Erick Vargas

Club algoritmia (ESCOM)

Club algoritmia (ESCOM)

Contents

1 Búsqueda binaria

1.1 Planetas

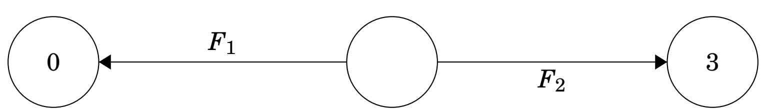
Tienes un conjunto de planetas y te dan las posiciones de estos planetas, quieres colocar un meteorito en cualquier lugar entre los planetas, el meteorito debe quedarse en su lugar y no ser atraido por la fuerza gravitacional de los planetas. Sabemos de física básica que la sumatoria de todas las fuerzas debe ser igual con cero. Sabemos que:

- $F_1 = F_2$ esta en equilibrio el sitema
- $F_1 > F_2$
- $F_1 < F_2$

Tenemos como información la siguiente fórmula:

$$\frac{1}{|X_i - M|}$$

Ejemplo:



Si utilizamos las fórmulas calculamos el valor de F_1, F_2

$$F_1 = \frac{1}{|0-1.5|} = \frac{1}{1.5}$$

y

$$F_2 = \frac{1}{|3-1.5|} = \frac{1}{1.5}$$

Podemos ver que si restamos ambas fuerzas efectivamente el sistema está en equilibrio.

Pero ¿Qué pasa si tenemos más de dos planetas? Tenemos n - 1 soluciones ya que debemos hacer binaria entre cada pareja de planetas, si tomamos solo el planeta origen y el más alejado vamos a perder muchas soluciones.

Recordemos que el meteorito está en una posición válida la suma de las fuerzas debe ser igual con cero:

$$F_1 + F_2 + F_3 + F_3 = 0$$

¿Que ocurre si la suma de fuerzas no es igual con cero? Debemos de mover nuestra binaria. Si la suma es mayor que cero nos movemos a la izquierda, si es menor que cero nos movemos a la derecha.

Tomar en cuenta que si eliminas de la fórmula el valor absoluto no necesitaremos comprobar la posición del planeta en la que estamos. Si es negativo el resultado de el elemento de la fórmula que estamos haciendo entonces la fuerza que tenemos esta a la derecha en otro caso está a la izquierda.

```
#include <bits/stdc++.h>

using namespace std;
vector< double > planets;
int n;
//\sum^{ n }_{ 0 }{ \frac{ 1 }{ X_{i} - M } }
double SumOfForces( double middle ){
    double sum = 0.0;
    for( int i = 0; i < n; i++ ){
        sum += 1 / ( planets[ i ] - middle );
    }
    return sum;
}

int main(){
    ios::sync_with_stdio( false );
    cout.tie( nullptr );
    cin.tie( nullptr );
    cin >> n;
    planets.resize( n );
    for( int i = 0; i < n; i++)
        cin >> planets[ i ];
    sort( planets.begin(), planets.end() );
    cout << n - 1 << endl;
    for( int i = 0; i < n - 1; i++ ){
        double begin = planets[ i ];
        double end = planets[ i + 1 ];
        double middle;
        for( int i = 0; i < 25; i++ ){
            middle = ( begin + end ) / 2;
            if( SumOfForces( middle ) < 0 ){
                begin = middle;
            } else {
                end = middle;
            }
        }
        cout << fixed;
        cout << setprecision( 3 );
        cout << middle << " ";
    }
    cout << endl;
    return 0;
}
```

2 Programación dinámica

Partimos de una solución recursiva bruta y podemos hacer uso de una función de memorización. El ejemplo más clásico es el de Fibonacci

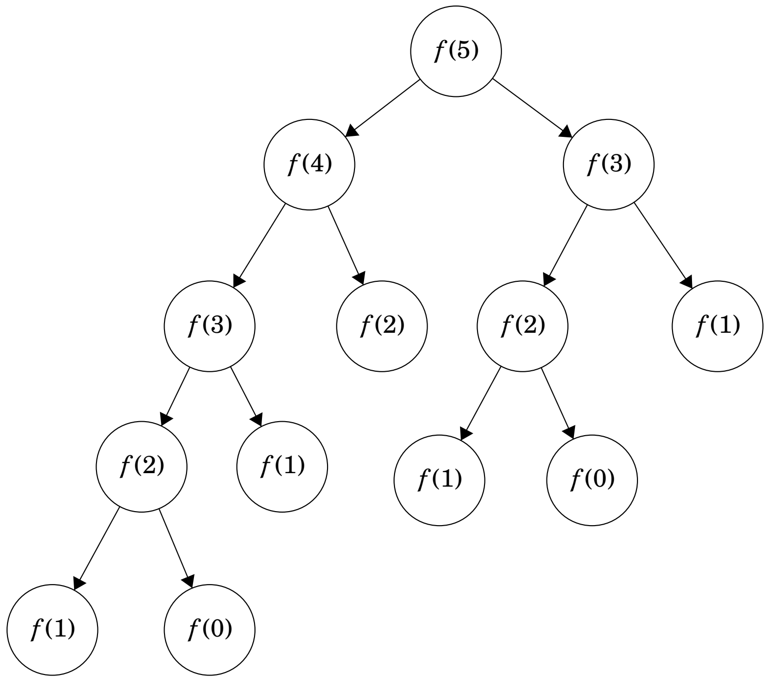
2.1 Fibonacci

$$fibonacci(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & , n \geq 2 \end{cases}$$

En código

```
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    return fibonacci( n - 1 ) * fibonacci( n - 2 );
}
```

Si dibujamos las llamadas recursivas como un árbol tenemos lo siguiente



Como podemos observar hay llamadas recursivas que se repiten varias veces como ejemplo tenemos la llamada recursiva con un valor de 3, o 2. ¿Podemos evitar esto?

```
int memoria[ 100000 ];
.
.
.
int fibonacci( int n ){
    if( n == 0 )
        return 0;
    if( n == 1 )
        return 1;
    if( memoria[ n ] != -1 )
        return memoria[ n ];
    return memoria[ n ] = ( fibonacci( n - 1 ) * fibonacci( n - 2 ) );
}

int main(){
    memset( memoria, -1, sizeof(memoria) );
    .
    .
    .
}
```

La complejidad la podemos calcular viendo los estados, en nuestro caso es el tamaño de la memoria, por ejemplo si llamamos a fibonacci de 10 siempre tenemos el mismo resultado, por tanto multiplicamos los estados por la complejidad de la función en nuestro caso es $10^5 * constante$

2.2 Coeficiente binomial

$$Coef\_Bin(n,k) = \begin{cases} 1 & , n = k \\ 1 & , k = 0 \\ Coef\_Bin(n-1,k-1) + Coef\_Bin(n-1,k) & , k \leq n \end{cases}$$

Si programamos esta función tenemos lo siguiente:

```
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    return ( coef_bin( n - 1, k - 1 ) + coef_bin( n - 1, k ) );
}
```

Si aplicamos DP

```
memoria[ 1000 ][ 1000 ];
.
.
.
int coef_bin( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    if( mem[ n ][ k ] != -1 )
        return memoria[ n ][ k ];
    return memoria[ n ][ k ] = ( coef_bin( n - 1, k - 1 ) + coef_bin( n - 1, k ) );
}
```

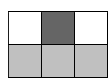
En complejidad tenemos $O(n*k)$ y sin la función de memorización tenemos $O(2^n)$

2.3 Problema

Dado un grid de nxm, cada casilla tiene un número. Obtener un camino de la fila 1 a la fila n con suma máxima, ejemplo:

3	5	10
6	4	3
2	1	0

Como restricciones tenemos que $n,m \leq 10^3$ y además $A_{i,j} \leq 10^6$ Además solo nos podemos mover de la siguiente manera. Supongamos que estamos en el recuadro con el color gris más oscuro, estando en esa posición podemos movernos a cualquiera de los tres rectangulos con color gris más claro, es decir, podemos movernos hacía $(f+1,c),(f+1,c-1),(f+1,c+1)$



2.3.1 Solución

```
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido
    if( c < 0 || c >= m )
        return MIN_INT //MIN_INT es como un menos infinito
    //Caso base
    if( f == n )
        return grid[ f ][ c ];
    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado caso podemos usar vector es
        decir ponemos max({A, B, C})
    return max( A, max(B, C) ) + grid[ f ][ c ];
}
```

