podría saber la potencia con que aparecerían.

Observé a 10! = 2(3)4(5)6(7)8(9)(10) = 2(3)2(2)5(2)3(7)(2)2(2)3(3)2(5) = $2^8(3^4)5^2(7)$

- Los 5 números pares que le integran a portan mínimo un factor 2. Ese se obtiene con 10/2.
- Los números que aportan doble 2 (1 vez otro nuevo dos quitando el inicial con el paso anterior) serían (10/2)/2 tomando la parte entera = 2.
- El número que aporta triple dos, un nuevo factor 2, sería ((10/2)/2)/2 = 1 en su parte entera.
- Cualquier otra aportación de dos dará cero en su parte entera.
- Así que las aportaciones de factor 2 son 5+2+1=8
- La parte entera es porque interesan los números incluidos desde 1 hasta 10; por lo que, si hay fracciones, esa mantisa indica un número que aún no se forma completamente y que aparecerá en un factoriales mayores, por lo que no se acumula con las otras mantisas de las otras divisiones de n entre las potencias de cada primo menores a n+1, ya que después sólo aportan cero factores. Aunque esto último es útil para definir una relación matemática para cualquier n con una infinidad de divisiones cuya parte entera participe en una suma convergente.

Lo que sucedió con el factor 2 análogamente con el 3, 5 y 7.

- El exponente del 3 sería [10/3] + [10/9] = 3+1 = 4.
- El exponente del 5 sería [10/5] = 2 y el del 7 sería [10/7] = 1.

Al ver a 11!, vi que la relación seguía funcionando y que las divisiones enteras aumentarían en 1 hasta la llegada un nuevo múltiplo del denominador respectivo. Por ejemplo, en 12! Con respecto a lo calculado en 11!, se aportaría un múltiplo para los denominadores 2, 4, 3 y, precisamente, sus factores primos son 2(2)3. Recordemos que el denominador 4 indica la presencia de un nuevo factor 2 si es que aumenta en uno su valor con el factorial anterior.

Así que, de entrada, se intuía confiable la relación. La programé, la probé para el cálculo de combinaciones muy grandes y me entusiasmó su éxito. La analicé con detalle y vislumbré su demostración convenciéndome de los pasos inductivos de la veracidad. Redacté meses después la demostración pues vi que no era desconocido esto.

Uno obtiene la factorización de una combinación en primos, lo que es un paso intermedio al comúnmente deseado entero final, pero la ventaja es que simplifica posteriores posibles operaciones con factoriales además de brindar un modo portátil para números enormes y que, si se desea obtener el resultado final, sólo se harán las multiplicaciones necesarias para obtenerlo, sin tener que calcular números más grandes que el resultado para que luego queden disminuidos por más operaciones.

Para calcular los primos menores a n+1 basta quedarnos con los enteros de 2 a n que no sean anulados por (o múltiplos de) los primos menores a 1 + la raíz cuadrada de n (o menores o iguales a ella). Esto surge de la pregunta qué números pueden dividir a n para saber si n es primo o no, ese grupo de candidatos contiene a los necesarios para determinar si n o sus anteriores hasta el 2 son primos o no.

- De entrada, ningún entero mayor a n le divide, él siempre se divide y el siguiente candidato cercano es n/2, el cual le dividiría si y sólo si 2 le divide. De modo que basta revisar si 2 le divide, si no lo hace, tampoco n/2 (ni siquiera sería entero este último).
- El siguiente candidato cercano sería n/3, como lo anterior, le dividiría sólo si el 3 le divide, por lo que basta revisar si el 3 le divide.
- El siguiente candidato sería n/4, cuya revisión equivalente es con el cuatro, que por no ser primo, su revisión ya fue revisada por algún primo anterior a él si hemos procedido del primo 2 en adelante.
- Esto que le pasó al 4 le pasará a todo número no primo, por lo que basta buscar candidatos primos.
- El siguiente candidato primo es el 5, involucrando la revisión de n/5 sin tener que hacerla.
- Y así seguimos hasta un primo p muy cercano o igual a la parte o frontera donde comienza lo no necesario de revisar al hacerlo con p y sus anteriores. Pues bien, p y n/p tenderán a ser lo más cercanos posibles o iguales, precisamente cuando $p<=n/p$, $p^2<=n$, estos es un p<=RaízCuadrada(n) o bien $p<n^{0.5}+1$ para escribirlo en desigualdad únicamente.
- No es necesario verificar divisiones, basta hacer un arreglo de números impares 3, 5, 7, 9, …, n. Anotando el 2 como primo que no está ahí, y comenzar a recorrer el arreglo anulando a los que se encuentren no anulados cada k posiciones a partir de que se encontró uno no anulado con valor k. Así, el 3 anulará al 9, 15, 21, … hasta llegar a n (le anule o no), regresamos al siguiente no anulado, que es el 5, con el anulamos al 15 (que ya lo estaba, pero funciona rápido si no agregamos más selección para anular), luego al 25, … hasta topar con n. Luego el 7 anulando cada 14 posiciones a partir de él, luego el 11… hasta llegar al entero menor o igual a la raíz de n.
- Así nos quedamos con una lista de números no anulados a la que agregando el 2 al principio nos da la lista de primos menores a n+1.

III. Demostraciones Matemáticas

3.1. Fórmula Base

3.1.1. Primer Modo

1. HERRAMIENTAS BÁSICAS.

1.1. **Natural**: Número entero positivo.

1.2. **Primo**: Natural mayor a 1 que tiene únicamente dos divisores distintos: él mismo y el 1.

1.2.1. De modo que, a un primo ningún primo le divide excepto él mismo.

1.3. **Teorema fundamental de la aritmética**: Todo natural mayor que 1 es un número primo o bien un único producto de números primos salvo permutaciones en el orden de los factores.

1.3.1. Si el entero positivo no es primo, entonces por lo menos le integran dos primos (pues si no, entonces le integra sólo uno y es él mismo, contradiciendo que no sea primo) y éstos son menores a él (como son factores deben ser menores a él o alguno igual a él y, como él no es primo, entonces ningún primo es igual a él).

1.4. **División euclídea con naturales**: Dados dos naturales a y b, la división euclídea asocia un cociente q y un resto r, ambos números enteros no negativos y únicos, que verifican que **a=bq+r** y que r<b, esto es que **r es elemento** del conjunto **{0, 1, 2, …, b-1}**.

1.4.1. **Divisibilidad con naturales** (división exacta). Se expresa con el símbolo |. Si dos naturales cualesquiera a, b verifican que **b|a** (b *divide exactamente a* a), entonces: b *es factor de* a y existe un natural q tal que bq = a. Usando 1.4. y la hipótesis de que la división es exacta, entonces el residuo es cero; el cociente q no puede ser cero porque entonces a=b(0)+0=0 contradiciendo que a sea natural (mayor a cero); por lo que q es entero no negativo por 1.4. y no es cero, es decir, q es natural.

1.4.2. **Parte entera de una división de naturales**. Dados dos naturales a y b, la parte entera de la división $\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{bq+r}{b} \right\rfloor = \left\lfloor q + \frac{r}{b} \right\rfloor = q + \left\lfloor \frac{r}{b} \right\rfloor = q + 0 = q$ ya que existen q y r únicos dados a y b naturales por 1.4.1. y $0 \leq r < b \rightarrow 0 \leq \frac{r}{b} < 1 \rightarrow \left\lfloor \frac{r}{b} \right\rfloor = 0$.

1.4.2.1. Si además b es factor de a (esto es que a es múltiplo de b y que b|a), sucede que $\frac{a}{b} = \frac{bq}{b} = q = \left\lfloor \frac{a}{b} \right\rfloor$

1.4.2.2. Si $\left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a+1}{b} \right\rfloor$ entonces $\left\lfloor \frac{a}{b} \right\rfloor + 1 = \left\lfloor \frac{a+1}{b} \right\rfloor$ y $b|(a+1)$

Pues $q = \left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a+1}{b} \right\rfloor = q + \left\lfloor \frac{r+1}{b} \right\rfloor \rightarrow \left\lfloor \frac{r+1}{b} \right\rfloor > 0 \rightarrow (r+1) = b$ ya que r=(b-1) es el único valor de r que satisface la última desigualdad.

Así que, $\left\lfloor \frac{a+1}{b} \right\rfloor = \left\lfloor \frac{qb+b}{b} \right\rfloor = q + 1 \rightarrow b|(a+1)$ y $\left\lfloor \frac{a+1}{b} \right\rfloor = q + 1 = \left\lfloor \frac{a}{b} \right\rfloor + 1$

1.4.2.3. Si $\left\lfloor \frac{a}{b} \right\rfloor + 1 = \left\lfloor \frac{a+1}{b} \right\rfloor$ y $b|(a+1)$ entonces $\left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a+1}{b} \right\rfloor$

Pues $\left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a}{b} \right\rfloor + 1 = \left\lfloor \frac{a+1}{b} \right\rfloor$

Además de que $b|(a+1) \rightarrow \left\lfloor \frac{a+1}{b} \right\rfloor = \frac{a+1}{b} > \frac{a}{b} \geq \left\lfloor \frac{a}{b} \right\rfloor \rightarrow \left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a+1}{b} \right\rfloor$

1.4.2.4. Por 1.4.2., 1.4.2.2. y 1.4.2.3. se tiene que dados dos naturales a, b se verifica que: $\left(\left\lfloor \frac{a}{b} \right\rfloor < \left\lfloor \frac{a+1}{b} \right\rfloor\right) \leftrightarrow \left(\left\lfloor \frac{a}{b} \right\rfloor + 1 = \left\lfloor \frac{a+1}{b} \right\rfloor\right) \leftrightarrow (b|(a+1))$

1.4.2.5. Continuando con lo expuesto en 1.4.2., 1.4.2.1., 1.4.2.2. y 1.4.2.3., si ($b=d^k$ y $b|a$) b es la potencia natural k de un natural d, entonces cada una de las potencias d, d^2, d^3, …, d^k dividen al natural **a**. Ya que si tomamos cualquiera de ellas y la llamamos potencia j sucede que: Sea j elemento (cualquiera) de {1, 2, 3, …, k}, entonces:

$$b = d^k = d^{k+0} = d^{k+j-j} = d^j d^{k-j} \quad además \quad b|a \rightarrow a = bq = d^j d^{k-j} q$$

Por lo que a es múltiplo de d^j, es decir, $d^j|a$

Como esto sucede para un j cualquiera de {1, 2, 3, …, k} entonces sucede para todos ellos.

1.4.2.6. Si dos naturales a, d cualesquiera verifican que d no divide al natural a, entonces ninguna potencia natural de d dividirá al natural a. Ya que, si hubiera alguna, llamada k, sucedería que $d^k|a$ y por 1.4.2.5. se tendría que $d^{k-1}|a$, $d^{k-2}|a$, …, $d|a$ y se contradiría que d no divide al natural a.

1.4.2.7. Notemos que $\left\lfloor \frac{a}{b} \right\rfloor > \left\lfloor \frac{a+1}{b} \right\rfloor$ no puede ser, ya que $\left\lfloor \frac{a}{b} \right\rfloor = q \leq q + \left\lfloor \frac{r+1}{b} \right\rfloor = \left\lfloor \frac{a+1}{b} \right\rfloor$ utilizando 1.4.2., de modo que $\left\lfloor \frac{a}{b} \right\rfloor$ o es menor a $\left\lfloor \frac{a+1}{b} \right\rfloor$ o son iguales. Por el resultado de 1.4.2.4. si b no divide al natural (a+1) entonces $\left\lfloor \frac{a}{b} \right\rfloor$ no es menor a $\left\lfloor \frac{a+1}{b} \right\rfloor$ por lo que sólo pueden ser iguales. Es decir, si (b no divide al natural (a+1)) entonces $\left\lfloor \frac{a}{b} \right\rfloor = \left\lfloor \frac{a+1}{b} \right\rfloor$

1.4.3. **Suma de partes enteras de divisiones de naturales**.

1.4.3.1. Considerando 1.4.2.4. y 1.4.2.5. pero con b|(a+1) se llega a que:

Dados dos naturales a, b tales que $b=d^k$ con d y k naturales y $b|(a+1)$ sucede que:

$$\sum_{j=1}^{k}\left\lfloor\frac{a+1}{d^j}\right\rfloor=\sum_{j=1}^{k}\left(\left\lfloor\frac{a}{d^j}\right\rfloor+1\right)=k+\sum_{j=1}^{k}\left\lfloor\frac{a}{d^j}\right\rfloor$$

1.4.3.2. Considerando 1.4.2.6. y 1.4.2.7. se llega a que:

Dados dos naturales a, b tales que b no divida al natural (a+1) sucede que ninguna de las potencias naturales de b dividen al natural (a+1). Sea j un natural cualquiera, entonces b^j no divide al natural (a+1) por lo que $\left\lfloor\frac{a}{b^j}\right\rfloor=\left\lfloor\frac{a+1}{b^j}\right\rfloor$. Como esto sucede para cualquier j natural, sucede para todos, así que:

$$\sum_{j=1}^{\infty}\left\lfloor\frac{a+1}{b^j}\right\rfloor=\sum_{j=1}^{\infty}\left\lfloor\frac{a}{b^j}\right\rfloor$$

1.4.3.3. Considerando 1.4.3.1. y 1.4.3.2. se llega a que:

Dados tres naturales a, d, k tales que $d^k|(a+1)$ y d^{k+1} no divide al natural (a+1) sucede que las potencias naturales d superiores a k no dividirán al natural (a+1) pues por 1.4.2.6. si no es así, entonces se contradiría que d^{k+1} no divide al natural (a+1). De modo que:

$$\sum_{j=1}^{\infty}\left\lfloor\frac{a+1}{d^j}\right\rfloor=\sum_{j=1}^{k}\left\lfloor\frac{a+1}{d^j}\right\rfloor+\sum_{j=k+1}^{\infty}\left\lfloor\frac{a+1}{d^j}\right\rfloor=k+\sum_{j=1}^{k}\left\lfloor\frac{a}{d^j}\right\rfloor+\sum_{j=k+1}^{\infty}\left\lfloor\frac{a}{d^j}\right\rfloor=k+\sum_{j=1}^{\infty}\left\lfloor\frac{a}{d^j}\right\rfloor$$

1.4.3.4. Dados tres naturales a, d, k tales que ($d^k > a$) sucede que la parte entera de la división de a entre cualquier potencia natural de d superior a k-1 tendrá como resultado cero. De modo que la suma de todas ellas será cero. Sea j cualquier entero no negativo, así que $d^{k+j}\geq d^k>a>0\rightarrow 1=\frac{d^{k+j}}{d^{k+j}}>\frac{a}{d^{k+j}}\geq\left\lfloor\frac{a}{d^{k+j}}\right\rfloor\geq 0\rightarrow 1>\left\lfloor\frac{a}{d^{k+j}}\right\rfloor\geq 0\rightarrow\left\lfloor\frac{a}{d^{k+j}}\right\rfloor=0$ por lo que $\sum_{i=k}^{\infty}\left\lfloor\frac{a}{d^i}\right\rfloor=0$

2. INDUCCIÓN MATEMÁTICA

2.1. Definiciones:

2.1.1. Sea **n** un entero mayor a uno.

2.1.1.1. n!=n(n-1)(n-2)…(2)(1) por lo que n! tiene como factores a todos los primos anteriores a (n+1) y los factores que tenga que no sean primos se factorizarán con primos anteriores a (n+1) por 1.3.1.