Ahora bien ¿Cómo reducimos la complejidad de el código anterior? Debemos aplicar programación dinámica, primeramente ¿Cuál es el tamaño máximo de nuestra función de memoria? en nuestro caso es 1005 * 1005

```
int memoria[ 1005 ][ 1005 ];
.
.
```

```
.
int max_sum( int f, int c ){
    //Primero comprobamos si es un caso válido (que no esté fuera de rango)
    if( c < 0 || c >= m )
        return -1 //-1 es un valor bandera que identifica si ya visitamos o no la
            casilla
    //Caso base
    if( f == n )
        return grid[ f ][ c ];

    if( memoria[ f ][ c ] != -1 )
        return memoria[ f ][ c ]

    int A = max_sum( f + 1, c - 1);
    int B = max_sum( f + 1, c );
    int C = max_sum( f + 1, c + 1 );
    //No exite una función max para 3 elementos, en dado caso podemos usar vector es
        decir ponemos max({A, B, C})
    return memoria[ f ][ c ] = max( A, max(B, C) ) + grid[ f ][ c ];
}
```

2.4 Problema

Del ejercicio anterior que sucede si podemos añadir ¿números negativos? Podemos hacer uso de otro arreglo auxiliar de booleanos, en el que marcamos si ya visitamos o no una casilla

3

Problemas clásicos de DP

3.1 Mochilas

Vas a una tienda y quieres comprar algunos productos, digamos que hay n productos y cada producto tiene un precio y tenemos M pesos para gastar. Queremos comprar la mayor cantidad de productos gastando lo más posible.
Restricciones:

- N productos $N \leq 1000$
- M Pesos $M \leq 1000$
- $A_i \leq 1000$

Ejemplo:
N = 3, M = 8, y tenemos los productos A = 3, 4, 6 nuestro solución es comprar 3, 4 el precio es 7 y nos llevamos 4 productos.

Para poder solucionar este problema debemos de calcular el caso de probar y no probar el i-ésimo elemento. Es decir generamos todos los posibles subconjuntos del arreglo:

- ()
- (3)
- (3, 4)
- (3, 6)
- (4)
- (4, 6)
- (6)
- (3, 4, 6)

```
int max_sum( int indice, int pesos_restantes ){
    //Validación, n es el número de elementos del arreglo A
    if( indice >= n )
        return 0;
    //Como no es una respuesta válida para descartarla regresamos un menos infinito
    if( pesos_restantes < 0)
        return MIN_INT;
    //Recursivamente puedo tomar o no tomar un elemento
```

```
        int tomar = max_sum( indice + 1, pesos - A[ indice ]) + A[ indice ];
        int no_tomar = max_sum( indice + 1, pesos);
        return max( tomar , no_tomar );
    }
```

Esta solución es la más bruta si queremos aplicar programación dinámica ¿Cuáles son los estados que tenemos? En este caso tendremos los índices y la cantidad de pesos restantes. Es decir:

```
    .
    .
    .
    int memoria[ 1005 ][ 1005 ];
    bool visitado[ 1005 ][ 1005 ];
    .
    .
    .
    int max_sum( int indice, int pesos_restantes ){
        //Validación, n es el número de elementos del arreglo A
        if( indice >= n )
            return 0;
        //Como no es una respuesta válida para descartarla regresamos un menos infinito
        if( pesos_restantes < 0)
            return MIN_INT;

        if( visitado[ indice ][ pesos_restantes ] )
            return memoria[ indice ][ pesos_restantes ];

        //Recursivamente puedo tomar o no tomar un elemento
        int tomar = max_sum( indice + 1, pesos - A[ indice ]) + A[ indice ];
        int no_tomar = max_sum( indice + 1, pesos);
        //Como no se ha visitado el cálculo actual lo marcamos como visitado
        visitado[ indice ][ pesos_restantes ] = true;
        //Almacenamos el valor calculado en nuestra memoria
        return memoria[ indice ][ pesos_restantes ] = max( tomar , no_tomar );
    }
```

Con esta solución tenemos una complejidad de $O(n^2)$ mientras que la solución bruta tiene una complejidad de $O(2^n)$

3.2 Mochila clásica

Queremos meter N piedras a una mochila cuya capacidad es de M y cada piedra tiene un valor V, queremos meter la mayor cantidad de piedras respetando que el peso límite soportado por la mochila no se exceda y se maximice el valor de las piedras dentro de la mochila
Restricciones:

- N piedras $N \leq 1000$
- M capacidad de la mochila $M \leq 1000$
- V_i valor de la piedra $V_i \leq 10^9$
- W_i peso de la piedra $W_i \leq 1000$

Ejemplo:
N = 3, M = 50;
V = [60, 100, 120]
W = [10, 20, 30]
Para este caso tenemos como solución tomar la segunda y tercer piedra teniendo un valor de 220 y un peso de 50, el cual puede soportar perfectamente nuestra mochila.

Podemos encontrar una solución siguiendo más o menos lo mismo que el ejercico anterior, vamos a tratar de maximizar el valor tomando o no tomando el i-ésimo peso

```
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
        return MIN_INT;
    //Si lo tomo cuál es el valor máximo que podemos tener con la respuesta actual?
    int tomar = sum_max( indice + 1, capacidad - W[ indice ] ) + V[ indice ];
    int no_tomar = sum_max( indice + 1, capacidad );
    return max(tomar, no_tomar);
}
```

Si aplicamos programación dinámica

```
.
.
.
int memoria[ 1005 ][ 1005 ];
.
.
.
int sum_max( int indice, int capacidad ){
    if( indice >= n )
        return 0;
    if( capacidad < 0 )
        return MIN_INT;
    if( memoria[ indice ][ capacidad ] != MIN_INT )
        return memoria[ indice ][ capacidad ];
    //Si lo tomo cuál es el valor máximo que podemos tener con la respuesta actual?
    int tomar = sum_max( indice + 1, capacidad - W[ indice ] ) + V[ indice ];
    int no_tomar = sum_max( indice + 1, capacidad );
    return memoria[ indice ][ capacidad ] = max(tomar, no_tomar);
}
```

Ahora bien debemos responder ¿dónde está la respuesta a nuestro problema?

```
.
.
.
int main(){
    .
    .
    .
    //Si indexamos desde cero aquí estará la solución
    cout << sum_max( 0, M );
    .
    .
    .
    return 0;
}
```

3.3 Problema

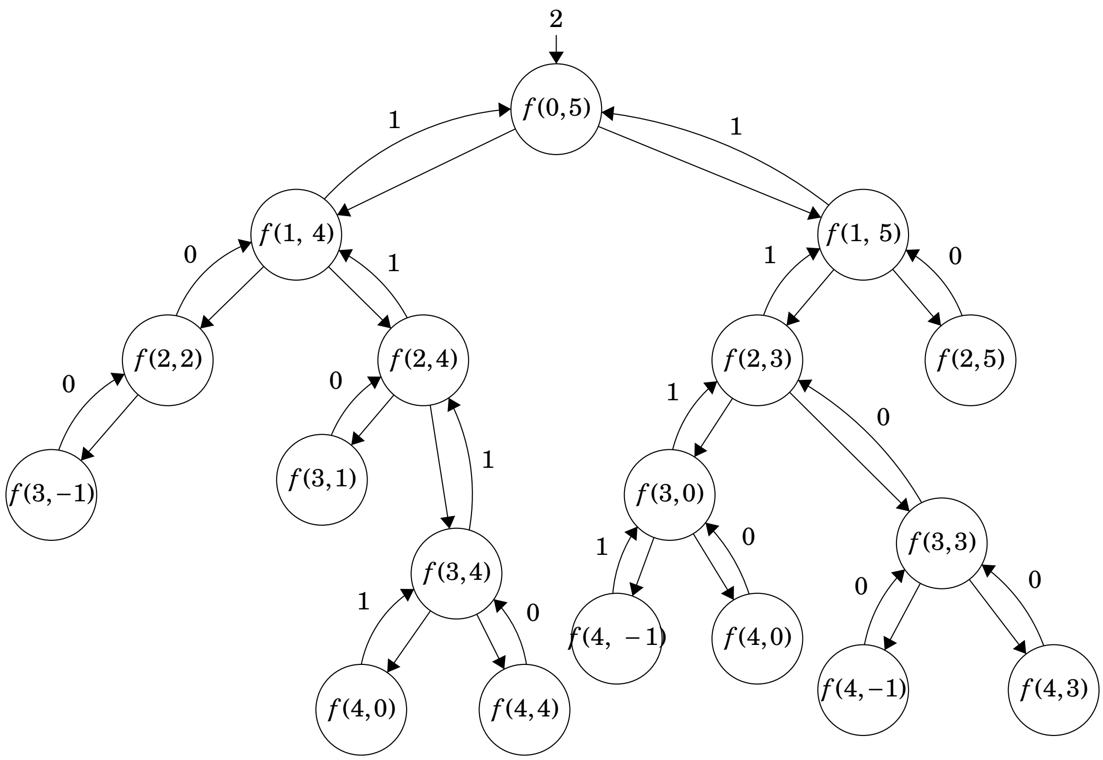
Imagina que compraste algo en algúna tienda, supermercado, etc, tienes que pagar con N monedas, pero quieres saber ¿De cuantas formas utilizando cualquiera de esas N monedas puedo dar el cambio (M)?
Restricciones:

- N monedas $N \leq 1000$
- M cambio $N \leq 1000$
- C_i Monedas $1 leq C_i \leq 1000$

Ejemplo: N = 4, M = 5
C = [1, 2, 3, 4] //Indexado desde cero

- (0, 3) = 1 + 4 = 5
- (1, 2) = 2 + 3 = 5

Es decir nuevamente debemos de considerar el problema como tomar o no tomar



Como solución bruta ¿Qué podemos programar?

```
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    return tomar + no_tomar;
}
```

Si añadimos programación dinámica tenemos lo siguiente

```
.
.
.
int memoria[ 1005 ][ 1005 ];
```

```
bool visitado[ 1005 ][ 1005 ];
.
.
.
int cuenta( int indice, int cambio ){
    //Verificamos que no nos salgamos del rango
    if( indice == n ){
        //Si llegamos al límite verificamos si lo que tenemos es una solución válida
        if( cambio == 0 )
            //Como encontramos una solución la contamos
            return 1;
        //Como no hay solución retornamos cero
        return 0;
    }
    //Si el cambio es negativo no es una combinación válida
    if( cambio < 0 )
        return 0;

    if( visitado[ indice ][ cambio ] )
        return memoria[ indice ][ cambio ];
    //Intentamos tomando el elemento actual
    int tomar = cuenta( indice + 1, cambio - C[ indice ] );
    int no_tomar = cuenta( indice + 1, cambio );
    visitado[ indice ][ cambio ] = true;
    return memoria[ indice ][ cambio ] = (tomar + no_tomar);
}
```

3.4 Boredrom

Vamos a tomar el a_k elemento, vamos remover de nuestro arrego el $a_k + 1$ y $a_k - 1$
Restricciones: 1 1 2 2 2 2 2 3 3
En este caso elegimos el 2 y removemos los unos y los tres nos queda
2 2 2 2 2
Como no tenemos más elementos para elegir nuestra respuesta es la suma de los elementos restantes es decir 5
¿Cómo podemos programar eso?

```
#include <bits/stdc++.h>

using namespace std;
typedef long long int lli;
lli bucket[ 100005 ];
lli DP[ 100005 ];
bool visited[ 100005 ];
lli n;
int MAX = -1;

lli max_sum( int index ){

    if( index >= MAX + 1)
        return 0;

    if( visited[ index ] )
        return DP[ index ];
    visited[ index ] = true;
    //Tomamos el elemento actual
    lli take = max_sum( index + 2 ) + ( bucket[ index ] * index );
    //Nos saltamos el elemento actual
    lli not_take = max_sum( index + 1 );

    return DP[ index ] = max( take, not_take );
}

int main(){
```

```
        cin >> n;
        for( int i = 0, v; i < n; i++){
            cin >> v;
            MAX = max( MAX, v );
            //Generamos nuestra cubeta
            bucket[ v ]++;
        }

        cout << max_sum( 1 ) << endl;
        return 0;
    }
```

3.5 LIS - Longest Increasing Subsequence

Nos sirve para encontrar la subsecuencia más larga que sea creciente. Por ejemplo:

[1, 5, 3, 7, 8, 4]

Tenemos algunas posibilidades

5, 7, 4 consideremos que el conjunto vacío también es válido. Pero esta subsecuencia no es creciente.

Pero si las siguientes 1, 5, 7, 8 y 1, 3, 7, 8. dónde la resuesta es 4.

¿Cómo podemos solucionar esto? Podemos calcular todos los pusibles subconjuntos haciendo algo similar al problema de la mochila, es decir, tomar o no tomar. También requerimos conocer el número anterior ya que este nos ayudará a decidir si el número actual podemos tomarlo o no. **Restricciones:**

- $0 < n \leq 1000$
- $0 < A_i \leq 1000$

```
int LIS( int index, int last ){
    //Caso base
    if( indice == n )
        return 0;

    //Vamos a tomar como válido el valor si y solo si si el i-ésimo elemento es mayor
         que el i-ésimo - 1
    if( A[ index ] > last ){
        int take = LIS( index + 1, A[ index ] ) + 1;
        int not_take = LIS( index + 1, last );
        return max( take, not_take );
    }
    return LIS( index + 1, last );
}
.
.
.
int main(){
    .
    .
    .
        cout << LIS( 0, INT_MIN ) << endl;
    .
    .
```

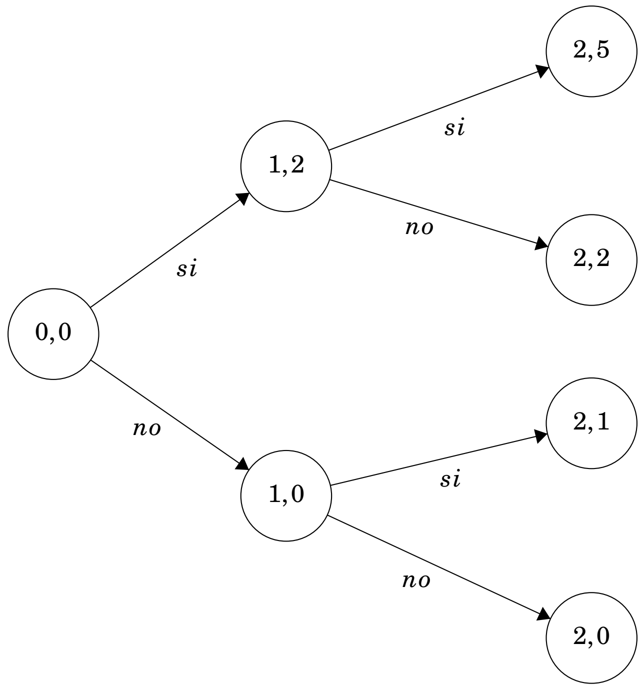
```
        .
    }
```

Esta solución obtiene todos los subconjuntos pero la complejidad es de $O(2^n)$. Si queremos optimizar usando memoria debemos ver los estados de nuestra DP, en este caso tenemos dos, el primero el índice y el segundo el valor más grande que puede valer last, es decir, 1000 y 1000 respectivamente. Por tanto obtenemos cono resultado

```
    .
    .
    .
    int DP[ 1005 ][ 1005 ] = { -1 };
    .
    .
    .
    int LIS( int index, int last ){
        //Caso base
        if( indice == n )
            return 0;
        if( DP[ index ][ last ] != -1 )
            return DP[ index ][ last ];
        //Vamos a tomar como válido el valor si y solo si si el i-ésimo elemento es mayor
             que el i-ésimo - 1
        if( A[ index ] > last ){
            int take = LIS( index + 1, A[ index ] ) + 1;
            int not_take = LIS( index + 1, last );
            return DP[ index ][ last ] = max( take, not_take );
        }
        return DP[ index ][ last ] = LIS( index + 1, last );
    }
    .
    .
    .
    int main(){
        .
        .
        .
            cout << LIS( 0, INT_MIN ) << endl;
        .
        .
        .
    }
```

Con esta solución la complejidad que obtenemos es de $O(n * max(A_i))$

Gráficamente tenemos algo como esto.



Si queremos imprimir la secuencia que tenemos ¿Que podemos hacer? Podríamos usar un string o vector o también recorrer la memoria ¿Hay alguna otra solución?