2.1.1.2. Por 1.3. y 1.3.1. (n+1) o es primo o es producto de primos anteriores a él.

2.1.2. Sea $\mathbf{p_m}$ el mayor de los primos menores a m+1, definido para cada m entero mayor a 1.

2.1.3. Sea $\mathbf{q_m}$ un primo menor a m+1, definido para cada m entero mayor a 1.

2.1.4. Sea **i** entero positivo.

2.1.5. Sea la propiedad **PP** verificada en n dada por la expresión en primos:

$$n!=2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot\ldots\cdot p_n{}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n{}^i}\right\rfloor}$$

2.1.5.1. Así que PP verificada en n+1 es:

$$(n+1)!=2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot\ldots\cdot p_{n+1}{}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor}$$

2.1.6. En esta inducción matemática se razonará así: El número entero 2, tiene la propiedad PP. El hecho de que cualquier número entero n, también tenga la propiedad PP, implica que n+1 también la tiene. Entonces todos los números enteros a partir de 2, tienen la propiedad PP.

2.2. **El natural 2 tiene la propiedad PP**, ya que:

2.2.1. $p_2 = 2$ por ser su única opción según la definición 2.1.2.

2.2.2. $p_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_2{}^i}\right\rfloor} = 2^{\left\lfloor\frac{2}{2^1}\right\rfloor+\sum_{i=2}^{\infty}\left\lfloor\frac{2}{2^i}\right\rfloor} = 2^{1+0} = 2 = 2\cdot 1 = 2!$ igualando la suma a cero para potencias de i>1 por 1.4.3.4., pues $2^2>2$.

2.3. **El natural (n+1) tiene la propiedad PP si es primo y n tiene la propiedad PP**, ya que:

2.3.1. Como (n+1) es primo, entonces $p_{n+1}=n+1$ por la definición 2.1.2. Pues si no, $p_{n+1}=p_n$ y no sería el mayor primo menor a (n+1)+1=n+2 pues n+1 lo es. Y las potencias naturales mayores a 1 de p_{n+1} son mayores a n+1 por lo que se usará el resultado de 1.4.3.4.

Así que $(n+1) = p_{n+1} = p_{n+1}{}^{1+0} = p_{n+1}^{\left\lfloor\frac{n+1}{p_{n+1}{}^1}\right\rfloor+\sum_{i=2}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor} = p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor}$

2.3.2. Como (n+1) es primo, entonces ningún primo anterior a él le divide por 1.2.1., los cuales son los factores primos de n!.

2.3.2.1. Tomemos a un q_n cualquiera según 2.1.3., entonces q_n no divide a (n+1) por lo que por 1.4.3.2. se tiene que $\sum_{j=1}^{\infty}\left\lfloor\frac{n}{q_n{}^j}\right\rfloor = \sum_{j=1}^{\infty}\left\lfloor\frac{n+1}{q_n{}^j}\right\rfloor$. Como esto sucede para un q_n cualquiera, entonces sucede para todos.

2.3.3. De modo que si n tiene la propiedad PP y se utilizan 2.3.1., 2.3.2. y 2.3.2.1. sucede que:

$$(n+1)! = (n+1)n! = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n{}^i}\right\rfloor}\cdot (n+1) =$$
$$2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_n{}^i}\right\rfloor}\cdot p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor}$$

Lo que vuelve cierta la inducción de verdad de la hipótesis (n tiene la propiedad PP) de n a n+1 (se ve cómo **n+1 verifica PP**) en el caso de que n+1 sea primo.

2.4. **El natural (n+1) tiene la propiedad PP si no es primo y n tiene la propiedad PP**, ya que:

2.4.1. Por lo expuesto en 2.1.1.2. n+1 ahora es producto de primos anteriores a él. Además, $\mathbf{p_n = p_{n+1}}$ pues si hubiera un primo mayor a p_n y menor a n+2 tendría que ser n+1, el cual no es primo.

2.4.2. Sea **k** natural la cantidad de factores primos de (n+1) (nótese que dicha cantidad es finita pues sus elementos son un subconjunto de {2,3,4,5,…,n+1}). Sean $\mathbf{s_1, s_2, \ldots, s_k}$ los factores primos distintos de (n+1) en orden ascendente. Sean $\mathbf{j_1, j_2, \ldots, j_k}$ las potencias naturales (máximas) correspondientes a los factores primos de (n+1) ordenados. De modo que: $(n+1) = s_1{}^{j_1}\cdot s_2{}^{j_2}\cdot \ldots \cdot s_k{}^{j_k}$ es la factorización en primos distintos de (n+1) acomodados ascendentemente.

2.4.2.1. Sea s cualquiera de los **s_1, s_2, …, s_k** y j su potencia. Como son distintos entre sí los **s_1, s_2, …, s_k** entonces todas las veces que aparece cualquiera de ellos está representada en su potencia, por lo que ella es máxima y por lo que sus sucesivas potencias naturales no dividen a (n+1) y su máxima sí divide a (n+1) pues es su factor. Con $s^j|(n+1)$, s^{j+1} no divide a n+1 y 1.4.3.3. se tiene que $\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s^i}\right\rfloor = j + \sum_{i=1}^{\infty}\left\lfloor\frac{n}{s^i}\right\rfloor$ Por lo que $s^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s^i}\right\rfloor} = s^{j+\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s^i}\right\rfloor} = s^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s^i}\right\rfloor}\cdot s^j$. Sucediendo esto en cada **s_1, …, s_k** con su correspondiente **j_1, …, j_k**. Si multiplicamos estos resultados se obtiene que: $s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_k^i}\right\rfloor} = s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_1^i}\right\rfloor}\cdot {s_1}^{j_1}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_2^i}\right\rfloor}\cdot {s_2}^{j_2}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_k^i}\right\rfloor}\cdot {s_k}^{j_k} = s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_2^i}\right\rfloor}\cdot\ldots\cdot {s_1}^{j_1}\cdot {s_2}^{j_2}\cdot\ldots\cdot {s_k}^{j_k} = s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_k^i}\right\rfloor}\cdot(n+1)$ usando 2.4.2. al final.

2.4.3. Sea **T** el conjunto de los factores primos (distintos por ser conjunto) de n! que no son factores primos de (n+1). Nótese que T es finito pues es un subconjunto de {2,3,4,…,n}.

2.4.3.1. Si T está vacío, entonces **s_1, …, s_k** son los factores primos de n! por lo que se pueden intercambiar por su correspondiente 2,3,5,7,11,…,p_n , agregando a esto que n verifica PP, 2.4.1. y 2.4.2.1. quedaría que: $(n+1)! = n!\cdot(n+1) = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot\ldots\cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n^i}\right\rfloor}\cdot(n+1) = s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_k^i}\right\rfloor}\cdot(n+1) = s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_k^i}\right\rfloor} = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot\ldots\cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_n^i}\right\rfloor} = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot\ldots\cdot p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}^i}\right\rfloor}$ usando 2.4.1. al final y quedando que **n+1 verifica PP si n verifica PP y T está vacío**.

2.4.3.2. Si T no está vacío, entonces sea **h** natural igual a la cantidad de elementos de T. Sean **t_1, t_2, …, t_h** los distintos elementos de T. Sea t un elemento cualquiera de T. Como t es primo y no es factor primo de n+1, entonces t no divide a n+1 (si algún factor entero de n+1 fuera múltiplo de t, entonces t dividiría a n+1 y sería un factor de n+1 contradiciendo que no lo es) y ninguna de sus potencias naturales lo hará por 1.4.3.2., por lo que $\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t^i}\right\rfloor = \sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t^i}\right\rfloor$ y $t^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t^i}\right\rfloor} = t^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t^i}\right\rfloor}$. Lo cual sucede para todos los elementos de T al haber sido t cualquiera de ellos.

2.4.3.2.1. Así que **s_1, …, s_k, t_1, t_2, …, t_h**, son los factores primos (distintos) de n!, por lo que se pueden intercambiar cada uno con su correspondiente 2,3,5,7,11,…,p_n, agregando a esto que n verifica PP, 2.4.1., 2.4.2.1. y 2.4.3.2. quedaría que: $(n+1)! = n!\cdot(n+1) = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot\ldots\cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n^i}\right\rfloor}\cdot(n+1) = t_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t_1^i}\right\rfloor}\cdot t_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t_2^i}\right\rfloor}\cdot\ldots\cdot t_h^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t_h^i}\right\rfloor}\cdot s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s_k^i}\right\rfloor}\cdot(n+1) = t_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t_1^i}\right\rfloor}\cdot t_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t_2^i}\right\rfloor}\cdot\ldots\cdot t_h^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t_h^i}\right\rfloor}\cdot s_1^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_1^i}\right\rfloor}\cdot s_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_2^i}\right\rfloor}\cdot\ldots\cdot s_k^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s_k^i}\right\rfloor} =$

$$2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot\ldots\cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_n{}^i}\right\rfloor}=2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot\ldots\cdot p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor}$$

usando 2.4.1. al final y quedando que **n+1 verifica PP si n verifica PP y T no está vacío**.

2.4.4. Por 2.4.3.1. y 2.4.3.2.1. se concluye que **si n verifica PP, entonces n+1 natural no primo verifica PP**.

2.5. Por 2.3.3. y por 2.4.4. se concluye que **si n natural tiene la propiedad PP, entonces n+1 tiene la propiedad PP**.

2.6. Así que la verdad del caso inicial con valor 2 según 2.2. ha sido inducida a los siguientes naturales por 2.5. Así pues, **se concluye que**:

Para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo mayor de los menores a n+1, sucede que:

$$n!=2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot\ldots\cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n{}^i}\right\rfloor}$$

La potencia total de cualquier primo menor a n+1, es la suma de las partes enteras de las divisiones de n entre cada una de las potencias del primo en cuestión.

Quod erat demonstrandum.

3.1.2. Segundo Modo:

1. **VARIABLES A UTILIZAR.**
   1.1. Sea **n** un entero mayor a uno.
   1.2. Sea $\mathbf{p_m}$ el mayor de los primos menores a m+1, definido para cada m entero mayor a 1.
   1.3. Sea la hipótesis de inducción dada por:

$$n! = 2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{{p_n}^i}\right\rfloor}$$

   1.4. Sea $\mathbf{q_m}$ un primo menor a m+1, definido para cada m entero mayor a 1.
   1.5. Sea **i** un entero positivo.
   1.6. Sea un número natural un entero mayor a cero.

2. **NOCIONES SOBRE LA SUMA DE PARTES ENTERAS** de divisiones entre las potencias naturales de un entero positivo.
   2.1. Cualquier suma $\sum_{i=1}^{\infty}\left\lfloor\frac{n}{{q_n}^i}\right\rfloor$ es convergente pues comprende una cantidad finita de enteros mayores a cero y una suma de una infinidad de ceros a partir de que las potencias superan a n, ya que:
      2.1.1. La parte entera de la división de **n** entre cada potencia de $\mathbf{q_n}$ que sea mayor a **n**, será la parte entera de un número mayor a cero y menor a uno, lo que es cero. Por lo que, a partir del valor de i con el que $\mathbf{q_n}^i$ sea mayor a n, sus sucesivos valores producirán sumandos con valor cero, siendo una suma infinita de ceros.
      2.1.2. Como $\mathbf{q_n}$ es menor o igual a **n**, entonces por lo menos puede valer 1 la suma de 2.1. y en alguna de sus sucesivas potencias comenzará a ser mayor a **n**, dejando detrás una suma de una cantidad finita de enteros positivos y aportando un cero a esa suma.
      2.1.3. Por 2.1.2. la parte significativa de sumandos de 2.1. es finita y la demás parte es la suma de una infinidad de ceros, dando como resultado una suma infinita convergente para 2.1.
      2.1.4. Como 2.1. aplica para $\mathbf{q_n}$, un número que sólo requiere ser primo y menor a **n+1**, entonces lo visto en estos apartados englobados en 2.1. aplican para todas las sumas estilo 2.1. donde su dividendo sea cualquier potencia natural de cualquier primo menor a **n+1**.

   2.2. Si $\left\lfloor\frac{n}{{q_n}^i}\right\rfloor < \left\lfloor\frac{n+1}{{q_n}^i}\right\rfloor$ entonces $\left\lfloor\frac{n}{{q_n}^i}\right\rfloor + 1 = \left\lfloor\frac{n+1}{{q_n}^i}\right\rfloor$, siendo n, i, q_n como se definieron en 1. ya que:
      2.2.1. $\mathbf{q_n}$ al ser primo es mayor a 1. Por lo que también cualquiera de sus potencias lo será.

2.2.2. $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor + 1 = \left\lfloor \frac{n}{{q_n}^i} + 1 \right\rfloor = \left\lfloor \frac{n+{q_n}^i}{{q_n}^i} \right\rfloor \geq \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor > \left\lfloor \frac{n}{{q_n}^i} \right\rfloor$ obteniendo el mayor o igual por 2.2.1. (mayor) y por la pérdida de decimales (posibilidad de igual) de la toma de la parte entera, y obteniendo la última desigualdad por la condición de 2.2.

2.2.3. Como $\left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor$ es entero y $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor + 1 \geq \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor > \left\lfloor \frac{n}{{q_n}^i} \right\rfloor$, entonces sólo puede ser $\left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor = \left\lfloor \frac{n}{{q_n}^i} \right\rfloor + 1$. Éste resultado aplica para cualquier n, i, q_n que satisfagan lo pedido en 1. Es decir, aplica para todas las potencias de cada primo menor a n+1 y para todo n entero mayor a 1.

2.3. Como $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor$ no puede ser menor a $\left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor$ ya que el numerador del segundo es mayor al del primero; y por la pérdida de decimales de la toma de la parte entera de la división, entonces sólo puede suceder que $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor = \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor$ o bien que $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor < \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor \rightarrow \left\lfloor \frac{n}{{q_n}^i} \right\rfloor + 1 = \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor$. Es decir, que o son iguales o son enteros consecutivos. Éste resultado aplica para cualquier n, i, q_n que satisfagan lo pedido en 1.

2.4. Si $\left\lfloor \frac{n}{{q_n}^i} \right\rfloor < \left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor$ entonces n+1 es múltiplo de q_n^i, ya que:

2.4.1. Por 2.2.3. $\left\lfloor \frac{n+1}{{q_n}^i} \right\rfloor = \left\lfloor \frac{n}{{q_n}^i} \right\rfloor + 1$, esto indica que se estaba cerca de pasar al siguiente entero en el cociente con numerador n y ello se logró en el cociente con numerador n+1; además de que, n puede pensarse como un múltiplo de su divisor q_n^i o cero más algún número entero entre 0 y q_n^i-1, así en n+1 se alcanza el siguiente múltiplo del divisor.