Podemos reutilizar nuestra función LIS, para reconstruir la solución. Dónde vamos a añadir el valor que nos convino más como solución a nuestra respuesta. Es decir podemos tener un vector de enteros que obtenga la reconstrucción.

```
.
.
.
vector< int > reaconstruct;
void reconstructLIS( int index, int last ){

    if( index >= n )
        return;

    if( A[index] > last ){
        //Como ya se han memorizado los valores de todos los subconjuntos válidos
            obtendremos en constante cual camino me es más conveniente tomar
        int take = LIS( index + 1, A[ index ]) + 1;
        int not_take = LIS( index + 1, last );
        //Aquí podemos decidir que camino nos conviene más si el de tomar o no tomar
        if( take > not_take ) {
            //Añadimos a nuestra respuesta
            reconstruct.push_back( A[ index ] );
            reconstructLIS( index + 1, A[ index ] );
        } else
            reconstructLIS( index + 1, last );
    }
    reconstructLIS( index + 1, last );
}
.
.
.
```

¿Qué complejidad tiene reconstruir la respuesta? $O(n)$ ya que solamente vamos ha realizar una sola llamada recursiva

3.6 Longest Common Subsequence

Dadas dos cadenas, hallar la subsecuencia común más larga.

Restricciones:

Ejemplo:

- S = AGPGTAB
- T = GXTXAYB

A	G	P	G	T	A	B
G	X	T	X	A	Y	B

¿Podemos atacar este problema utilizando la misma lógica que en knapsack? Si intentamos hacer esto tendremos varios problemas. Podemos remover un caracter de alguna de las dos cadenas en algún punto en el que la cadena no coincida. Si ámbos caracteres coinciden ambos los ”quitamos” y aumentamos en 1 la respuesta actual.

```
int LCS( string S, string T ){

    if( S.empty() || T.empty() )
        return 0;

    string S2 = S;
    string T2 = T;
    //erase elimina el caracter (es un iterador) en nuestro caso el primero de la
        cadena S2
    S2.erase( S2.begin() );
    T2.erase( T2.begin() );

    //Si ámbos coinciden aumentamos en uno la respuesta
    if( S[ 0 ] == T[ 0 ] )
        //Como el caracter es el mismo en ámbos la respuesta aumenta en uno
        return LCA( S2, T2 ) + 1;

    int delete_first = LCS( S2, T );
    int delete_second = LCS( S, T2 );

    return max( delete_first, delete_second );
}
```

¿Como podemos asegurar que nos conviene realizar lo anterior? o bien ¿Hay algún caso que no se ha cubierto? y ¿Cuál es la complejidad de este algoritmo?

Supongamos que tenemos una cadena donde un caracter de la respuesta aparece más de una vez, si consideramos el segundo y no el primero es posible que nuestra respuesta o no mejore o empeore.

La complejidad de realizar esto es $O(2^n * n)$ (El n esta ahí por la función erase). ¿Cómo podemos mejorar esta solución?

Nótese que solo estamos utilizando lo caracteres del final, es decir solo utilizamos el sufijo (caracteres del final), así pues en lugar de utilizar una subcadena podemos utilizar un apuntador.

```
int LCS( int index_s, int index_t ){
```

```
        //Si ya no hay caracteres que analizar terminamos las recursiones
        if( index_s == S.size() || index_t == S.size() )
            return 0;

        if( S[index_s] == T[index_t] )
            return LCA( index_s + 1, index_t + 1) + 1;

        int move_S = LCS( index_s + 1, index_t );
        int move_T = LCS( index_s, index_t + 1 );

        return max( move_S, move_T );
    }
```

¿Ahora que complejidad obtenemos? $O(2^n)$ ya que hemos eliminado el uso de la función erase. Finalmente ¿Como implementamos programación dinámica? En este caso tenemos dos estados index_s e index_t.

```
    .
    .
    .
    int DP[ 1005 ][ 1005 ];
    .
    .
    .
    int LCS( int index_s, int index_t ){
        //Si ya no hay caracteres que analizar terminamos las recursiones
        if( index_s == S.size() || index_t == S.size() )
            return 0;
        //Nuestra ''bandera'' es el valor de -1 ya que nunca nos devolverá ese valor la
            función
        if( DP[ index_s ][ index_t ] != -1 )
            return DP[ index_s ][ index_t ]

        if( S[index_s] == T[index_t] )
            return DP[ index_s ][ index_t ] = LCA( index_s + 1, index_t + 1) + 1;

        int move_S = LCS( index_s + 1, index_t );
        int move_T = LCS( index_s, index_t + 1 );

        return DP[ index_s ][ index_t ] = max( move_S, move_T );
    }
    .
    .
    .
    int main(){
        .
        .
        .
        cout << LCS( 0, 0 ) << endl;
        .
        .
        .
        return 0;
    }
```

Y finalmente hemos optimizado sustancialmente ese problema, hemos pasado de tener una complejidad de $O(2^n * n)$ a $O(2^n)$ a finalmente $O(|S| * |T|)$ o bien en $O(n^2)$

4

DP con máscaras de bits

4.1 Manipulación de bits

Para estar en sintonía en un número binario debemos mencionar lo siguiente

31	30	...	1	0
0	0	...	0	0_{2}
MSB				LSB

También tenemos varias operaciones en binario por ejemplo el AND

	1	1	0	0
&	1	0	1	0
	1	0	0	0

Otra operación es OR

	1	1	0	0
|	1	0	1	0
	1	1	1	0

Otra operación es la XOR

	1	1	0	0
^	1	0	1	0
	0	1	1	0

Negación

~	1	1	0	0
	0	0	1	1

Y también tenemos optras operaciones útiles como los corrimientos de bits a la derecha o a la izquierda por ejemplo

5	<<	2	=	00...010100
5	>>	2	=	00...0001
5	>>	3+2	=	5>>5

Algunas operaciones que podemos hacer por ejemplo:

Saber si el k-ésimo bit está encendido

```
isSet( n, k ) bool ( n & ( 1 << k ) );
```

Colocar un uno en la k-ésima posición

```
setBit( n, k ) bool ( n | ( 1 << k ) );
```

Apagar el k-ésimo bit

```
clearBit( n, k ) bool ( n &~ ( 1 << k ) );
```

Cambiar el valor del k-ésimo bit (si es cero se vuelve uno y viceversa)
Saber si el k-ésimo bit está encendido

```
isSet( n, k ) bool ( n ^ ( 1 << k ) );
```

Si queremos saber en constante el bit menos significativo de un número

8	=	0	...	1	0	0	0
~8	=	1	...	0	1	1	1
-8	=	1	...	1	0	0	0
8&-8	=	0	...	1	0	0	0

Por tanto obtenemos en código

```
lowestBit( n, k ) (b &- n)
```

Si queremos encender los primeros K-bits pordemos hacer $2^k - 1$ pero ¿Podemos hacerlo más rápido?

```
setFirstK( k ) ( ( 1 << k ) - 1);
```

¿Cómo contar el número de bits encendidos rápido? Podemos utilizar un ciclo y contar los bits ¿Pero hay una forma más fácil?
en C++ tenemos una función llamada __builtin_popcount(n) si queremos trabajar con long long int podemos utilizar __builtin_popcountll(n)

4.2 Máscaras de bits

Tenemos lo siguiente
Los elementos que tienen un bit encendido pertenecen a un conjunto.
Si el conjunto es el vacío es pues es el que cuyo valor es cero, es decir:
$\emptyset = 0$

k elementos
$U = 2^k - 1$

Como estamos trabajando con conjuntos ¿Que operación es la unión de dos conjuntos A y B?
$A|B$

¿La intersección de A y B?
$A\&B$

El complemento
A & U, A ∧ U

Insertar K en A
setBit(A, k);

Eliminar K de A:
clearBit(A, k);

- $\{\emptyset\}$
- $\{0\}$
- $\{0,1\}$
- $\{0,1,2\}$
- $\{0,2\}$
- $\{1\}$
- $\{1,2\}$
- $\{2\}$

¿Como itero sobre todos los subconjuntos?

```
for( i = 0 ... i << k - 1 )
    for( j = 0 ... k - 1 )
        if( isSet(i, j) )
```

¿Cuál es la complejidad de lo anterior? $O(2^k * k)$

4.3 Ejercicio

Nos dan máscaras de bits, llamémosle A queremos todos los subconjuntos que hay en A por ejemplo:

31							0
0	0	...	1	1	0	1	0
			1	1	0	1	0
			0	1	0	1	0
			0	0	0	1	0
			1	0	0	0	0
			1	0	0	1	0
			0	0	0	0	0
			1	1	0	0	0
			0	1	0	0	0

5 Upsolving RPC

5.1 I. Überwatch

Podemos utilizar DP el problema se resume en tomar y no tomar como el problema de la mochila.