3. **NOCIONES SOBRE LA DIVISIÓN EXACTA** de un entero positivo.

3.1. La división exacta o con residuo cero se representa con |.

3.2. Si r es un factor de n entonces r divide exactamente a n: r | n.

3.3. Si r_2 es un factor de r, entonces también divide a n: $r_2|r$ y $r|n \rightarrow r_2|n$.

3.4. Si r^j | n para algún j natural y r factor de n, entonces $r^j|n$, $r^{j-1}|n$, $r^{j-2}|n$, …, $r^{j-(j-1)}|n$ por la aplicación sucesiva de 3.3. Esto indica que una cantidad j de potencias naturales de r dividen a n.

3.5. Si r^j no divide a n para algún j natural y r natural, entonces r^{j+1} tampoco divide a n, ni cualquier otra potencia superior de r le dividirá, pues de lo contrario habría contradicción con que r^j no le divida, pues si una potencia superior dividiera a n, por 3.4. resultaría que sus potencias anteriores sí le dividirían, en particular r^j.

3.6. Un natural divide a otro si es su factor y si le divide es uno de sus factores, es decir, un natural divide a otro si y sólo si es su factor.

4. **NOCIONES SOBRE LOS ENTEROS POSITIVOS MAYORES A 1.**
   4.1. Un entero tal como n, según 1.1., es un número primo o bien un único producto de números primos, según el teorema fundamental de la aritmética.
   4.2. Si n no es primo, entonces es un producto de primos menores a n. Pues ninguno mayor le puede dividir y no hay alguno igual a él, ya que n no es primo en esta opción.
   4.3. Como un primo sólo es dividido (exactamente) por él mismo y el 1, entonces a un primo ningún primo anterior a él le divide.

5. **INDUCCIÓN MATEMÁTICA SOBRE n**.
   5.1. **Verdad en el valor inicial n=2**, ya que:
      5.1.1. $p_2 = 2$ por ser su única opción según la definición 1.2.
      5.1.2. $p_2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_2{}^i}\right\rfloor} = 2^{\left\lfloor\frac{2}{2^1}\right\rfloor+\sum_{i=2}^{\infty}\left\lfloor\frac{2}{2^i}\right\rfloor} = 2^{1+0} = 2 = 2\cdot 1 = 2!$ igualando la suma a cero para potencias de i>1 por 2.1.1., pues $2^i>2$.

   5.2. **La verdad se irá induciendo** de n=2 a n=3, de n=3 a n=4, llegando a todos los enteros mayores a 2 **si es que** la hipótesis de inducción 1.3., aplicada para un valor n cualquiera según 1.1., implica la verdad para dicha fórmula aplicada al entero sucesor n+1. Lo cual es cierto ya que:
      5.2.1. Como n+1 es entero positivo mayor a 1, entonces o es primo o producto de primos anteriores a él por 4. Veamos si la inducción se cumple en el **caso de que n+1 sea primo**:
         5.2.1.1. Como n+1 es primo, entonces $p_{n+1}=n+1$ por la definición 1.2. Si no, $p_{n+1}=p_n$ y no sería el mayor primo menor a n+2 pues n+1 lo es.
            5.2.1.1.1. $n+1 = p_{n+1} = {p_{n+1}}^{1+0} = p_{n+1}^{\left\lfloor\frac{n+1}{p_{n+1}{}^1}\right\rfloor+\sum_{i=2}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor} = p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_{n+1}{}^i}\right\rfloor}$ igualando la suma a cero para potencias de i>1 por 2.1.1., pues como $p_{n+1}=n+1$, entonces las potencias naturales >1 de p_{n+1} son mayores a n+1.
         5.2.1.2. Como n+1 es primo, entonces ningún primo anterior a él le divide por 4.3. Lo que conlleva que ningún primo desde el 2 hasta p_n le dividen.
            5.2.1.2.1. Por 3.5. ninguna potencia de cualquiera de los primos del 2 hasta p_n divide a n+1.
            5.2.1.2.2. Por lo que $\left\lfloor\frac{n}{{q_n}^i}\right\rfloor = \left\lfloor\frac{n+1}{{q_n}^i}\right\rfloor$ por 2.3., siendo q_n cualquiera de los primos del 2 hasta p_n ya que si fuera $\left\lfloor\frac{n}{{q_n}^i}\right\rfloor + 1 = \left\lfloor\frac{n+1}{{q_n}^i}\right\rfloor$ entonces sí le dividiría $q_n{}^i$ a n+1 por 2.4.
            5.2.1.2.3. Así que $2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{5^i}\right\rfloor}\cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{p_n{}^i}\right\rfloor}$ es igual a $2^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{2^i}\right\rfloor}\cdot 3^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{3^i}\right\rfloor}\cdot 5^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{5^i}\right\rfloor}\cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{p_n{}^i}\right\rfloor}$
         5.2.1.3. Uniendo 1.3., la hipótesis de inducción, 5.2.1.1.1. y 5.2.1.2.3. se tiene que:

$$(n+1)! = (n+1)n! = (n+1) \cdot 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{5^i} \right\rfloor} \cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{p_n{}^i} \right\rfloor}$$
$$= 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{5^i} \right\rfloor} \cdot \ldots \cdot p_n^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{p_n{}^i} \right\rfloor} \cdot p_{n+1}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{p_{n+1}{}^i} \right\rfloor}$$

Lo que vuelve cierta la inducción de verdad de la hipótesis de n a n+1 en el caso de que n+1 sea primo.

5.2.2. Veamos si la inducción se cumple en el **caso de que n+1 no sea primo**:

5.2.2.1. Por lo expuesto en 5.2.1. n+1 ahora es producto de primos anteriores a él. Además, $\mathbf{p_n = p_{n+1}}$ pues si hubiera un primo mayor a p_n y menor a n+2 tendría que ser n+1, el cual no es primo.

5.2.2.2. Sea S el conjunto de los factores primos de n+1. S no es vacío por 5.2.1., por lo menos tiene dos elementos y es un subconjunto de los factores primos de n! por ser 2, 3, 5, …, p_n primos menores a n+1 por 1.2. Además, S es finito pues los posibles integrantes son mayores a 1, menores a n+2 y enteros.

5.2.2.3. Sea **s** un elemento cualquiera de S. Entonces:

5.2.2.3.1. Existe un natural **j** tal que s^j es factor de n+1 y s^{j+1} no lo es, es decir, j es la potencia máxima de s tal que s^j es factor de n+1. Como un factor divide a aquel de quien es factor, entonces $s^j|(n+1)$.

5.2.2.3.2. Como $s^j|(n+1)$, entonces j potencias de s dividen a n+1 por 3.4. y son 1, 2, 3, …, j. Por 2.4. y 2.4.1., cada una de estas potencias o valores para **i** verifica que $\left\lfloor \frac{n+1}{s^i} \right\rfloor = \left\lfloor \frac{n}{s^i} \right\rfloor + 1$ y **se concluye que**
$\sum_{i=1}^{j}\left(\left\lfloor \frac{n}{s^i} \right\rfloor + 1\right) = \sum_{i=1}^{j}\left\lfloor \frac{n+1}{s^i} \right\rfloor = \left(\sum_{i=1}^{j}\left\lfloor \frac{n}{p_n{}^i} \right\rfloor\right) + j$

5.2.2.3.3. Como s^{j+1} no es factor de n+1 por 5.2.2.3.1. y por 3.6., entonces no divide (exactamente) a n+1 y tampoco lo harán potencias naturales mayores de s por 3.5.

5.2.2.3.4. En 2.4. se dice que si $\left\lfloor \frac{n}{s^i} \right\rfloor < \left\lfloor \frac{n+1}{s^i} \right\rfloor$ entonces n+1 es múltiplo de s^i, en particular para valores de i en j+1, j+2, j+3, … . Por lo expuesto anteriormente en 5.2.2.3.3. en estos valores para i sucede que s^i no divide a n+1 y, como no le divide, no es su factor por 3.6., por lo que n+1 no es su múltiplo, por lo que no sucede que $\left\lfloor \frac{n}{s^i} \right\rfloor < \left\lfloor \frac{n+1}{s^i} \right\rfloor$ (ya que A implica B si y sólo si no B implica a no A, en este caso se niega la consecuencia: n+1 no es múltiplo de s^i por lo que la causa es negada). Por 2.3. sólo puede suceder el caso $\left\lfloor \frac{n}{s^i} \right\rfloor = \left\lfloor \frac{n+1}{s^i} \right\rfloor$ y se verifica para toda i elemento de {j+1, j+2, j+3, …}. **De modo que** $\sum_{i=j+1}^{\infty}\left\lfloor \frac{n}{s^i} \right\rfloor = \sum_{i=j+1}^{\infty}\left\lfloor \frac{n+1}{s^i} \right\rfloor$

5.2.2.3.5. Por 5.2.2.2., 5.2.2.3., 5.2.2.3.2. y 5.2.2.3.4., como sucedió para un elemento s cualquiera de S, el conjunto de factores primos de n+1, entonces para cualquier factor primo s de n+1 sucede que:

$$s^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{s^i}\right\rfloor} = s^{\sum_{i=1}^{j}\left\lfloor\frac{n+1}{s^i}\right\rfloor+\sum_{i=j+1}^{\infty}\left\lfloor\frac{n+1}{s^i}\right\rfloor} = s^{\sum_{i=1}^{j}\left\lfloor\frac{n}{s^i}\right\rfloor+j+\sum_{i=j+1}^{\infty}\left\lfloor\frac{n}{s^i}\right\rfloor} = s^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{s^i}\right\rfloor} \cdot s^j$$

5.2.2.4. Sea **T** el conjunto de los factores primos de n! que no son factores primos de n+1.

5.2.2.4.1. Si T es vacío entonces todos los factores primos de n+1 son todos los factores primos de n! y se puede continuar sin problema alguno con 5.2.2.5.

5.2.2.4.2. Si T no es vacío, sea **t** un elemento cualquiera de T. Entonces:

5.2.2.4.2.1. Como t es primo y no es factor primo de n+1, entonces t no divide a n+1 (si algún factor entero de n+1 fuera múltiplo de t, entonces t dividiría a n+1, lo que no sucede) y ninguna de sus potencias naturales lo hará por 3.5., por lo que en $\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t^i}\right\rfloor$ sucede lo descrito en 5.2.2.3.3. y 5.2.2.3.4. para s ahora con t, quedando que:

$\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t^i}\right\rfloor = \sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t^i}\right\rfloor$ y $t^{\sum_{i=1}^{\infty}\left\lfloor\frac{n}{t^i}\right\rfloor} = t^{\sum_{i=1}^{\infty}\left\lfloor\frac{n+1}{t^i}\right\rfloor}$

5.2.2.4.3. Como 5.2.2.4.2.1. sucede para cualquier t elemento de T, sucede entonces para todos ellos. Notemos que cada uno de los elementos de t es igual a uno y sólo uno de los factores primos de n!, por la definición de T en 5.2.2.4.

5.2.2.5. Ahora bien, como sucede 5.2.2.3.5. para cada uno de los factores primos de n+1, cada uno con su potencia máxima indicada por su j correspondiente descrita en 5.2.2.3.1. (que bien puede distinguirse desde su definición como j_s) entonces con el extremo derecho de la ecuación anterior, cada factor s elevado a su j correspondiente (o j_s) multiplicado uno con otro hasta haber incluido a todos los de S, formarán precisamente a n+1 pues son su factorización en primos. Y cada uno de estos factores primos de n+1 están presentes en los factores primos de n! (ver 5.2.2.2.) podrán mantener su presentación de la hipótesis de inducción con valor en n, multiplicado por cada uno de los factores primos que reconstruyen a n+1, y se podrá apreciar la inducción de verdad a n+1 de la siguiente manera:

Sea **k** natural igual a la cantidad de elementos de S (notar que S es finito).
Sean $\mathbf{s_1, s_2, \ldots, s_k}$ los distintos elementos de S y $\mathbf{j_1, j_2, \ldots, j_k}$ las correspondientes potencias naturales máximas, como se describe en 5.2.2.3.1. para cada elemento de S.
De modo que: $(n+1) = {s_1}^{j_1} \cdot {s_2}^{j_2} \cdot \ldots \cdot {s_k}^{j_k}$ por las definiciones hechas ya que es su factorización en primos.

Sea **h** natural igual a la cantidad de elementos de T (también es finito, pues sus elementos son un subconjunto de {2, 3, 4, 5, …, n} ver 5.2.2.4. para su definición).
Sean $\mathbf{t_1, t_2, \ldots, t_h}$ los distintos elementos de T.

Notemos que la unión de T y S es el conjunto de los factores primos de n!, el cual, a su vez, es el conjunto de los factores primos de (n+1)! por las definiciones de T y S. Así, es posible intercambiar todos los factores primos 2, 3, 5, …, p_n por sus correspondientes s_1, s_2, …, s_k, t_1, t_2, …, t_h.

Ahora bien, veamos la reunión final de lo analizado anteriormente:

$$(n+1)! = (n+1)n! = ({s_1}^{j_1} \cdot {s_2}^{j_2} \cdot \ldots \cdot {s_k}^{j_k}) \cdot 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{2^i} \right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{p_n}^i} \right\rfloor}$$
$$= \left({s_1}^{j_1} \cdot {s_1}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{s_1}^i} \right\rfloor} \cdot \ldots \cdot {s_k}^{j_k} \cdot {s_k}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{s_k}^i} \right\rfloor} \right) \cdot {t_1}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{t_1}^i} \right\rfloor} \cdot \ldots \cdot {t_h}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{t_h}^i} \right\rfloor}$$
$$= {s_1}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{s_1}^i} \right\rfloor} \cdot \ldots \cdot {s_k}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{s_k}^i} \right\rfloor} \cdot {t_1}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{t_1}^i} \right\rfloor} \cdot \ldots \cdot {t_h}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{t_h}^i} \right\rfloor}$$
$$= 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{5^i} \right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{p_n}^i} \right\rfloor}$$
$$= 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{5^i} \right\rfloor} \cdot \ldots \cdot {p_{n+1}}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n+1}{{p_{n+1}}^i} \right\rfloor}$$

Habiéndose usado 5.2.2.3.5. y 5.2.2.4.2.1. (si T fuera vacío, basta con 5.2.2.3.5 y se llega a lo mismo sin haber escritos los factores t). Por último, se usó 5.2.2.1 donde se explicita que $p_{n+1}=p_n$.

Esto vuelve cierta la inducción de verdad de la hipótesis de n a n+1 en el caso de que n+1 no sea primo.

5.3. Así que la verdad del caso inicial ha sido inducida a sus siguientes casos por lo mostrado en 5.2.1.3. y 5.2.2.5., ya que en ellos se muestra que la hipótesis de inducción asumida para n! implica su certeza en (n+1)! al cumplir los requisitos de la fórmula para n+1. Como esta especie de conexión sucede para cualquier n natural mayor a 1, entonces sucede para todo n natural mayor a 1 y **se concluye que**:

Para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo menor o igual a n más cercano a n, sucede que:

$$n! = 2^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{5^i} \right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\infty}\left\lfloor \frac{n}{{p_n}^i} \right\rfloor}$$

La potencia total de cada primo menor a n o igual a él, es la suma de las partes enteras de las divisiones de n entre cada una de las potencias de cada primo.

Quod erat demonstrandum.

3.2. Fórmula Complementaria

Sea **N** el conjunto de los enteros positivos.

$$P.D.\ \ \forall\, a,b,c \in N\left(\left\lfloor\frac{a}{b^{c+1}}\right\rfloor=\left\lfloor\frac{\left\lfloor\frac{a}{b^{c}}\right\rfloor}{b}\right\rfloor\right)$$

Sean **a**, **b**, **c** enteros positivos.

Por el teorema de la división euclídea:

$$\exists\, k,r \in N\cup\{0\}\colon (a=(k)(b^{c})+r\,, r\in\{0,1,2,\dots,b^{c}-1\})$$

Obsérvese que:

$$(0\le r\le b^{c}-1<b^{c})\to\left(0=\frac{0}{b^{c}}\le\frac{r}{b^{c}}<\frac{b^{c}}{b^{c}}=1\right)\to\left(0\le\frac{r}{b^{c}}<1\right)\to\left\lfloor\frac{r}{b^{c}}\right\rfloor=0$$

Por lo que:

$$\left\lfloor\frac{a}{b^{c}}\right\rfloor=\left\lfloor\frac{(k)(b^{c})+r}{b^{c}}\right\rfloor=\left\lfloor k+\frac{r}{b^{c}}\right\rfloor=k+\left\lfloor\frac{r}{b^{c}}\right\rfloor=k+0=k \qquad \left\lfloor\frac{\left\lfloor\frac{a}{b^{c}}\right\rfloor}{b}\right\rfloor=\left\lfloor\frac{k}{b}\right\rfloor$$

Además de que:

$$\left(0\le\frac{r}{b^{c}}<1\right)\to\left(\frac{k}{b}=\frac{k+0}{b}\le\frac{k+\frac{r}{b^{c}}}{b}<\frac{k+1}{b}\right)$$

Por el teorema de la división euclídea:

$$\exists\, d,e \in N\cup\{0\}\colon (k=(d)(b)+e\,, e\in\{0,1,2,\dots,b-1\})$$

Junto con que $\left(0\le\frac{r}{b^{c}}<1\right)$ resulta que:

$$d\le\frac{(d)(b)+e}{b}=\frac{k}{b}\le\frac{k}{b}+\frac{\frac{r}{b^{c}}}{b}\le d+\frac{b-1}{b}+\frac{\frac{r}{b^{c}}}{b}=d+1-\frac{1}{b}+\frac{\frac{r}{b^{c}}}{b}$$

$$<d+1-\frac{1}{b}+\frac{1}{b}=d+1$$

Esto es que:

$$d \leq \frac{k}{b} + \frac{\frac{r}{b^c}}{b} < d + 1$$

Por lo que:

$$\left\lfloor \frac{k}{b} \right\rfloor = d \leq \left\lfloor \frac{k}{b} + \frac{\frac{r}{b^c}}{b} \right\rfloor \leq \frac{k}{b} + \frac{\frac{r}{b^c}}{b} < d + 1 \qquad \left\lfloor \frac{k}{b} \right\rfloor = \left\lfloor \frac{k + \frac{r}{b^c}}{b} \right\rfloor$$

Con lo que se observa que:

$$\left\lfloor \frac{a}{b^{c+1}} \right\rfloor = \left\lfloor \frac{(k)(b^c) + r}{(b)(b^c)} \right\rfloor = \left\lfloor \frac{k + \frac{r}{b^c}}{b} \right\rfloor = \left\lfloor \frac{k}{b} \right\rfloor$$

Y se encuentra la igualdad concluyente:

$$\left\lfloor \frac{a}{b^{c+1}} \right\rfloor = \left\lfloor \frac{k}{b} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b^c} \right\rfloor}{b} \right\rfloor \qquad \left\lfloor \frac{a}{b^{c+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b^c} \right\rfloor}{b} \right\rfloor$$

Como a, b, c son cualesquiera enteros positivos, esta igualdad aplica para todos ellos, en particular para los n, p, i requeridos en cada integrante de la fórmula base.

$$\forall\, a, b, c \in N \left(\left\lfloor \frac{a}{b^{c+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{a}{b^c} \right\rfloor}{b} \right\rfloor \right)$$

Quod erat demonstrandum.

Esta fórmula indica que la parte entera de la división de un natural “a” entre una potencia natural “c+1” de un natural “b” equivale al resultado de la división entera entre p del resultado de la división entera de “a” entre la potencia “c” de “b”.

Con esto, es posible reutilizar los resultados de cálculos anteriores en la determinación de la potencia de cualquier primo específico de la fórmula base, la cual es la suma de las divisiones enteras de potencias naturales del primo en cuestión. Esta reutilización de resultados conlleva una optimización de cálculos.

3.3. Fórmula Base en Presentaciones Prácticas

En la sección **1.4.3.4.** del primer modo (**3.1.1.**) y en la sección **2.1.** del segundo modo (**3.1.2.**) se indica que la suma infinita de las partes enteras de las divisiones de "**n**" entre las potencias naturales de cada primo "**p**" es convergente, ya que las potencias naturales de p mayores a n aportan cero como parte entera. Esta suma infinita está constituida por una suma infinita de ceros y una cantidad finita de partes enteras de la división de "**n**" entre potencias naturales de "**p**" menores o iguales a "**n**".

Veamos a detalle estas indicaciones aceptables.

Sea **n** un entero mayor a 1. Sea **N** el conjunto de los enteros positivos.

Sea $\mathbf{q_m}$ un primo menor al m+1, definido para cada m entero mayor a 1. Obsérvese que $\mathbf{q_n}$ puede ser cualquier primo ≤ n y lo que verifique $\mathbf{q_n}$ lo verificará cualquier primo ≤ n, pues basta con reunir el requisito de ser un primo menor a n + 1.

$$q_n \leq n \quad \rightarrow \quad 1 \leq \frac{n}{q_n} \quad \rightarrow \quad 1 \leq \left\lfloor \frac{n}{{q_n}^i} \right\rfloor \leq \sum_{i=1}^{\infty} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor$$

Por el teorema de la división euclídea:

$$\exists\, k_1, r_1 \in N \cup \{0\}\colon (n = (k_1)(q_n) + r_1\,, r_1 \in \{0, 1, 2, \dots, q_n - 1\})$$

$$\left\lfloor \frac{n}{q_n} \right\rfloor = \left\lfloor \frac{(k_1)\left(q_n\right) + r_1}{q_n} \right\rfloor = \left\lfloor k_1 + \frac{r_1}{q_n} \right\rfloor = k_1 + \left\lfloor \frac{r_1}{q_n} \right\rfloor = k_1$$

Por el orden de los naturales o $k_1 < \mathbf{q_n}$ o $k_1 \geq \mathbf{q_n}$ y usando la fórmula complementaria:

Si $k_1 < q_n$, entonces $\left\lfloor \frac{n}{{q_n}^2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{n}{q_n} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{k_1}{q_n} \right\rfloor = 0 \quad y \quad \sum_{i=2}^{\infty} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = 0$

Si $k_1 \geq q_n$, entonces $\exists\, k_2, r_2 \in N \cup \{0\}\colon (k_1 = (k_2)(q_n) + r_2\,, r_2 \in \{0, 1, 2, \dots, q_n - 1\})$

En este caso, se procedería análogamente a lo anterior: o $k_2 < q_n$ o $k_2 \geq q_n$

Si $k_2 < q_n$, entonces $\left\lfloor \frac{n}{{q_n}^3} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{n}{{q_n}^2} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{k_1}{q_n} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{k_2}{q_n} \right\rfloor = 0 \quad y \quad \sum_{i=3}^{\infty} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = 0$

Si $k_2 \geq q_n$, entonces se procede análogamente con los respectivos k_3 y r_3.

Este procedimiento llegará hasta un **entero positivo j** con el cual $k_{j+1} < q_n \leq k_j$. Pues q_n es primo natural, es mayor a uno, lo que implica que $n > k_1 > k_2 > k_3 > \ldots > k_j \geq q_n > k_{j+1} \geq 0$.

La función logarítmica identifica directamente ese valor j; pues, partiendo de su definición:

Sea **R** el conjunto de los números reales. N es subconjunto de R.

$$\exists x \in R : \left(\log_{q_n} n = x \ \leftrightarrow \ (q_n)^x = n\right)$$

$$Sea\ y \in R : \ (0 < y < 1)$$

Por las propiedades de los exponentes reales, se verifica qué:

$$(q_n)^{x-y} < (q_n)^x = n < (q_n)^{x+y}$$

Lo cual aplica a todos y cada uno de los valores de y en el intervalo indicado.

Observemos que:

$$x - 1 < x - y < x < x + y < x + 1$$

Y la parte entera de x verifica que:

$$x - 1 < \lfloor x \rfloor \leq x < \lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor \leq x + 1$$

Por lo que, la parte entera de x o es x o es un valor x – y, esto conduce a que:

$$(\lfloor x \rfloor = x) \rightarrow \left((q_n)^{\lfloor x \rfloor} = (q_n)^x = n < (q_n)^{\lfloor x \rfloor + 1}\right)$$

$$(\lfloor x \rfloor < x) \rightarrow \left((q_n)^{\lfloor x \rfloor} < (q_n)^x = n < (q_n)^{\lfloor x \rfloor + 1}\right)$$

$$(\lfloor x \rfloor \leq x) \rightarrow \left((q_n)^{\lfloor x \rfloor} \leq (q_n)^x = n < (q_n)^{\lfloor x \rfloor + 1}\right)$$

Esto es que:

$$\lfloor x \rfloor = \left\lfloor \log_{q_n} n \right\rfloor = j$$

Sea $k_0 = n$

$$Sean\ k_i, r_i \in N \cup \{0\}: (k_{i-1} = (k_i)(q_n) + r_i\ , r_i \in \{0, 1, 2, \ldots, q_n - 1\})\ para\ i\ \epsilon\ N$$

Los números que existen gracias al teorema de la división euclídea.

Aplicando la fórmula complementaria a estos números se verifica que:

$$\left\lfloor \frac{n}{q_n} \right\rfloor = \left\lfloor \frac{k_0}{q_n} \right\rfloor = \left\lfloor \frac{k_1 \cdot q_n + r_1}{q_n} \right\rfloor = k_1 + \left\lfloor \frac{r_1}{q_n} \right\rfloor = k_1 \ \ pues\ 0 \leq r_1 < q_n$$

$$\left\lfloor \frac{n}{{q_n}^2} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{k_0}{q_n} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{k_1}{q_n} \right\rfloor = \left\lfloor \frac{k_2 \cdot q_n + r_2}{q_n} \right\rfloor = k_2 + \left\lfloor \frac{r_2}{q_n} \right\rfloor = k_2 \;\; pues\; 0 \leq r_2 < q_n$$

$$Si\; se\; verifica\; \left\lfloor \frac{n}{{q_n}^{i+1}} \right\rfloor = \left\lfloor \frac{k_i}{q_n} \right\rfloor = k_{i+1} \;\; para\; todo\; i \leq l \;\in N\;,\;\; entonces$$

$$\left\lfloor \frac{n}{{q_n}^{l+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{n}{{q_n}^{l}} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{k_l}{q_n} \right\rfloor = \left\lfloor \frac{k_{l+1} \cdot q_n + r_{l+1}}{q_n} \right\rfloor = k_{l+1} + \left\lfloor \frac{r_{l+1}}{q_n} \right\rfloor = k_{l+1}$$

$$pues\; 0 \leq r_{l+1} < q_n$$

La primera igualdad es por la fórmula complementaria.

Con esto se demuestra que, para todo **i** natural, dados n y q_n, se verifica que:

$$\left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor = k_i \;\;,\;\; \left\lfloor \frac{n}{{q_n}^{i+1}} \right\rfloor = \left\lfloor \frac{\left\lfloor \frac{k_{i-1}}{{q_n}^{i}} \right\rfloor}{q_n} \right\rfloor = \left\lfloor \frac{k_i}{q_n} \right\rfloor = k_{i+1}$$

Siendo $k_0 = n$ y $k_i, r_i \in N \cup \{0\}$: $(k_{i-1} = (k_i)(q_n) + r_i\;, r_i \in \{0, 1, 2, \dots, q_n - 1\})\; para\; i \in N$ los números que existen debido al teorema de la división euclídea.

Además, existe un j_{q_n} natural para cada q_n tal que:

$$j_{q_n} = \lfloor \log_{q_n} n \rfloor \;, (q_n)^{j_{q_n}} \leq n < (q_n)^{j_{q_n}+1} \;, \left\lfloor \frac{n}{{q_n}^{j_{q_n}}} \right\rfloor \geq 1 \;, \left\lfloor \frac{n}{{q_n}^{j_{q_n}+l}} \right\rfloor = 0 \;\forall\; l \in N$$

$$\text{Pues } n < {q_n}^{j_{q_n}+1} < {q_n}^{j_{q_n}+2} < \cdots \;\; al\; ser\; q_n > 1$$

La suma infinita de partes enteras de la división de n entre las potencias naturales de q_n queda:

$$1 \leq \sum_{i=1}^{\infty} \left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor = \sum_{i=1}^{j_{q_n}} \left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor + \sum_{i=j_{q_n}+1}^{\infty} \left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor = \sum_{i=1}^{j_{q_n}} \left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor + 0 = \sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{{q_n}^{i}} \right\rfloor$$

$$\sum_{i=1}^{j_{q_n}} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = \sum_{i=1}^{j_{q_n}} \left\lfloor \frac{k_{q_n,\ i-1}}{q_n} \right\rfloor = \sum_{i=1}^{j_{q_n}} k_{q_n,\ i} \quad siendo\ n = k_{q_n,\ 0}\ , \lfloor \log_{q_n} n \rfloor = j_{q_n}$$

Es decir que:

$$1 \leq \sum_{i=1}^{\infty} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = \sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = \sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} k_{q_n,i} \quad con\ k_{q_n,0} = n\ , k_{q_n,i} = \left\lfloor \frac{k_{q_n,i-1}}{q_n} \right\rfloor \forall i \in N$$

Lo cual sucede para cualesquiera **n** entero mayor a 1 y $\mathbf{q_n}$ primo menor al n+1. Por lo que aplica en cada integrante de la fórmula base. Quedando las siguientes:

Presentaciones Prácticas de la Fórmula Base

Para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo mayor de los menores a n+1, sucede que:

$$(1)\ n! = 2^{\sum_{i=1}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor} \cdot 3^{\sum_{i=1}^{\lfloor \log_3 n \rfloor} \left\lfloor \frac{n}{3^i} \right\rfloor} \cdot 5^{\sum_{i=1}^{\lfloor \log_5 n \rfloor} \left\lfloor \frac{n}{5^i} \right\rfloor} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\lfloor \log_{p_n} n \rfloor} \left\lfloor \frac{n}{{p_n}^i} \right\rfloor}$$

$$(2)\ n! = 2^{\sum_{i=1}^{\lfloor \log_2 n \rfloor} k_{2,i}} \cdot 3^{\sum_{i=1}^{\lfloor \log_3 n \rfloor} k_{3,i}} \cdot 5^{\sum_{i=1}^{\lfloor \log_5 n \rfloor} k_{5,i}} \cdot \ldots \cdot {p_n}^{\sum_{i=1}^{\lfloor \log_{p_n} n \rfloor} k_{p_n,i}}$$

$$siendo\ k_{q_n,0} = n\ , k_{q_n,i} = \left\lfloor \frac{k_{q_n,i-1}}{q_n} \right\rfloor \forall i \in N\ , \forall q_n \in \{primos \leq n\}$$