5.2 D. Pants on Fire

Se quiere verificar si lo que dice un períodico es verdadero, falso o me falta información para que sea verdad. En el siguiente ejemplo es evidente que los Rusos son Peóres que los Mexicanos pero podemos ver que los Mexicanos son peóres que los Americanos lo podemos representar así:
Rusos > Mexicanos > Americanos por tanto los Rusos son peores que los americanos (transitividad)

Periódico dice:				
Mexicans	are	worst	than	Americans
Russians	are	worst	than	Mexicans
NorthKoreans	are	worst	than	Germans
Canadians	are	worst	than	Americans
Donald Trump dice:				
Russians	are	worst	than	Americans
Germans	are	worst	than	NorthKoreans
NorthKoreans	are	worst	than	Mexicans
NorthKoreans	are	worst	than	French
Mexicans	are	worst	than	Canadians

Ahora sabemos que los Norcoreanos son peóres que los Alemanes según el periódico pero Trump dijo que los Alemanes son peores peores que los Norcoreanos. Es decir tenemos:
Norcoreanos > Alemanes y Alemanes peores que Norcoreanos por tanto tenemos un alternative fact.
Si no es verdad lo que se dice aún que se invierta el órden entonces imprimimos Pants of fire.

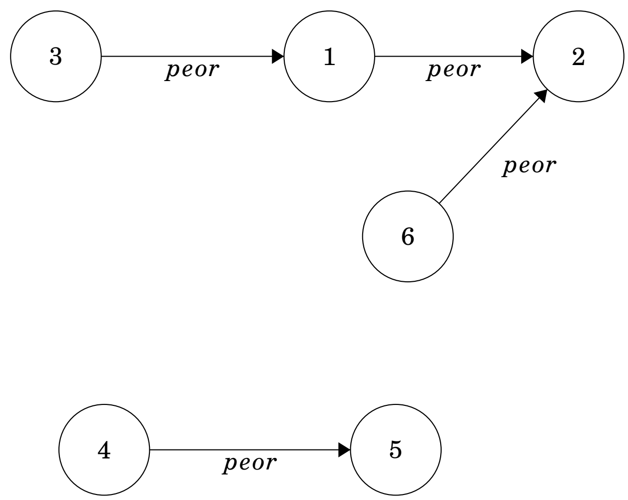
Solución:

Supongamos que:

- Mexicanos = 1
- Americanos = 2
- Rusos = 3

- Norcoreanos = 4
- Alemanes = 5
- Canadiences = 6

Si modelamos esto con un grafo tenemos lo siguiente



Si quiero saber si X es peor que Y pero no lo tengo directo ¿Que puedo hacer? podemos pararnos en la arista X y ver si puedo llegar a la arista Y.
Se de una arista no puedo llegar a otra quiere decir que no es una verdad, pero debemos ver si es un alternative Fact, esto lo logramos inviertiendo nuestra búsqueda si buscamos desde Y a X y se cumple tenemos un alternative fact.
Finalmente si ninguno de los dos se cumple imprimimos Pants on fire.
¿Que pasa si no conocemos una nacionalidad? Si no esta en nuestro diccionaro es una mentira por tanto mandamos Pants on fire, es decir ni existe.

Para implementar que ¿podemos hacer?

```
#include <bits/stdc++.h>

using namespace std;

map<string, int> dictionary;
vector< vector<int> > adjList;
vector< bool > visited;

bool DFS( int u, int v ){
    bool ans = false;
    if( u == v )
        return true;
    visited[ u ] = true;
    for( int i = 0; i < adjList[ u ].size(); i++ ){
        int w = adjList[ u ][ i ];
        if( !visited[ w ] ){
            if( DFS( w, v ) ){
                ans = true;
                break;
            }
        }
    }
    return ans;
}
```

```
int main(){
    int count = 0;
    int m, n, u, v, a, b;
    cin >> m >> n;
    for( int i = 0; i < m; i++ ){
        cin >> a >> b;
        if( dictionary.find( a ) != dictionary.end() )
            u = dictionary[ a ];
        else
            u = dictionary[ a ] = count++;
        if( dictionary.find( b ) != dictionary.end() )
            v = dictionary[ b ];
        else
            v = dictionary[ b ] = count++;
        //Enlazamos los nodos (creamos el grafo)
        adj[ u ].push_back( v );
    }
    //Procesamos las queries
    for( int i = 0; i < n; i++ ){

        //Extraemos los IDs
        cin >> a >> b;
        //Buscamos el ID de la primer nacionalidad
        if( dictionary.find( a ) != dictionary.end() )
            u = dictionary[ a ];
        //Buscamos el ID de la segunda nacionalidad
        if( dictionary.find( b ) != dictionary.end() )
            v = dictionary[ b ];

        //Se cumplió totalmente lo que se ha dicho
        if( DFS(u, v) )
            cout << "Fact\n";
        //Si invertimos se cumple
        else if( DFS(v, u) )
            cout << "Alternative fact\n";
        //No es verdad
        else
            cout << "Paths on fire\n";
    }
    return 0;
}
```

6 Teoría de números

Recordemos los conjuntos de números

- Números enteros: $Z = ..., -3, -2, -1, 0, 1, 2, 3, ...$
- Números naturales/Números positivos $N = 1, 2, 3, ...,$

6.1 Divisibilidad

d divide a **a** si existe un entero **q** tal que $a = dq$, y se escribe $d|a$
Ejemplos:

- $2|6 \rightarrow 6 = 2(3)$
- $5|6 \rightarrow 6 = 5(q), q = 1.2$

6.1.1 Algoritmo de la división

Sean $a, b\epsilon Z$. Entonces existen $q, e\epsilon Z$ tal que:

$$a = bq + r$$

donde: $0 \leq r < |b|$

Ejemplo
a = 15, b = 6
15 = 6(2) + 3
dónde:

- El cociente = 2
- El residuo = 3

De forma general

$$q = piso\left(\frac{a}{b}\right)$$

Y también

$$r = a\%b$$

¿Cómo hallar los divisores de un número?

```
vector< int > divisores( int n ){
    vector< int > div;
    for( int d = 1; d <= n; d++)
        if( n % d == 0 )
            div.push_back( d );
    return div;
}
```

La complejidad del algoritmo anterior es $O(n)$ siendo así vamos a tratar de optimizar esta función

Que pasa si queremos los divisores de un n = 24, los divisores de este número son $\{1,2,3,4,6,8,12,24\}$ podemos observar que siempre los divisores van en "parejas" es decir siempre tenemos lo siguiente:

- d = 1, q = 24
- d = 2, q = 12
- d = 3, q = 8
- d = 4, q = 6

Además sabemos que n = dq por tanto

$$d \le q$$
$$d^2 \le dq$$
$$\sqrt{d^2} \le \sqrt{n}$$
$$d \le \sqrt{n}$$

¿En código que obtenemos?

```
vector< int > divisores( int n ){
    vector< int > div;
    for( int d = 1; d *d <= n; d++){
        if( n % d == 0 ){
            int q = n / d;
            div.push_back( d );
            //Para evitar números repqtidos, analizar el caso del 25
            if( q != d)
                div.push_back( q );
        }
    }
    return div;
}
```

La complejidad de este algoritmo es de $O(\sqrt{n})$

Que pasa si hacemos uso de números negativos ¿Qué sucede?

$$inta = -13, b = 4$$
$$intq = \frac{a}{b}$$
$$intr = a\%b$$
$$a = bq + r$$
$$-13 = 4(-3) + (-1)$$
$$-13 = -12 - 1$$

Lo cual no es del todo correcto ya que nuestro residuo debería ser 3

$$-13 = 4(-4) + 3$$

$$-13 = -16 + 3$$

¿Como reparamos esto?

```
if( r < 0 )
    r += b
```

6.2 Números primos

Sea $p \epsilon N$ con $p \geq 2$ p es primo si: sus unicos divisores son 1 y p.