IV. Algoritmos, Complejidades y Códigos en C#

4.1. Procedimientos Generales

4.1.1. Potencia de un Factor Primo de un Factorial

La segunda presentación de la fórmula base indica que, para todo entero positivo **n** mayor a 1, siendo $\mathbf{p_n}$ el número primo mayor de los menores a n+1, sucede:

$$n! = 2^{\sum_{i=1}^{\lfloor \log_2 n \rfloor} k_{2,i}} \cdot 3^{\sum_{i=1}^{\lfloor \log_3 n \rfloor} k_{3,i}} \cdot 5^{\sum_{i=1}^{\lfloor \log_5 n \rfloor} k_{5,i}} \cdot \ldots \cdot p_n^{\sum_{i=1}^{\lfloor \log_{p_n} n \rfloor} k_{p_n,i}}$$

$$siendo\ k_{q_n,0} = n\ , k_{q_n,i} = \left\lfloor \frac{k_{q_n,i-1}}{q_n} \right\rfloor \forall i \in N\ , \forall q_n \in \{primos \leq n\}$$

Con esto, la potencia de cada primo se puede obtener del siguiente modo, cuya complejidad es **O(log n)**:

```
public uint PotenciaPrimoDeFactorial(uint n, uint
primo)
{
    uint potencia = 0;
    uint k = n / primo;

    while (k > 0)
    {
        potencia += k;
        k /= primo;
    }
    return potencia;
}
```

Operaciones: 4 + 4 * [$\log_{primo}(n)$]
→ O(log n)
1
2

$\lfloor \log_{primo} n \rfloor + 1$

2
2

1

4.1.2. Primos Hasta Algún Natural Mayor a Uno

El modo que se propone para encontrar los primos que hay hasta un natural n > 1, que también son los anteriores a n + 1, tiene complejidad O(n log n) y consiste en que cada primo anule la posibilidad de ser primo de sus múltiplos en un arreglo de candidatos a primos:

1. Crear un arreglo de posiciones relativas a los impares hasta n, el cual tiene [(n+1)/2] integrantes, que se usará con valores true o false para indicar con si el impar 2*i+1, relativo a la posición i, es primo o no. Todos los impares anteriores a n + 1 comienzan siendo candidatos a primo. El 1 se reemplaza por el 2.
    - Con esto se vuelven implícitas las anulaciones de candidatos por el primo 2.
    - Los múltiplos mayores de cualquier primo impar p se encuentran en incrementos p de posición a partir de la posición del primo p.
    - Al recorrer el arreglo de indicadores de inicio a fin, el impar de la posición consultada será primo si aún es candidato a primo.
2. Recorrer el arreglo de indicadores de candidatos de inicio a fin. Deteniéndose a anular múltiplos sólo si es primo menor o igual a la raíz cuadrada de n, ya que estos primos bastan para anular no primos en el segmento de impares de interés en este procedimiento.
    - Quien aún sea candidato, es primo pues ningún primo anterior a él le anuló.
    - Quien ya no sea candidato no requiere mayor revisión.
    - Para saber si los múltiplos de un primo ya fueron todos anulados por las anulaciones de múltiplos de primos anteriores a él, basta con verificar si su cuadrado excede a n. Ante el primer primo que verifique esto, ya no será necesario hacer esta verificación en primos mayores; esto es que se puede parar el proceso de anulación.
    - Los primos que bastan para anular a todos los candidatos a primos hasta n, dejando necesariamente a candidatos que sí son primos, son cuyos cuadrados no sean mayores a n, pues para saber si un natural k > 1 es primo, basta que ningún natural mayor a uno le divida. El conjunto de estos números a revisar si le dividen o no, está integrado por números primos o compuestos. Los compuestos son divisibles entre primos, necesariamente entre primos anteriores a ellos; por lo que, si los primos que componen a un compuesto no dividen a k, entonces el compuesto no divide a k y si algún factor primo del compuesto sí divide a k, entonces no es necesario hacer más verificaciones de si k es primo, pues no lo es.
        - De modo que, para saber si k es primo, basta revisar si ningún primo anterior a él le divide.
        - Viendo esto de otra manera: de los primos anteriores a k, quienes le pueden dividir son los que estén en el conjunto de valores { k/2, k/3, k/4, …, k/(k-1) } cuando estas divisiones sean un número natural. Si la división de k entre algún compuesto es un natural, también lo será la división de k entre cada uno de los factores primos de ese compuesto, por lo que el conjunto de valores puede reducirse a los de k entre primos anteriores que den como resultado un número natural.

- Además, los primos mayores a la mitad de k seguro que no le dividen y si 2 no divide (exactamente) a k, entonces no hay un primo = k/2; si 3 no divide a k, tampoco hay un primo = k/3; esto es que si un primo p no divide a k, entonces k/p no es natural y el primo mayor de estos primos que disminuyen revisiones, es el primo q ≤ k/q en el que q y k/q tengan la mayor cercanía posible. Por lo que, si p * p > k, ya no se requiere revisar si le divide o no.
- Con esto, k es primo sólo si ningún primo menor o igual a la raíz cuadrad de k le divide.
- Extendiendo este resultado, los naturales anteriores a k son primos si ningún primo anterior a la raíz cuadrada de k les divide. Esto es que, no son primos los naturales mayores a uno y anteriores a k, si son múltiplos de algún primo cuyo cuadrado no es mayor que k.
- Por lo que, para obtener la lista de primos anteriores al natural n > 1, basta tomar al 2 y a los impares menores a n+1 que no sean múltiplos de los primos cuyo cuadrado no es mayor a n.
- Además, la anulación de múltiplos de cada primo comienza a requerirse hasta su cuadrado.

La declaración de variables y creación de arreglos es de complejidad lineal **O(n)**. Se dejará comentado un arreglo de candidatos desde el inicio para facilitar la comprensión del siguiente procedimiento. Dado un entero **n > 1**:

```
public uint[][] PrimosConPotenciaHasta(uint n)
{
    /* Complejidad O(n log n)
     *
     * Los primos < n + 1 son los enteros p que
     * 1 < p < (n+1) no divididos por los primos
     * menores a la raíz cuadrada de (n+1), por
     * lo que la anulación de múltiplos ya no es
     * necesaria a partir del primo p: p * p > n
     *
     * Impares como candidatos: 3, 5, ..., [(n+1)/2]
     * noEsPrimo[i] indica si 2*i+1 no es primo
     *
     * Cada primo p puede anular nuevos candidatos
     * no anulados a partir de p*p y avanzando de
     * p en p posiciones.
     * p*p es impar pues p es impar, por lo que
     * está como candidato a primo.
     *
     * p = 2*i+1, p*p = 4*i*i + 4*i + 1
     * p está en posicion i = p/2
     *
     */

    if (n < 2)
    {
        return null;
    }
```

```
            uint i, j;
            uint max = (1 + n) / 2;
            bool[] noEsPrimo = new bool[max];

            for (i = 0; i < max; i++)
            {
                noEsPrimo[i] = false;
            }
```

Recorrer los indicadores de los impares inicialmente candidatos se propone de la siguiente manera, considerando que la cantidad de primos anteriores a cualquier número x > 1 es menor o igual a **x / (log x)**:

```
uint posCuadradoPrimo = 4;

/* posCuadradoPrimo = 2i(i+1)
 * p*p <= n  ->  2i(i+1) <= (n-1)/2 < max
 *  n par -> n=2m, (n-1)/2=m-1, max=m
 *  n impar -> n=2m+1, (n-1)/2=m, max=m+1
 * por lo que 2i(i+1) <= (n-1)/2
 * equivale a 2i(i+1) < max
 */

for (i = 1; posCuadradoPrimo < max; i++)
{
    if (noEsPrimo[i])
    {
        continue;
    }
    // El impar 2*i+1 que llega aquí es primo
    uint primo = 2 * i + 1;
    posCuadradoPrimo = 2 * i * (i + 1);

   // Aseguramiento de anulación de múltiplos
   for (j = posCuadradoPrimo; j < max; j += primo)
   {
       noEsPrimo[j] = true;
   }
}
```

Se cumple el filtro if [n / (log n)] veces a lo más. Si se pasa el filtro, candidatos[i] es primo.

La verificación p*p > n es la condición de paro principal del primer ciclo for. Se actualiza el valor para el paro aquí y no en cada impar, pues los múltiplos que deben ser anulados comienzan hasta el cuadrado de un nuevo primo detectado. Mientras no se encuentre un nuevo primo para que anule múltiplos, no es necesario actualizar.

Las operaciones de estos filtros son de complejidad constante. El ciclo for inicial revisa a cada impar > 1 que sea ≤ √n. Por lo que la aplicación del filtro es de complejidad **O(√n)**.

Si la complejidad de las anulaciones a realizar es de complejidad mayor a O(√n), ella será la complejidad del proceso de anulación de candidatos hasta aquí.

Sobre la anulación de los múltiplos de cada primo p = 2 * i + 1 que ha pasado el filtro, nótese que:

- Los múltiplos anteriores a p^2 ya han sido anulados por primos anteriores.
- Es impar $p^2 = (2i+1)^2 = 4i(i+1)+1$ por lo que está considerado como candidato y tiene una posición, la cual es 2*i*(i+1).
- Puede hacerse es[j] = 0 siendo ya antes cero, por lo que esto es un aseguramiento de que no sea considerado primo el impar 2*j+1.
- La cantidad de anulaciones es a lo más $\left\lfloor \frac{\frac{n-p^2}{p}}{2} \right\rfloor = \left\lfloor \frac{n-p^2}{2\,p} \right\rfloor =: rev_i \leq \frac{n-p^2}{2\,p}$

$$\frac{n - p^2}{2\,p} = \frac{n - 4i * (i+1) - 1}{4i + 2} < \frac{n - 4i^2}{4i} = \frac{n}{4i} - i$$

La cantidad de primos que anulan múltiplos en este proceso ofrece las siguientes observaciones:

- Hay a lo más $\left\lfloor \frac{\sqrt{n}}{\log\sqrt{n}} \right\rfloor = \left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor$ primos que anularán candidatos. Aquellos cuyo cuadrado ≤ n.
- Cada uno de estos primos ordenados de menor a mayor es o igual o mayor a su impar correspondiente en orden de menor a mayor, lo que la cantidad de anulaciones de los primos es menor o igual a la cantidad de anulaciones de la fórmula rev_i con la i aplicada a los impares 2i+1. Con esto hay una cota útil para el cálculo de la complejidad de este procedimiento, pues estos impares no hacen menos anulaciones.

$$\sum_{i=1}^{\left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor} \left(\frac{n}{4i} - i\right) = \frac{n}{4} \sum_{i=1}^{\left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor} \frac{1}{i} - \sum_{i=1}^{\left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor} i \quad \rightarrow \quad O(n\log n - n) \rightarrow O(n\log n)$$

Pues:

$$O\left(\frac{n}{4} \sum_{i=1}^{\left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor} \frac{1}{i} - \sum_{i=1}^{\left\lfloor \frac{2\cdot\sqrt{n}}{\log n} \right\rfloor} i\right) \subset O\left(\frac{n}{4} \int_1^{\frac{2\cdot\sqrt{n}}{\log n}} \frac{1}{x} - \left(\frac{\sqrt{n}}{\log n}\right)^2\right) \subset O\left(n\log\frac{2\cdot\sqrt{n}}{\log n} - n\right) \subset O(n\log n)$$

- Para realizar cada anulación se tiene una complejidad constante O(1), gracias a que se ha elegido trabajar con un arreglo, cuyo inicio y fin es conocido, así como la longitud constante de sus bloques consecutivos. De modo que la complejidad de las anulaciones a asegurar es del orden O(n log n).

Así pues, la complejidad del filtro es O(√n) y la de hacer todas las anulaciones requeridas es O(n log n), por lo que la complejidad total para detectar los primos de un intervalo de impares menores a n + 1 es **O(n log n)**.

Ahora bien, las operaciones finales de conteo y puesta en un arreglo a entregar con los primos requeridos se realiza con complejidad **O(n)**:

```
// Conteo de primos
uint cantidad = 1;   // primo 2 ya contado
for (i = 1; i < max; i++)
{
    if (!noEsPrimo[i])
    {
        cantidad++;   // Conteo de primos
    }
}

uint[][] primos = new uint[cantidad][];

for (j = 0; j < cantidad; j++)
{
```

```
            primos[j] = new uint[2];
        }
        primos[0][0] = 2;
        primos[0][1] = 1;
        j = 1;

        for (i = 1; i < max; i++)
        {
            if (!noEsPrimo[i])
            {
                primos[j][0] = 2 * i + 1;
                primos[j][1] = 1;
                j++;
            }
        }

        return primos;
    }
```

De modo que la complejidad total para obtener los primos menores a n + 1 es O(n + n log n + n) = **O(n log n)**.