6.2.1 Test de primalidad

¿Cómo puedo saber si un n es primo?

```
bool esPrimo( int n ){
    if( n < 2 )
        return false;

    for( int d = 2; d*d <= n; d++ )
        if( n % q == 0 )
            return false;
    return true;
}
```

La complejidad de este algoritmo es $O(\sqrt{n})$

Problema

Hallar todos los primos menores o iguales a n
Ejemplo: $n = 10 \rightarrow \{2, 3, 5, 7\}$

```
vector< int > primos;
for( int d = 2; d <= n; d++ ){
    if( esPrimo( d ) )
        primos.push_back( d );
}
```

La complejidad de este algoritmo es $O(n\sqrt{n})$ en este caso a lo mucho podríamos procesar en un segundo si $n \leq 5 * 10^5$

6.3 Criba de eratóstenes

En este caso nos ”paramos” en un número y ”marcamos” todos sus múltiplos.

Empezamos con el 2

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		x		x		x		x		x		x		x		x		x

Nos paramos en el 3

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		x		x		x	x	x		x		x	x	x		x		x

Nos paramos en el 4

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		x		x		x	x	x		x		x	x	x		x		x

Ya hemos hecho lo mismo con los 20 números

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		x		x		x	x	x		x		x	x	x		x		x

¿En código como sería esto?

```
vector< int > criba( int n ){
    vector< bool > esPrimo( n + 1, true );
    vector< int > factoresPrimos;
    factoresPrimos.push_back( 2 );
    for( int i = 4; i <= n; i += 2 )
        esPrimo[ i ] = false;

    for( int i = 3; i <= n; i++ ){
        if( esPrimo[ i ] ){
            factoresPrimos.push_back( i );
            for( int j = i * 2; j <= n; j += i ){
                esPrimo[ j ] = false;
            }
        }
    }
}
```

La complejidad de este algoritmo ¿cual es? $O\left(n + \frac{n}{2} + \frac{n}{3} + ... + \frac{n}{n}\right)$ lo que es igual a $O\left(\sum_{i=1}^{n}(\frac{n}{i})\right)$ si opeamos $O\left(n\sum_{i=2}^{n}\frac{1}{i}\right)$ asintóticamente esta suma es igual a $log(n)$ es decir la complejidad de este algoritmo es $O(nlog(n))$ la complejidad espacial (en memoria) es $O(n)$ lo cual es muy bueno si $n \leq 10^7$ Podemos optimizar un poco máscaras

```
vector< int > criba( int n ){
    vector< bool > esPrimo( n + 1, true );
    vector< int > factoresPrimos;
    factoresPrimos.push_back( 2 );
    for( int i = 4; i <= n; i += 2 )
        esPrimo[ i ] = false;

    for( int i = 3; i <= n; i++ ){
        if( esPrimo[ i ] ){
            factoresPrimos.push_back( i );
            //Iniciamos en i * i no en i * 2
            for( int j = i * i; j <= n; j += i ){
                esPrimo[ j ] = false;
            }
        }
    }
}
```

La complejidad de esta $O(nlog(log(n)))$ podemos mejorarla un poco más

```
vector< int > criba( int n ){
    vector< bool > esPrimo( n + 1, true );
    vector< int > factoresPrimos;
```

```
        factoresPrimos.push_back( 2 );
        for( int i = 4; i <= n; i += 2 )
            esPrimo[ i ] = false;

        for( int i = 3; i <= n; i++ ){
            if( esPrimo[ i ] ){
                factoresPrimos.push_back( i );
                //Incrementamos en j += 2*i
                for( int j = i * i; j <= n; j += 2 * i ){
                    esPrimo[ j ] = false;
                }
            }
        }
    }
```

6.4 Obtención de factores primos

Ya conocemos la criba, con la misma podemos obtener otra información. En esta ocasión vamos a obtener el factor primo más pequeño de un número. En este caso utilizaremos un arreglo de booleanos.

Un factor primo recordemos que se puede definir de la siguiente forma

$$N = P_1^{\alpha_1} P_2^{\alpha_2} ... P_k^{\alpha_k}$$

Por ejemplo, ¿Cuál es la factorización prima del 36?

$$36 = 2^2 3^2$$

Esto también es comocido como el teorema fundamental de la aritmética. También es importante saber que esta factorización es **única**.
También consideremos que el factor primo de un número primo es si mismo. Por ejemplo los factores primos del 3 es el 3 mismo.
Tenemos varios algoritmos para obtener el factor primo más pequeño, nosotros usaremos el siguiente:

```
    vector< int > lowestPrime( n + 1 );
    //Marcamos todos los pares
    for( int i = 2; i <= n; i += 2 )
        lowestPrime[ i ] = 2;
    //Recorremos todos los impares
    for( int i = 3; i <= n; i += 2 ){
        //No se ha marcado aún
        if( lowestPrime[ i ] == 0 ){
            lowestPrime[ i ] = i;
            for( int j = i * i; j <= n; j+= 2 ){
                //Quizá alguien ya lo marcó, por tanto preguntamos si esta disponible
                    para usar
                if( lowestPrime[ j ] == 0 )
                    lowestPrime[ j ] = i;
            }
        }
    }
```

Este algoritmo tiene una complejidad de $O(nln(n))$

Ahora vamos a obtener la descomposición prima de un número n, si lo hacemos en papel, podemos construir una tabla, nuevamente tomemos de ejemplo al 36

36	2
18	2
9	3
3	3
1	

```
vector< int > factorizar( int n ){
    vector< int > factor;
    while( n > 1 ){
        int p = lowestPrime[ n ];
        factor.push_back( p );
        n /= p;
    }
    return factor;
}
```

La complejidad de obtener los factores primos es aproximadamente $O(n\,primos)$ esto es aproximadamente $O(log_2(n))$

¿Cómo podemos obtener los primos cuya potencia es par?

```
#define vector<int, int> pii;
vector< pii > factorizar( int n ){
    vector< pii > factor;
    while( n > 1 ){
        int p = lowestPrime[ n ];
        int pow = 0;
        while( n % p == 0 )
            n /= p, pow++;
        //emplace_back llama al constructor de pair
        factor.emplace_back( p, pow );

    }
    return factor;
}
```

Otra característica de los números enteros es que los divisores de un n tienen los mismos divisores primos pero con potencias diferentes. Por ejemplo, nuevamente para el 36:

1	
2	
3	
4	$2^2 3^0$
6	
9	
12	$2^2 3^3$
18	
36	

Más formalmente

$$N = P_1^{\alpha_1} P_2^{\alpha_2} ... P_k^{\alpha_k}$$

x divide a N

$$x = P_1^{\beta_1} P_2^{\beta_2} ... P_k^{\beta_k}$$

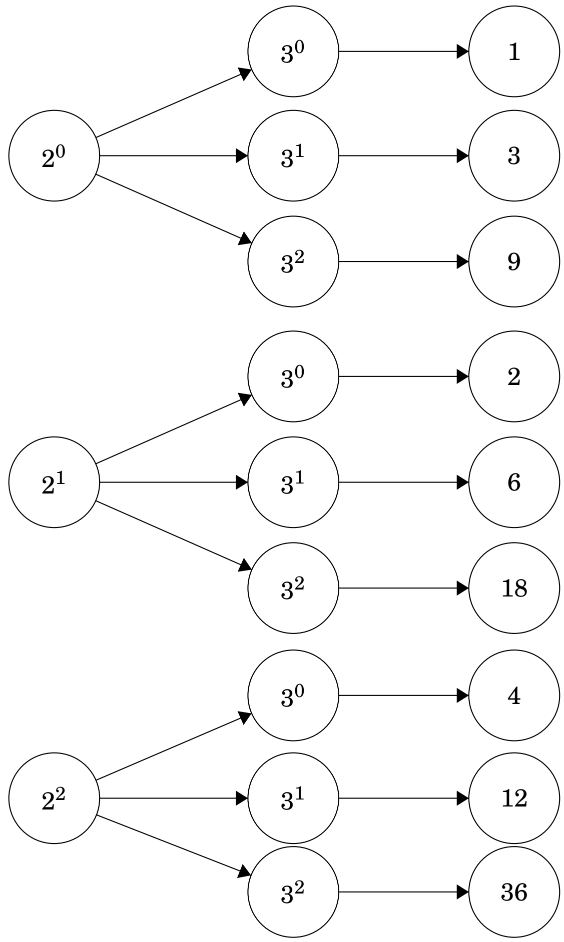
$$\beta_1 \leq \alpha_1, \beta_2 \leq \alpha_2, ..., \beta_k \leq \alpha_k$$