El procedimiento completo propuesto es:

```
public uint[][] PrimosConPotenciaHasta(uint n)
{
    /* Complejidad O(n log n)
     *
     * Los primos < n + 1 son los enteros p que
     * 1 < p < (n+1) no divididos por los primos
     * menores a la raíz cuadrada de (n+1), por
     * lo que la anulación de múltiplos ya no es
     * necesaria a partir del primo p: p * p > n
     *
     * Impares como candidatos: 3, 5, ...,
[(n+1)/2]
     * noEsPrimo[i] indica si 2*i+1 no es primo
     *
     * Cada primo p puede anular nuevos candidatos
     * no anulados a partir de p*p y avanzando de
     * p en p posiciones.
     * p*p es impar pues p es impar, por lo que
     * está como candidato a primo.
     *
     * p = 2*i+1, p*p = 4*i*i + 4*i + 1
     * p está en posicion i = p/2
     *
     */

    if (n < 2)
    {
        return null;
    }

    uint i, j;
    uint max = (1 + n) / 2;
    bool[] noEsPrimo = new bool[max];

    for (i = 0; i < max; i++)
    {
        noEsPrimo[i] = false;
    }

    uint posCuadradoPrimo = 4;

    /* posCuadradoPrimo = 2i(i+1)
     * p*p <= n  ->  2i(i+1) <= (n-1)/2 < max
     *  n par -> n=2m, (n-1)/2=m-1, max=m
     *  n impar -> n=2m+1, (n-1)/2=m, max=m+1
     * por lo que 2i(i+1) <= (n-1)/2
     * equivale a 2i(i+1) < max
     */
    for (i=1; posCuadradoPrimo<max; i++)
    {
        if (noEsPrimo[i])
        {
            continue;
        }
        // El impar 2*i+1 que llega aquí
es primo
        uint primo = 2 * i + 1;
        posCuadradoPrimo = 2*i*(i + 1);

        // Aseguramiento de anulación de
múltiplos
        for (j=posCuadradoPrimo; j<max;
j += primo)
        {
            noEsPrimo[j] = true;
        }
    }

    // Conteo de primos
    uint cantidad = 1;   // primo 2
    for (i = 1; i < max; i++)
    {
        if (!noEsPrimo[i])
        {
            cantidad++;
        }
    }

    uint[][] primos = new
uint[cantidad][];

    for (j = 0; j < cantidad; j++)
    {
        primos[j] = new uint[2];
    }
    primos[0][0] = 2;
    primos[0][1] = 1;
    j = 1;

    for (i = 1; i < max; i++)
    {
        if (!noEsPrimo[i])
        {
            primos[j][0] = 2 * i + 1;
            primos[j][1] = 1;
            j++;
        }
    }

    return primos;
}
```

4.1.3. Factorial en Factores Primos

La expresión en factores primos del factorial de n, siendo n natural > 1, ocupa del cálculo de los primos menores a n + 1, lo cual es del orden **O(n log n)**, y entrega a lo más (n / log n) primos. La potencia de cada uno de estos primos se obtiene con operaciones del orden O(log n) por lo que, la complejidad de la expresión en primos del factorial n es:

$$O\left(n\log n + \frac{n}{\log n}\cdot\log n\right) \rightarrow O(n\log n + n) \rightarrow O(n\ \log n)$$

Se tiene la posibilidad de operar con 0! y 1!, cuyo valor es 1. Lo que se entrega es 2^0, de modo que los cálculos requeridos y el diseño de datos se mantenga.

```
public uint[][] FactorialEnPrimos(uint n)
{
    uint[][] res;

    if (n < 2)   // 0!, 1! = 1 = 2^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
    }

    res = PrimosConPotenciaHasta(n);
    int c = res.Count();
    uint k;

    for (int i = 0; i < c; i++)
    {
        res[i][1] = 0;   // potencia
        k = n / res[i][0];

        while (k > 0)
        {
            res[i][1] += k;
            k /= res[i][0];
        }
    }

    return res;
}
```

4.1.4. Valor Numérico de Factores Primos

Se presenta un procedimiento para datos no negativos para que se pueda usar el mayor cálculo posible de una computadora con los tipos de datos estándar. Este procedimiento puede ser utilizado para cualquier expresión de factores naturales con su potencia natural indicada; sin embargo, la complejidad que se medirá es la relativa al usar los factores primos de un factorial, combinación, permutación o variación. Esta complejidad es **O(n log n)** y puede ser medida gracias a que hay una relación entre cada primo y su potencia del estilo n/p.

La fórmula base indica que, para todo entero positivo **n** mayor a 1, siendo $\mathbf{q_n}$ cualquier número primo menor a n+1, la potencia de q_n es:

$$\sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor = \sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} \left\lfloor \frac{n}{q_n^{\lfloor \log_{q_n} n \rfloor - i}} \right\rfloor = \sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} \left\lfloor \frac{n \cdot {q_n}^i}{q_n^{\lfloor \log_{q_n} n \rfloor}} \right\rfloor \leq \sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} \left\lfloor \frac{n \cdot {q_n}^i}{q_n^{\log_{q_n} n}} \right\rfloor = \sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} \lfloor {q_n}^i \rfloor$$

$$\sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} \lfloor {q_n}^i \rfloor = \sum_{i=0}^{\lfloor \log_{q_n} n \rfloor - 1} {q_n}^i = \frac{{q_n}^{\lfloor \log_{q_n} n \rfloor} - 1}{q_n - 1} < \frac{{q_n}^{\log_{q_n} n}}{q_n - 1} = \frac{n}{q_n - 1}$$

Con esto, la potencia de cada primo p es menor que n / (p-1). Como p ≤ n, la potencia es ≥ 1. De modo que, el valor numérico de cada primo elevado a su potencia respectiva ocupa menos de n/(p-1) multiplicaciones. Como hay a lo más (n / (log n)) primos menores a n + 1 y siendo $\mathbf{q_{n,i}}$ el primo menor a n+1 de posición i con orden de menor a mayor, se realizará una cantidad de multiplicaciones menor a:

$$\sum_{i=1}^{\lfloor \frac{n}{\log n} \rfloor} \frac{n}{q_{n,\ i} - 1} = n + \sum_{i=2}^{\lfloor \frac{n}{\log n} \rfloor} \frac{n}{q_{n,\ i} - 1} \leq n + \sum_{i=1}^{\lfloor \frac{n}{\log n} \rfloor - 1} \frac{n}{(2 * i + 1) - 1} = n + \frac{n}{2} \sum_{i=1}^{\lfloor \frac{n}{\log n} \rfloor - 1} \frac{1}{i}$$

Cuyo orden es:

$$O\left(n + \frac{n}{2} \sum_{i=1}^{\lfloor \frac{n}{\log n} \rfloor - 1} \frac{1}{i} \right) = O\left(n + \frac{n}{2} \int_1^{\frac{n}{\log n}} \frac{1}{x} \right) = O\left(n + n \log\left(\frac{n}{\log n} \right) \right) = O(n \log n)$$

De modo que es de complejidad **O(n log n)** cualquiera de los siguientes procedimientos, si se trabaja con factores primos provenientes de un factorial. También se tendrá esta complejidad si se trabaja con factores primos de la división de factoriales de una combinación, variación o permutación, ya que las potencias resultantes de cada primo serán menores o iguales a las correspondientes a n!, exceptuando las variaciones con reposición, que son $nVRr = n^r$, la cual, como se describe en 4.2.6., es del orden **$O(n^2 \log n)$** en la obtención de su valor.

Se presentan a continuación dos procedimientos para este cálculo. El segundo agrega la posibilidad de dar el cálculo con respecto a un módulo correspondiente. En este caso, se ha utilizado la propiedad que indica que el módulo de un producto es el producto del módulo de los factores. El resultado con módulo es útil para pruebas exhaustivas de las posibilidades computacionales y la validez del procedimiento para generar resultados verdaderos. Además, en C# el ulong va aplicando modulo max ulong = 18,446,744,073,709,551,615

```
public ulong ValorNumericoPrimosConPotencia(uint[][] fp)
{
    ulong val = 1;
    int c;   // c es cantidad
    c = fp.Count();

    for (int i = 0; i < c; i++)
    {
        for (int j = 0; j < fp[i][1]; j++)
        {
            val *= fp[i][0];
        }
    }

    return val;
}

public ulong ValorNumericoPrimosConPotenciaModulo(uint[][] fp, ulong mod)
{
    ulong val = 1;
    int c;   // c es cantidad

    c = fp.Count();

    for (int i = 0; i < c; i++)
    {
        for (int j = 0; j < fp[i][1]; j++)
        {
            val = (val * fp[i][0]) % mod;

            if (val == 0)
            {
                break;
            }
        }
    }

    return val;
}
```

4.2. Procedimientos Particulares de Combinatoria

4.2.1. Combinaciones sin Repetición

Las combinaciones sin repetición nCr son la cantidad de agrupaciones de r elementos distintos pudiendo elegir de entre n elementos. Una combinación es distinta a otra si al menos un integrante es distinto y no importa el orden o acomodo de los integrantes.

$$nCr = \binom{n}{r} = \frac{n!}{r!\,(n-r)!}, \qquad n \in \{1, 2, 3, \dots\}, \qquad r\epsilon\{0, 1, 2, \dots, n\}$$

De la primera presentación práctica de la fórmula base, la división de factoriales requeridos se convierte en una resta de exponentes. Como r y (n-r) son ≤ n, se pueden utilizar los primos del factorial n para el cálculo de todos los exponentes requeridos, ya que incluye los necesarios para r! y (n-r)! y no afecta al cálculo agregar más pues aportan cero a la suma de exponentes. El modo de encontrar cada exponente funciona adecuadamente incluso si r = 0 ó (n-r) = 0. El cálculo de exponentes está descrito en 4.1.1.

$$nCr = 1 \cdot \prod_{q_n\epsilon\{primos \le n\}} q_n^{\sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor - \sum_{i=1}^{\lfloor \log_{q_n} r \rfloor} \left\lfloor \frac{r}{{q_n}^i} \right\rfloor - \sum_{i=1}^{\lfloor \log_{q_n} (n-r) \rfloor} \left\lfloor \frac{n-r}{{q_n}^i} \right\rfloor}$$

Si n = 1, no hay proceso de cálculo de potencias y sólo queda el factor 1. La complejidad es **O(n log n)** por lo descrito en la sección 4.1.3. Factorial en Factores Primos, considerando que el cálculo agregado de exponentes para r y (n-r) sigue siendo de complejidad O(log n) en cada primo. Si se requiere el valor numérico de las combinaciones, se continua teniendo complejidad O(n log n), pues se realizarían dos procesos de orden O(n log n) por separado; en la sección 4.1.4. Valor Numérico de Factores Primos se detalla la complejidad de este procedimiento.

```
public uint[][] CombinacionesSinRepEnPrimos(uint n, uint r)
{
    uint[][] res;
    if (n < 2)   // 0C0, 1C0, 1C1 = 1 = 2^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
    }
    res = PrimosConPotenciaHasta(n);
    int c = res.Count();
    uint k;
```

```
        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
            k = n / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            k = r / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }

            k = (n - r) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }



    public ulong CombinacionesSinRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(CombinacionesSinRepEnPrimos(n, r));
    }
```

4.2.2. Combinaciones con Repetición

Las combinaciones con repetición nCRr son la cantidad de agrupaciones de r elementos que pueden repetirse pudiendo elegir de entre n elementos. Una combinación es distinta a otra si al menos un integrante es distinto y no importa el orden o acomodo de los integrantes.

$$nCRr = \binom{n+r-1}{r} = \frac{(n+r-1)!}{r!\,(n-1)!}, \qquad n \in \{1, 2, 3, \dots\}, \qquad r\epsilon\{0, 1, 2, \dots, n\}$$

De la primera presentación práctica de la fórmula base, la división de factoriales requeridos se convierte en una resta de exponentes. Como r y (n-1) son ≤ n, se pueden utilizar los primos del factorial (n+r-1) para el cálculo de todos los exponentes requeridos, ya que incluye los necesarios para r! y (n-r)! y no afecta al cálculo agregar más pues aportan cero a la suma de exponentes. El modo de encontrar cada exponente funciona adecuadamente incluso si r = 0 ó (n-1) = 0. El cálculo de exponentes está descrito en 4.1.1.

$$nCRr = 1 \cdot \prod_{q\epsilon\{primos \le n+r-1\}} q^{\sum_{i=1}^{\lfloor \log_q n \rfloor} \left\lfloor \frac{n}{q^i} \right\rfloor - \sum_{i=1}^{\lfloor \log_q r \rfloor} \left\lfloor \frac{r}{q^i} \right\rfloor - \sum_{i=1}^{\lfloor \log_q (n-1) \rfloor} \left\lfloor \frac{n-1}{q^i} \right\rfloor}$$

La complejidad es **O(n log n)** = O((2n)log(2n)) tomando en cuenta lo descrito en 4.2.1. y que n+r-1 < 2n. El valor numérico, análogamente a lo descrito en 4.2.1., es del orden O(n log n).

```
public uint[][] CombinacionesConRepEnPrimos(uint n, uint r)
{
    uint[][] res;
    if (n + r - 1 < 2)   // 0CR0, 1CR0, 1CR1 = 1 = 2^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
    }

    res = PrimosConPotenciaHasta(n + r - 1);
    int c = res.Count();
    uint k;

    for (int i = 0; i < c; i++)
    {
        res[i][1] = 0;   // potencia
        k = (n + r - 1) / res[i][0];

        while (k > 0)
        {
```

```
                res[i][1] += k;
                k /= res[i][0];
            }

            k = r / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }

            k = (n - 1) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }



    public ulong CombinacionesConRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(CombinacionesConRepEnPrimos(n, r));
    }
```

4.2.3. Permutaciones sin Repetición

Las permutaciones sin repetición Pn ó nP son la cantidad de modos distintos de ordenar o acomodar n elementos en un grupo de n elementos considerados distintos entre ellos, aunque sus valores coincidan.

$$nP = n! = 1 \cdot \prod_{q_n \epsilon \{primos \leq n\}} q_n^{\sum_{i=1}^{\left\lfloor \log_{q_n} n \right\rfloor} \left\lfloor \frac{n}{q_n{}^i} \right\rfloor}, \qquad n \in \{1, 2, 3, \ldots\}$$

En la sección 4.1.3. se encuentra una descripción del procedimiento para n! y su complejidad **O(n log n)**. La cual se mantiene si se requiere el valor numérico de la factorización en primos de n!.

```
public uint[][] PermutacionesSinRepEnPrimos(uint n)
{
    return FactorialEnPrimos(n);
}


public ulong PermutacionesSinRepeticion(uint n)
{
    return ValorNumericoPrimosConPotencia(FactorialEnPrimos(n));
}
```

4.2.4. Permutaciones con Repetición

Las permutaciones con repetición $nPRr_1, r_2, r_3, \ldots, r_m$ son la cantidad de modos distintos de ordenar o acomodar los m elementos de valores distintos de un grupo de n elementos, estos es que $r_1+r_2 + \ldots + r_m = n$. Cada elemento de distinto valor indica la cantidad de repetidos que tiene en su correspondiente r_j.

$$nPRr_1, r_2, \ldots, r_m = \frac{n!}{\prod_{j=1}^{m} r_j!} = 1 \cdot \prod_{q_n \epsilon \{primos \leq n\}} {q_n}^{\Sigma_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{{q_n}^i} \right\rfloor - \Sigma_{j=1}^{k} \Sigma_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \left\lfloor \frac{r_j}{{q_n}^i} \right\rfloor}$$

$$n \in \{1, 2, 3, \ldots\}, m \epsilon \{1, 2, \ldots, n\}, j \epsilon \{1, 2, \ldots, m\}, r_j \epsilon \{1, 2, \ldots, n\} : \sum_{j=1}^{m} r_j = n$$

Se puede apreciar que las permutaciones sin repetición son las permutaciones con repetición considerando m=n y cada r_j=1. Si n = 1, no hay proceso de cálculo de potencias y sólo queda el factor 1.