6.5 Ejercicio

Dados factores primos y sus potencias, hallar todos los divisores de N $N \leq 10^{18}$
Ejemplo:
N = 36
2, 2
3, 2
El Output sería
Res = 1, 2, 3, 4, 6,... , 36
Solución:
Sabemos del teorema fundamental del cálculo que las potencias a las que estan elevados los números primos (factores de nuestro número N). Haciendo uso de ese hecho podemos iterar sobre todas las potencias que nos dan, desde cero hasta la k potencia. En código podemos probar todas las combinaciones de forma recursiva.

```
#define pair< int , int > pii
.
.
.
vector< int > divisors;
//Aquí vamos a guardar los números y sus potencias
vector< pii > factors;

void findDivisors( int index , int current_product ){

    //Guardamos el divisor
    if( index == factors.size() )
        divisors.push_back( current_product );

    for( int i = 0; i <= factors[ index ].second; i++ ){
        findDivisors( index + 1, current_product * pow( factors[ index ].first , i ) )
            ;
    }

}
.
.
.
int main(){
    pii factor;
    for( int i = 0; i < k; i++ ){
        //first = número primo
        //second = la potencia
        cin >> factor.first >> factor.second;
        factors.push_back( factor );
    }
}
```

Si dibujamos las llamadas recursivas tendríamos algo como esto



6.6 Máximo Común Divisor (GCD)

La notación ha utilizar será GCD(a, b) = g, dónde:

- g divide a a
- g divide a b
- g es el máximo entero que cumple las propiedades

Por ejemplo, ¿Cuáles son los divisores comúnes de 48 y 72?
$\{1,2,3,4,6,8,12,24\}$
El divisor común con valor mayor es el 24, este es el GCD(48, 72)

Algunas propiedades

- El GCD entre 0 y N es N es decir $GCD(0,N) = N$
- El GCD entre 1 y N es 1 es decir $GCD(1,N) = 1$
- El GCD entre N y N + 1 siempre es 1 es decir $GCD(N,N+1) = 1$
- Si el GCD entre a y b es igual con 1 esos números son comprimos es decir $GCD(A,B) = 1$
- EL GCD entre un primo P_1 y un primo P_2 siempre es 1 ya que ambos números tienen como único divisor y el número mismo, por tanto el único divisor común es 1 es decir $GCD(P_1,P_2) = 1$

6.6.1 Algoritmo de Euclides

Dado un número a y un número b, ¿Cómo podemos encontrar su GCD?

```
int Euclides( int a, int b ){
    if( a > b )
        swap(a, b);
    while( a != 0 ){
        b %= a;
        swap(a, b)
    }
    return b;
}
```

Lo anterior sirve si y solo si Si $a > b$
Si se cumple esa condición sabemos que siempre: $b\%a < a$
También existe una desigualdad la cual dice que: $b\%a \leq \frac{b}{2}$
La cual es demostrable si separamos en dos casos:

1. $a \leq \frac{b}{2}$
2. $a > \frac{b}{2}$

Otra versión del algoritmo de euclides es:

```
int Euclides( int a, int b ){
    return a ? ( b % a, a ) : b;
}
```

Si usamos C++ podemos usar

```
g = __gcd(a, b);
```

6.7 Mínimo común multiplo (LCM)

Dados dos números a y b se cumple que:

- a divide a l
- b divide a l
- l es el menor entero que cumple

Ejemplo
El MCM entre 10 y 15 es el 30 ya que:

- $\frac{10}{30} = 3$
- $\frac{15}{30} = 2$

¿Como obtenemos esto? consideremos que

$$a * b = GCD(a,b) * LCM(a,b);$$

Despejando

$$LCM(a,b) = \frac{a * b}{GCD(a,b)}$$

También debemos preguntarnos ¿Cómo es que $a * b = GCD(a,b) * LCM(a,b)$; es verdad?
Sea:

- $a = P_1^{\alpha_1} P_2^{\alpha_2} ... P_k^{\alpha_k}$
- $b = P_1^{\beta 1} P_2^{\beta 2} ... P_k^{\beta k}$

- $g = P_1^{\gamma 1} P_2^{\gamma 2} ... P_k^{\gamma k}$

Sabemos que las potencias α seran menores que las potencias de γ lo mismo con las potencias de β, es decir:

- $\gamma_k \leq \alpha_k$
- $\gamma_k \leq \beta_k$

El GCD lo obtenemos de el mínimo entre ambas potencias es decir
$\gamma_k = min(\alpha_k, \beta_k)$
Mientras que el LCM lo obtenemos de entre el máximo de las potencias, tomemos en cuenta que

- $a = P_1^{\alpha_1} P_2^{\alpha_2} ... P_k^{\alpha_k}$
- $b = P_1^{\beta 1} P_2^{\beta 2} ... P_k^{\beta k}$
- $l = P_1^{\pi_1} P_2^{\pi_2} ... P_k^{\pi_k}$

Nuevamente

- $\pi_k \leq \alpha_k$
- $\pi_k \leq \beta_k$

El LCM lo obtenemos de el máximo entre ambas potencias es decir:
$\pi_k = min(\alpha_k, \beta_k)$

Si utilizamos la definición de LCM que mencionamos anteriormente pero sustituyendo los valores:
$a * b = (P_1^{\alpha_1} P_2^{\alpha_2} ... P_k^{\alpha_k})(P_1^{\beta 1} P_2^{\beta 2} ... P_k^{\beta k})$
Esto es igual a la suma de las potencias es decir $P_1^{\alpha_1+\beta_1} P_2^{\alpha_2+\beta_2} ... P_k^{\alpha_k+\beta_k}$
Por tanto
$GCD(a,b) * LCM(a,b) = \left(P_1^{min(\alpha_1,\beta_1)} \dots P_k^{min(\alpha_k,\beta_k)} \right)$

6.8 Aritmética modular

Anteriormente se ha utilizado el módulo para saber por ejemplo si un número es par o impar, es decir hacemos:

```
if( (a % 2) == 0 )
    cout << "Es par" << endl;
else
    cout << "Es impar" << endl;
```

Formalmente el módulo lo definimos como

$$a = b * q + r$$

dónde:

- a = número
- b = módulo
- q = entero multiplicado tal que nos da un valor igual o menos que el a
- r = residuo

Por ejemplo si tenemos el 7

$$7 = 2 * 3 + 1$$

Entonces el módulo es el residuo de una división. Esto solo aplica a números enteros no para reales. El módulo nos forma familias, las familias son cíclicas. También debemos mencionar que al aplicar módulo n obtendremos un número que está entre $[0, 1, 2, \dots, n-1]$

6.8.1 Congruencia

Un número es congruente si se cumple que:

$$n \equiv a mod b$$

Por ejemplo los números pares

$$4mod2 = 0$$

$$6mod2 = 0$$

Estos siempre ”viven” en las mismas familias

6.8.2 Operaciones

Suma

Supongamos que tenemos lo siguiente

$$(a+b)\%N$$

Podemos representar esto como

$$((a\%n)+(b\%n))\%n$$

Siempre debemos de modular después de una suma ya que si no modulamos nos saldríamos de la familia en la que estamos.

Multiplicación

$$(a+b)\%N$$

Podemos representar esto como

$$((a\%n)*(b\%n))\%n$$

6.8.3 Inverso modular

Ya sabemos que exite la suma y la multiplicación modular, pero ¿existe la división? realmente no hay una definición para esto, pero si sabemos que:

$$\frac{a}{b} = a * b^{-1}$$

Sabiendo esto y también conociendo la regla de congruencia, podemos calcular ese b^{-1} es decir:

$$a * x \equiv 1 mod N$$

$$a * a^{-1} \equiv 1 mod N$$

6.9 Exponenciación binaria

Hasta ahora sabemos que podemos exponenciar un número n a una potencia b es decir

```
pow(n, b);
```

También podemos hacer esto multiplicando nuestro n, b veces

```
int res = 1;
for( int i = 0; i < b; i++ )
    res *= a;
```

El problema de lo anterior es que la complejidad es $O(b)$ es muy ineficiente.