El procedimiento para este cálculo es análogo al de la sección 4.2.1. Combinaciones sin Repetición. En el cálculo de exponentes se agregan a lo más n procesos log n, pero su cantidad de operaciones no incrementa la complejidad O(log n) pues:

$$\sum_{j=1}^{m} \sum_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \left\lfloor \frac{r_j}{{q_n}^i} \right\rfloor = \sum_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \sum_{j=1}^{m} \left\lfloor \frac{r_j}{{q_n}^i} \right\rfloor \leq \sum_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \sum_{j=1}^{m} \frac{r_j}{{q_n}^i} = \sum_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \frac{1}{{q_n}^i} \sum_{j=1}^{m} r_j = \sum_{i=1}^{\lfloor \log_{q_n} r_j \rfloor} \frac{n}{{q_n}^i}$$

Lo cual es del orden O(log n). De modo que, el procedimiento sigue siendo **O(n log n)**.

```
public uint[][] PermutacionesConRepEnPrimos(uint n, uint[] r)
{
    uint[][] res;

    if (n < 2)   // 0PR0, 1PR1 = 1 = 2^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
    }

    res = PrimosConPotenciaHasta(n);
    uint k;
    int c = res.Count();
    int m = r.Count();
```

```
        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
            k = n / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            for (int j = 0; j < m; j++)
            {
                k = r[j] / res[i][0];

                while (k > 0)
                {
                    res[i][1] -= k;
                    k /= res[i][0];
                }
            }
        }

        return res;
    }



    public ulong PermutacionesConRepeticion(uint n, uint[] r)
    {
        return ValorNumericoPrimosConPotencia(PermutacionesConRepEnPrimos(n, r));
    }
```

4.2.5. Variaciones sin Repetición

Las variaciones sin repetición nVr son la cantidad de agrupaciones de r elementos considerados distintos pudiendo elegir de entre n elementos. Una variación es distinta a otra si al menos difiere un integrante o, coincidiendo los integrantes, difiere su orden o acomodo.

$$nVr = \frac{n!}{(n-r)!} = 1 \cdot \prod_{q_n \epsilon \{primos \leq n\}} q_n^{\sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{q_n{}^i} \right\rfloor - \sum_{i=1}^{\lfloor \log_{q_n} (n-r) \rfloor} \left\lfloor \frac{n-r}{q_n{}^i} \right\rfloor}$$

$$n \in \{1, 2, 3, \dots\}, \qquad r \epsilon \{0, 1, 2, \dots, n\}$$

Se puede apreciar que las variaciones sin repetición son las combinaciones sin repetición, pero ahora considerándolas distintas si también su acomodo es distinto.

Si n = 1, no hay proceso de cálculo de potencias y sólo queda el factor 1. La descripción del procedimiento es análogo a lo descrito en 4.2.1. Combinaciones sin Repetición. Asimismo, la complejidad es **O(n log n)**.

```
public uint[][] VariacionesSinRepEnPrimos(uint n, uint r)
{
    uint[][] res;

    if (n < 2)   // 0V0, 1V0, 1V1 = 1 = 1^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
    }

    res = PrimosConPotenciaHasta(n);
    uint k;
    int c = res.Count();

    for (int i = 0; i < c; i++)
    {
        res[i][1] = 0;   // potencia
        k = n / res[i][0];

        while (k > 0)
        {
            res[i][1] += k;
            k /= res[i][0];
        }
```

```
            k = (n - r) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }



    public ulong VariacionesSinRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(VariacionesSinRepEnPrimos(n, r));
    }
```

4.2.6. Variaciones con Repetición

Las variaciones con repetición nVRr son la cantidad de agrupaciones de r elementos considerados distintos, pudiendo elegir de entre n elementos y teniendo la posibilidad de repetir cualquiera ya seleccionado.

$$nVRr = n^r = (n\ V\ 1)^r = 1 \cdot \prod_{q_n \epsilon \{primos \leq n\}} q_n^{\,r*\sum_{i=1}^{\lfloor \log_{q_n} n \rfloor} \left\lfloor \frac{n}{q_n{}^i} \right\rfloor - r*\sum_{i=1}^{\lfloor \log_{q_n}(n-1) \rfloor} \left\lfloor \frac{n-1}{q_n{}^i} \right\rfloor}$$

$$n \in \{\ 1, 2, 3, \dots\ \}, \qquad r \epsilon \{0, 1, 2, \dots, n\}$$

El procedimiento propuesto es obtener los primos de n para multiplicar sus potencias por r, a través de los factores primos de la variación sin repetición de n elementos tomados de 1 en 1. La complejidad de esta variación nV1 es **O(n log n)** y se agrega una reducción de cálculos por la peculiaridad de tener n! / (n-1)!, ya que las potencias de sus primos tienden a sumar y restar lo mismo pronto, si no lo hicieran aún seguiría siendo de esta complejidad el siguiente procedimiento.

El cálculo del valor numérico de los factores primos obtenidos con un procedimiento O(n log n), tiene una complejidad mayor: retomando lo expuesto en 4.1.4. sobre la relación de cada primo con su potencia se llegó a que la cantidad total de multiplicaciones en la formación del producto total a entregar es del orden:

$$O\left(n + \frac{n}{2} \sum_{i=1}^{\left\lfloor \frac{n}{\log n} \right\rfloor - 1} \frac{1}{i}\right) = O\left(n + \frac{n}{2} \int_1^{\frac{n}{\log n}} \frac{1}{x}\right) = O\left(n + n \log\left(\frac{n}{\log n}\right)\right) = O(n \log n)$$

Esta cantidad de operaciones, ahora es a su vez multiplicada por r ≤ n, por lo que el orden sería **O(n² log n)**:

$$O\left(\left(n + n \log\left(\frac{n}{\log n}\right)\right)(n)\right) = O(n^2 + n^2 \log n) = O(n^2 \log n)$$

```
public uint[][] VariacionesConRepEnPrimos(uint n, uint r)
{
    uint[][] res;

    if (n < 2)   // 0V0, 1V0, 1V1 = 1 = 2^0
    {
        res = new uint[1][];
        res[0] = new uint[2];
        res[0][0] = 2;
        res[0][1] = 0;
        return res;
```

```
        }

        res = PrimosConPotenciaHasta(n);
        uint k1, k2;
        int c = res.Count();

        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;    // potencia
            k1 = n / res[i][0];
            k2 = (n - 1) / res[i][0];
            /* k1-k2=0,1 y es indicador
             *   de si el primo res[i][0]
             *   es factor o no de n.
             * A partir de que k1 = k2
             *   ya no hay aporte para
             *   la potencia.
             */

            while ( k1 > k2 )
            {
                k1 /= res[i][0];
                k2 /= res[i][0];
                res[i][1] += r;
            }
        }

        return res;
    }


    public ulong VariacionesConRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(VariacionesConRepEnPrimos(n, r));
    }
```

4.3. Otros Procedimientos

4.3.1. Factores Primos de un Natural Mayor a Uno

La descripción de este procedimiento del orden **O(n log n)** está en 4.2.6. Se utilizará para multiplicaciones o divisiones de un grupo de factores primos por algún entero positivo mayor a uno.

```
public uint[][] FactoresPrimosDeN(uint n)
{
    return VariacionesConRepEnPrimos(n, 1);
}
```

4.3.2. Algunas Operaciones con Factores Primos

Las operaciones entre factores primos de multiplicación y división, se reducen a sumas y restas de exponentes, los cuales, manteniendo el orden de menor a mayor de sus respectivos primos, hacen prescindible el conocer a los primos y basta con operar con sus potencias. La elevación a un exponente entero no negativo de un arreglo de factores primos es la multiplicación de cada uno de sus exponentes por el entero. Todas estas operaciones se realizan en complejidad lineal O(n), a más preciso, con complejidad O(n / log n) por la acotación utilizada de la cantidad de primos hasta n. Con respecto a un arreglo dado, la complejidad es lineal con respecto a la cantidad de elementos, sin contar las potencias.

La multiplicación de factores primos por algún entero mayor a 1 se propone primero convirtiendo a ese entero en su presentación en factores primos, lo cual es O(n log n) como se ha propuesto anteriormente, y luego en multiplicar ambos factores primos con complejidad considerada lineal O(n), por lo que la complejidad total es O(n log n). La división se hace con un procedimiento similar y su complejidad es como la de la multiplicación.

El código presentado a continuación conlleva la sobrecarga de operadores para poder hacer estas operaciones con objetos de factores primos y poder trabajar con números relativos a combinatoria cuyo valor tal vez no sea posible computacionalmente, pero sí tratable con factores primos.

```
        public static FactoresPrimos operator ^(FactoresPrimos fp1, uint k)
        {
            FactoresPrimos fpRes = new FactoresPrimos(fp1);

            for (int i = 0; i < fp1.Factores.Count(); i++)
            {
                fpRes.PotenciasParaOperar[i] *= k;
            }

            return fpRes;
        }

        public static FactoresPrimos operator *(FactoresPrimos fp1, FactoresPrimos fp2)
        {
            FactoresPrimos fpRes;
            int min = fp1.PotenciasParaOperar.Count();
            int max = fp2.PotenciasParaOperar.Count();

            if (max < min)
            {
                min = max;
                fpRes = new FactoresPrimos(fp1);

                for (int i = 0; i < min; i++)
                {
                    fpRes.PotenciasParaOperar[i] += fp2.PotenciasParaOperar[i];
                }
            }
            else
            {
                fpRes = new FactoresPrimos(fp2);

                for (int i = 0; i < min; i++)
                {
                    fpRes.PotenciasParaOperar[i] += fp1.PotenciasParaOperar[i];
                }
            }

            return fpRes;
        }

        public static FactoresPrimos operator *(FactoresPrimos fp1, uint[][] fp2)
        {
            FactoresPrimos fpRes;
            int min = fp1.PotenciasParaOperar.Count();
            int max = fp2.Count();

            if (max < min)
            {
                min = max;
                fpRes = new FactoresPrimos(fp1);

                for (int i = 0; i < min; i++)
                {
                    fpRes.PotenciasParaOperar[i] += fp2[i][1];
                }
```

```
            }
            else
            {
                fpRes = new FactoresPrimos(fp2);

                for (int i = 0; i < min; i++)
                {
                    fpRes.PotenciasParaOperar[i] += fp1.PotenciasParaOperar[i];
                }
            }

            return fpRes;
        }

        public static FactoresPrimos operator *(FactoresPrimos fp1, uint k)
        {
            if (k == 0)
            {
                return null;
            }
            if (k == 1)
            {
                return new FactoresPrimos(fp1);
            }
            return new FactoresPrimos(fp1 * fp1.FactoresPrimosDeN(k));
        }

        public static FactoresPrimos operator /(FactoresPrimos fp1, FactoresPrimos fp2)
        {
            FactoresPrimos fpRes = new FactoresPrimos();
            int min = fp1.PotenciasParaOperar.Count();
            int max = fp2.PotenciasParaOperar.Count();

            if (max < min)
            {
                min = max;
                fpRes = new FactoresPrimos(fp1);
            }
            else
            {
                fpRes = new FactoresPrimos(fp2);
                for (int i = 0; i < max; i++)
                {
                    fpRes.PotenciasParaOperar[i] *= -1;
                }
            }

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] = fp1.PotenciasParaOperar[i] -
fp2.PotenciasParaOperar[i];
            }

            return fpRes;
        }
```

```
public static FactoresPrimos operator /(FactoresPrimos fp1, uint[][] fp2)
{
    FactoresPrimos fpRes = new FactoresPrimos();
    int min = fp1.PotenciasParaOperar.Count();
    int max = fp2.Count();

    if (max < min)
    {
        min = max;
        fpRes = new FactoresPrimos(fp1);
    }
    else
    {
        fpRes = new FactoresPrimos(fp2);
        for (int i = 0; i < max; i++)
        {
            fpRes.PotenciasParaOperar[i] *= -1;
        }
    }

    for (int i = 0; i < min; i++)
    {
        fpRes.PotenciasParaOperar[i] = fp1.PotenciasParaOperar[i] - fp2[i][1];
    }

    return fpRes;
}

public static FactoresPrimos operator /(FactoresPrimos fp1, uint k)
{
    if (k == 0)
    {
        return null;
    }
    if (k == 1)
    {
        return new FactoresPrimos(fp1);
    }
    return new FactoresPrimos(fp1 / fp1.FactoresPrimosDeN(k));
}
```

V. Código en C#

```
using System.Linq;

namespace Combinatoria_FactorizacionEnPrimos
{
    public class FactoresPrimos
    {
        public uint[][] Factores { get; set; }
        public long[] PotenciasParaOperar { get; set; }

        /* Siendo k entero > 1 y fp un arreglo de factores primos,
         * las operaciones fp*fp, fp/fp, fp^k, fp*k, fp/k
         * sólo requieren de trabajar con un arreglo long de potencias
         * ya que, por su posición, harán referencia al mismo primo.
         */
        #region Constructores
        public FactoresPrimos()
        {

        }

        public FactoresPrimos(uint n)
        {
            if (n > 1)
            {
                Factores = FactoresPrimosDeN(n);
                int c = Factores.Count();
                PotenciasParaOperar = new long[c];

                for (int i = 0; i < c; i++)
                {
                    PotenciasParaOperar[i] = Factores[i][1];
                }
            }
        }

        public FactoresPrimos(FactoresPrimos fp)
        {
            int c = fp.Factores.Count();
            PotenciasParaOperar = new long[c];
            Factores = new uint[c][];

            for (int i = 0; i < c; i++)
            {
                Factores[i] = new uint[2];
            }

            for (int i = 0; i < c; i++)
            {
```

```
            Factores[i][0] = fp.Factores[i][0];
            Factores[i][1] = fp.Factores[i][1];
            PotenciasParaOperar[i] = fp.PotenciasParaOperar[i];
        }
    }

    public FactoresPrimos(uint[][] fp)
    {
        int c = fp.Count();
        PotenciasParaOperar = new long[c];
        Factores = new uint[c][];

        for (int i = 0; i < c; i++)
        {
            Factores[i] = new uint[2];
        }

        for (int i = 0; i < c; i++)
        {
            Factores[i][0] = fp[i][0];
            Factores[i][1] = fp[i][1];
            PotenciasParaOperar[i] = fp[i][1];
        }
    }

    #endregion

    #region Metodos Generales

    public string FactoresEnTexto(uint[][] fp)
    {
        string texto = "";
        int c = fp.Count();

        for (int i = 0; i < c; i++)
        {
            if (fp[i][1] == 0)
            {
                continue;
            }

            if (fp[i][1] == 1)
            {
                texto += "(" + fp[i][0] + ") ";
            }
            else
            {
                texto += "(" + fp[i][0] + "^" + fp[i][1] + ") ";
            }
        }

        return texto;
    }

    public string FactoresConPotenciasParaOperarEnTexto(FactoresPrimos fp)
    {
```

```
        string texto = "";
        int c = fp.Factores.Count();

        for (int i = 0; i < c; i++)
        {
            if (fp.PotenciasParaOperar[i] == 0)
            {
                continue;
            }

            if (fp.PotenciasParaOperar[i] == 1)
            {
                texto += "(" + fp.Factores[i][0] + ") ";
            }
            else
            {
                texto += "(" + fp.Factores[i][0] + "^" + fp.PotenciasParaOperar[i] + ") ";
            }
        }

        return texto;
    }

    public string PrimosHastaEnTexto(uint[][] fp)
    {
        int c = fp.Count();
        string texto = "";

        for (int i = 0; i < c - 1; i++)
        {
            texto += fp[i][0] + ", ";
        }

        texto += fp[c - 1][0];

        return texto;
    }

    public uint PotenciaPrimoDeFactorial(uint n, uint primo)
    {
        uint potencia = 0;
        uint k = n / primo;

        while (k > 0)
        {
            potencia += k;
            k /= primo;
        }
        return potencia;
    }

    public uint[] PrimosHasta(uint n)
    {
        /* Complejidad O(n log n)
         *
         * Los primos < n + 1 son los enteros p que
```

```
 * 1 < p < (n+1) no divididos por los primos
 * menores a la raíz cuadrada de (n+1), por
 * lo que la anulación de múltiplos ya no es
 * necesaria a partir del primo p: p * p > n
 *
 * Impares como candidatos: 3, 5, ..., [(n+1)/2]
 * noEsPrimo[i] indica si 2*i+1 no es primo
 *
 * Cada primo p puede anular nuevos candidatos
 * no anulados a partir de p*p y avanzando de
 * p en p posiciones.
 * p*p es impar pues p es impar, por lo que
 * está como candidato a primo.
 *
 * p = 2*i+1, p*p = 4*i*i + 4*i + 1
 * p está en posicion i = p/2
 *
 */
if (n < 2)
{
    return null;
}

uint i, j;
uint max = (1 + n) / 2;
bool[] noEsPrimo = new bool[max];

for (i = 0; i < max; i++)
{
    noEsPrimo[i] = false;
}

uint posCuadradoPrimo = 4;

/* posCuadradoPrimo = 2i(i+1)
 * p*p <= n  ->  2i(i+1) <= (n-1)/2 < max
 *  n par -> n=2m, (n-1)/2=m-1, max=m
 *  n impar -> n=2m+1, (n-1)/2=m, max=m+1
 * por lo que 2i(i+1) <= (n-1)/2
 * equivale a 2i(i+1) < max
 */

for (i = 1; posCuadradoPrimo < max; i++)
{
    if (noEsPrimo[i])
    {
        continue;
    }
    // El impar 2*i+1 que llega aquí es primo
    uint primo = 2 * i + 1;
    posCuadradoPrimo = 2 * i * (i + 1);

    // Aseguramiento de anulación de múltiplos
    for (j = posCuadradoPrimo; j < max; j += primo)
    {
        noEsPrimo[j] = true;
```

```
            }
        }

        // Conteo de primos
        uint cantidad = 1;   // primo 2 ya contado
        for (i = 1; i < max; i++)
        {
            if (!noEsPrimo[i])
            {
                cantidad++;   // Conteo de primos
            }
        }

        uint[] primos = new uint[cantidad];

        primos[0] = 2;
        j = 1;

        for (i = 1; i < max; i++)
        {
            if (!noEsPrimo[i])
            {
                primos[j] = 2 * i + 1;
                j++;
            }
        }

        return primos;
    }

    public uint[][] PrimosConPotenciaHasta(uint n)
    {
        /* Complejidad O(n log n)
         *
         * Los primos < n + 1 son los enteros p que
         * 1 < p < (n+1) no divididos por los primos
         * menores a la raíz cuadrada de (n+1), por
         * lo que la anulación de múltiplos ya no es
         * necesaria a partir del primo p: p * p > n
         *
         * Impares como candidatos: 3, 5, ..., [(n+1)/2]
         * noEsPrimo[i] indica si 2*i+1 no es primo
         *
         * Cada primo p puede anular nuevos candidatos
         * no anulados a partir de p*p y avanzando de
         * p en p posiciones.
         * p*p es impar pues p es impar, por lo que
         * está como candidato a primo.
         *
         * p = 2*i+1, p*p = 4*i*i + 4*i + 1
         * p está en posicion i = p/2
         *
         */

        if (n < 2)
        {
```

```
    return null;
}

uint i, j;
uint max = (1 + n) / 2;
bool[] noEsPrimo = new bool[max];

for (i = 0; i < max; i++)
{
    noEsPrimo[i] = false;
}

uint posCuadradoPrimo = 4;

/* posCuadradoPrimo = 2i(i+1)
 * p*p <= n  ->  2i(i+1) <= (n-1)/2 < max
 *  n par -> n=2m, (n-1)/2=m-1, max=m
 *  n impar -> n=2m+1, (n-1)/2=m, max=m+1
 * por lo que 2i(i+1) <= (n-1)/2
 * equivale a 2i(i+1) < max
 */

for (i = 1; posCuadradoPrimo < max; i++)
{
    if (noEsPrimo[i])
    {
        continue;
    }
    // El impar 2*i+1 que llega aquí es primo
    uint primo = 2 * i + 1;
    posCuadradoPrimo = 2 * i * (i + 1);

    // Aseguramiento de anulación de múltiplos
    for (j = posCuadradoPrimo; j < max; j += primo)
    {
        noEsPrimo[j] = true;
    }
}

// Conteo de primos
uint cantidad = 1;   // primo 2 ya contado
for (i = 1; i < max; i++)
{
    if (!noEsPrimo[i])
    {
        cantidad++;   // Conteo de primos
    }
}

uint[][] primos = new uint[cantidad][];

for (j = 0; j < cantidad; j++)
{
    primos[j] = new uint[2];
}
primos[0][0] = 2;
```

```
            primos[0][1] = 1;
            j = 1;

            for (i = 1; i < max; i++)
            {
                if (!noEsPrimo[i])
                {
                    primos[j][0] = 2 * i + 1;
                    primos[j][1] = 1;
                    j++;
                }
            }

            return primos;
        }

        public uint[][] FactorialEnPrimos(uint n)
        {
            uint[][] res;

            if (n < 2)   // 0!, 1! = 1 = 2^0
            {
                res = new uint[1][];
                res[0] = new uint[2];
                res[0][0] = 2;
                res[0][1] = 0;
                return res;
            }

            res = PrimosConPotenciaHasta(n);
            int c = res.Count();
            uint k;

            for (int i = 0; i < c; i++)
            {
                res[i][1] = 0;   // potencia
                k = n / res[i][0];

                while (k > 0)
                {
                    res[i][1] += k;
                    k /= res[i][0];
                }
            }

            return res;
        }

        public ulong ValorNumericoPrimosConPotencia(uint[][] fp)
        {
            ulong val = 1;
            int c;   // c es cantidad

            c = fp.Count();

            for (int i = 0; i < c; i++)
```

```
        {
            for (int j = 0; j < fp[i][1]; j++)
            {
                val *= fp[i][0];
            }
        }

        return val;
    }

    public ulong ValorNumericoPrimosConPotenciaModulo(uint[][] fp, ulong mod)
    {
        ulong val = 1;
        int c;   // c es cantidad

        c = fp.Count();

        for (int i = 0; i < c; i++)
        {
            for (int j = 0; j < fp[i][1]; j++)
            {
                val = (val * fp[i][0]) % mod;

                if (val == 0)
                {
                    break;
                }
            }
        }

        return val;
    }

    public uint[][] FactoresPrimosDeN(uint n)
    {
        return VariacionesConRepEnPrimos(n, 1);
    }

    #endregion

    #region Combinatoria

    public uint[][] CombinacionesSinRepEnPrimos(uint n, uint r)
    {
        uint[][] res;
        if (n < 2)   // 0C0, 1C0, 1C1 = 1 = 2^0
        {
            res = new uint[1][];
            res[0] = new uint[2];
            res[0][0] = 2;
            res[0][1] = 0;
            return res;
        }
        res = PrimosConPotenciaHasta(n);
        int c = res.Count();
        uint k;
```

```
        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
            k = n / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            k = r / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }

            k = (n - r) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }

    public ulong CombinacionesSinRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(CombinacionesSinRepEnPrimos(n, r));
    }

    public uint[][] CombinacionesConRepEnPrimos(uint n, uint r)
    {
        uint[][] res;
        if (n + r - 1 < 2)   // 0CR0, 1CR0, 1CR1 = 1 = 2^0
        {
            res = new uint[1][];
            res[0] = new uint[2];
            res[0][0] = 2;
            res[0][1] = 0;
            return res;
        }

        res = PrimosConPotenciaHasta(n + r - 1);
        int c = res.Count();
        uint k;

        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
```

```
            k = (n + r - 1) / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            k = r / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }

            k = (n - 1) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }

    public ulong CombinacionesConRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(CombinacionesConRepEnPrimos(n, r));
    }

    public uint[][] PermutacionesSinRepEnPrimos(uint n)
    {
        return FactorialEnPrimos(n);
    }

    public ulong PermutacionesSinRepeticion(uint n)
    {
        return ValorNumericoPrimosConPotencia(FactorialEnPrimos(n));
    }

    public uint[][] PermutacionesConRepEnPrimos(uint n, uint[] r)
    {
        uint[][] res;

        if (n < 2)   // 0PR0, 1PR1 = 1 = 2^0
        {
            res = new uint[1][];
            res[0] = new uint[2];
            res[0][0] = 2;
            res[0][1] = 0;
            return res;
        }
```

```
        res = PrimosConPotenciaHasta(n);
        uint k;
        int c = res.Count();
        int m = r.Count();

        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
            k = n / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            for (int j = 0; j < m; j++)
            {
                k = r[j] / res[i][0];

                while (k > 0)
                {
                    res[i][1] -= k;
                    k /= res[i][0];
                }
            }
        }

        return res;
    }

    public ulong PermutacionesConRepeticion(uint n, uint[] r)
    {
        return ValorNumericoPrimosConPotencia(PermutacionesConRepEnPrimos(n, r));
    }

    public uint[][] VariacionesSinRepEnPrimos(uint n, uint r)
    {
        uint[][] res;

        if (n < 2)   // 0V0, 1V0, 1V1 = 1 = 1^0
        {
            res = new uint[1][];
            res[0] = new uint[2];
            res[0][0] = 2;
            res[0][1] = 0;
            return res;
        }

        res = PrimosConPotenciaHasta(n);
        uint k;
        int c = res.Count();

        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;   // potencia
```

```
            k = n / res[i][0];

            while (k > 0)
            {
                res[i][1] += k;
                k /= res[i][0];
            }

            k = (n - r) / res[i][0];

            while (k > 0)
            {
                res[i][1] -= k;
                k /= res[i][0];
            }
        }

        return res;
    }

    public ulong VariacionesSinRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(VariacionesSinRepEnPrimos(n, r));
    }

    public uint[][] VariacionesConRepEnPrimos(uint n, uint r)
    {
        uint[][] res;

        if (n < 2)   // 0V0, 1V0, 1V1 = 1 = 2^0
        {
            res = new uint[1][];
            res[0] = new uint[2];
            res[0][0] = 2;
            res[0][1] = 0;
            return res;
        }

        res = PrimosConPotenciaHasta(n);
        uint k1, k2;
        int c = res.Count();

        for (int i = 0; i < c; i++)
        {
            res[i][1] = 0;    // potencia
            k1 = n / res[i][0];
            k2 = (n - 1) / res[i][0];
            /* k1-k2=0,1 y es indicador
             *   de si el primo res[i][0]
             *   es factor o no de n.
             * A partir de que k1 = k2
             *   ya no hay aporte para
             *   la potencia.
             */

            while ( k1 > k2 )
```

```
            {
                k1 /= res[i][0];
                k2 /= res[i][0];
                res[i][1] += r;
            }
        }

        return res;
    }

    public ulong VariacionesConRepeticion(uint n, uint r)
    {
        return ValorNumericoPrimosConPotencia(VariacionesConRepEnPrimos(n, r));
    }
    #endregion

    #region Algunas operaciones (fp, fp) y (fp, entero > 0)

    public long[] PasoDePotenciasUintALong(uint[][] fp)
    {
        int c = fp.Count();
        long[] pl = new long[c];

        for (uint i = 0; i < c; i++)
        {
            pl[i] = fp[i][1];
        }

        return pl;
    }

    public static FactoresPrimos operator ^(FactoresPrimos fp1, uint k)
    {
        FactoresPrimos fpRes = new FactoresPrimos(fp1);

        for (int i = 0; i < fp1.Factores.Count(); i++)
        {
            fpRes.PotenciasParaOperar[i] *= k;
        }

        return fpRes;
    }

    public static FactoresPrimos operator *(FactoresPrimos fp1, FactoresPrimos fp2)
    {
        FactoresPrimos fpRes;
        int min = fp1.PotenciasParaOperar.Count();
        int max = fp2.PotenciasParaOperar.Count();

        if (max < min)
        {
            min = max;
            fpRes = new FactoresPrimos(fp1);

            for (int i = 0; i < min; i++)
            {
```

```
                fpRes.PotenciasParaOperar[i] += fp2.PotenciasParaOperar[i];
            }
        }
        else
        {
            fpRes = new FactoresPrimos(fp2);

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] += fp1.PotenciasParaOperar[i];
            }
        }

        return fpRes;
    }

    public static FactoresPrimos operator *(FactoresPrimos fp1, uint[][] fp2)
    {
        FactoresPrimos fpRes;
        int min = fp1.PotenciasParaOperar.Count();
        int max = fp2.Count();

        if (max < min)
        {
            min = max;
            fpRes = new FactoresPrimos(fp1);

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] += fp2[i][1];
            }
        }
        else
        {
            fpRes = new FactoresPrimos(fp2);

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] += fp1.PotenciasParaOperar[i];
            }
        }

        return fpRes;
    }

    public static FactoresPrimos operator *(FactoresPrimos fp1, uint k)
    {
        if (k == 0)
        {
            return null;
        }
        if (k == 1)
        {
            return new FactoresPrimos(fp1);
        }
        return new FactoresPrimos(fp1 * fp1.FactoresPrimosDeN(k));
```

```
        }

        public static FactoresPrimos operator /(FactoresPrimos fp1, FactoresPrimos fp2)
        {
            FactoresPrimos fpRes = new FactoresPrimos();
            int min = fp1.PotenciasParaOperar.Count();
            int max = fp2.PotenciasParaOperar.Count();

            if (max < min)
            {
                min = max;
                fpRes = new FactoresPrimos(fp1);
            }
            else
            {
                fpRes = new FactoresPrimos(fp2);
                for (int i = 0; i < max; i++)
                {
                    fpRes.PotenciasParaOperar[i] *= -1;
                }
            }

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] = fp1.PotenciasParaOperar[i] -
fp2.PotenciasParaOperar[i];
            }

            return fpRes;
        }

        public static FactoresPrimos operator /(FactoresPrimos fp1, uint[][] fp2)
        {
            FactoresPrimos fpRes = new FactoresPrimos();
            int min = fp1.PotenciasParaOperar.Count();
            int max = fp2.Count();

            if (max < min)
            {
                min = max;
                fpRes = new FactoresPrimos(fp1);
            }
            else
            {
                fpRes = new FactoresPrimos(fp2);
                for (int i = 0; i < max; i++)
                {
                    fpRes.PotenciasParaOperar[i] *= -1;
                }
            }

            for (int i = 0; i < min; i++)
            {
                fpRes.PotenciasParaOperar[i] = fp1.PotenciasParaOperar[i] - fp2[i][1];
            }
```

```
            return fpRes;
        }

        public static FactoresPrimos operator /(FactoresPrimos fp1, uint k)
        {
            if (k == 0)
            {
                return null;
            }
            if (k == 1)
            {
                return new FactoresPrimos(fp1);
            }
            return new FactoresPrimos(fp1 / fp1.FactoresPrimosDeN(k));
        }

        #endregion
    }
}
```