Ya sabemos que:

$$a^{\alpha} * a^{\beta} = a^{\alpha+\beta}$$

Y también sabemos que

$$(a^{\alpha})^{\beta}$$

Tomemos 3^{11} sabemos que podemos descomponer este número en $3^{8} * 3^{2} * 3^{1}$. Ahora, vamos a convertir a base binaria la potencia:

3	2	1	0
1	0	1	1
3^{8}		3^{2}	3^{1}

```
int res = 1;
while( b ){
    if( b & 1 )
        res *= a;
    b >>= 1;
    a *= a;
}
```

Si queremos modular entonces tenemos

```
a = a % N;
int res = 1;
while( b ){
    if( b & 1 )
        res = ( res * a ) % N;
    b >>= 1;
    a = ( a % a ) % N;
}
```

Sea p primo y $a \epsilon Z$ entonces $a^{p} \equiv 1 mod p$

Si $p|a$ entonces $a^{p-1} \equiv 1 mod p$ además $p|a = gcd(a,b)$

También debemos considerar la definición de inverso modular

Sean a y $b \epsilon Z$ si $ab \equiv 1 mod n$, entonces b es el inverso multiplicativo de a módulo n. También $a^{-1} \equiv b mod n$, por ejemplo:

$$2^{-1} = 4 mod 7$$

Es decir $(4 * 2) mod 7 = 8 mod 7 = 1$

6.10 Algoritmo extendido de euclides

Sean $a, b \epsilon Z$ si $d = gcd(a,b)$ se cumple que:

1. $d > 0$
2. $d|a$ y $d|b$
3. d es el máximo valor posible, es decir, si existe $c \epsilon Z$ tal que $c|a$ y $c|b$, entonces $c|d$

Ejemplo:
El gcd(6, 15) = 3
Sabemos que los divisores de 6 son $1,2,3,6$ y los divisores de 15 son $1,3,5,15$, los únicos divisores en común son 1 y 3 pero de aquí el máximo es 3.

Teorema

Si $a = bq + r$ entonces $gcd(a,b) = gcd(b,r)$

Demostración Sea $d = gcd(a,b)$ Queremos probar que $d = gcd(b,r)$ ya sabemos que $d|a$, $d|b$ y $d > 0$

- $d|b$
  $r = a - bq$

  Versión recursiva

```
int gcd(int a, int b){
    if( b == 0 )
        return a;
    return gcd(b, a % b);
}
```

  Versión iterativa

```
int gcd(int a, int b){
    while( b != 0 ){
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

  La complejidad de el GCD es $O(log_{\Phi}(min(a,b)))$

6.11 Combinatoria

Generalmente lo utilizamos cuando debemos contar cosas. Por ejemplos, cuantos conjuntos cumplen con X propiedad. Generalmente son números muy grandes así que la respuesta usualmente va modulada. Así que los temás más importantes a utilizar son:

- Aritmética modular
- Inverso modular

  En combinatoria utilizaremos dos principios:

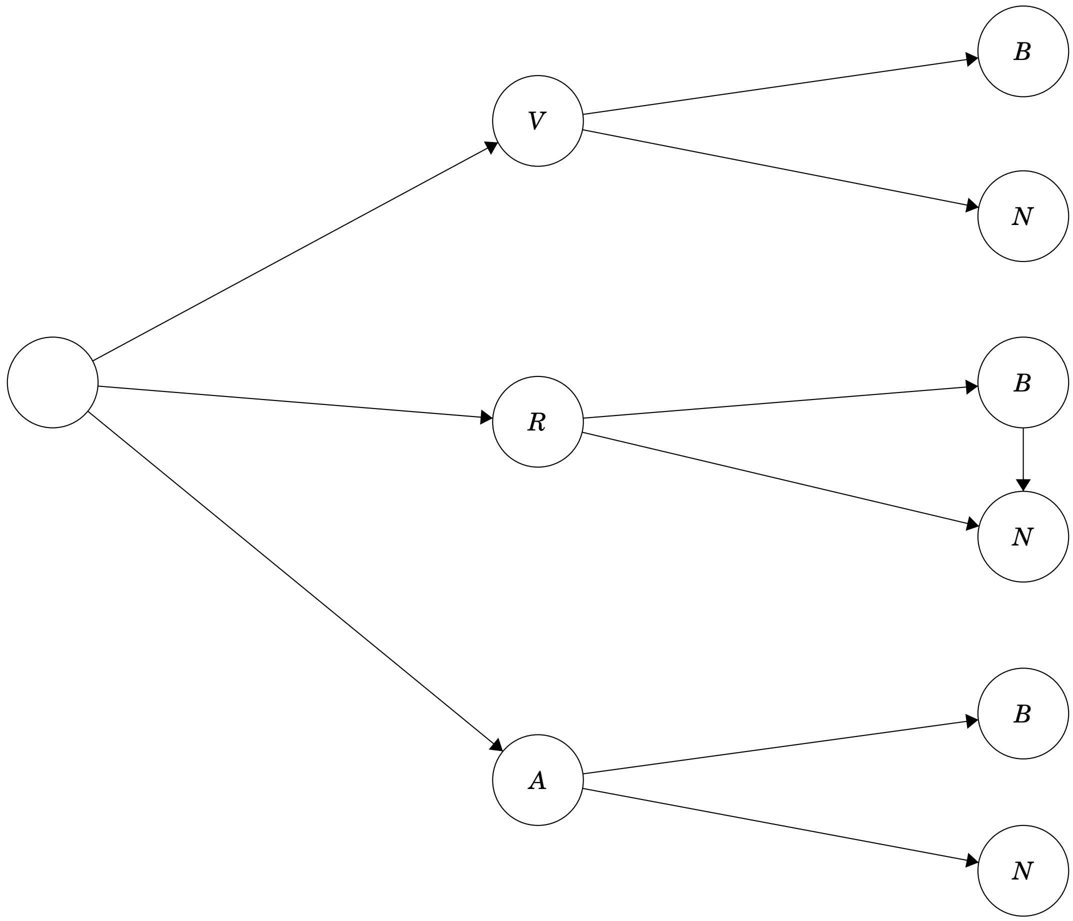
6.11.1 Principio del producto

Si algo puede ser hecho de n formas y luego otra cosa puede ser hecha de m formas entonces ambas se pueden hacer de mn formas. Un ejemplo clásico es el siguiente:
Supongamos que tenemos

- 3 Pantalones (V, R, A)
- 2 Camisas (B, N)

¿De cuantas formas me puedo vestir?

Podemos solucionarlo con un árbol de decision



[H]
Como respuesta tenemos 6 formas de vestirnos.

6.11.2 Principio de la suma

Si algo se puede hacer de n formas y luego se puede hacer otra cosa de m formas pero no se pueden hacer al mismo tiempo, entonces alguna de las dos se puede hacer de $m+n$ formas.
Un ejemplo clásico es:
Si hay un grupo e 20 alumnos y otro con 30 alumnos entonces podemos elegir 50 alumnos de cualquiera de los grupos.

Problema

¿Cuantos números enteros entre 1 y 100 hay que no tengas dígitos pares (0, 2, 4, 6, 8)?
Podemos separar en dos conjuntos, el primero son los números compuestos por un dígito y el segundo que tiene dos dígitos. Imaginemos que tenemos casillas, entonces con un dígito solo podemos tener: 1, 3, 5, 7, 9 en el caso de dos dígitos podemos poner cualquiera de los antes mencionados en la primer casilla y lo mismo en una segunda casilla es decir tenemos 5 * 5 resultados posibles, por principio de la suma el total de números es 5 * 5 + 5 = 30.

6.12 Permutaciones

Variación en el órden de una colección de números.

6.12.1 Permutaciones sin repetición

Dado un conjunto de n elementos, ¿De cuantas formas se pueden colocar en una permutación de tamaño n? Como ejemplo tenemos $1,2,3$ tenemos los subconjuntos:

- $1,2,3$
- $1,3,2$
- $2,1,3$
- $2,3,1$
- $3,1,2$
- $3,2,1$

En total tenemos 6 permutaciones de $1,2,3$.

De manera más formal de n elementos vamos a tener como opciones

$$n(n-1)(n-2)\dots 1 = n!$$

6.12.2 Permutacion de un subconjunto de n elementos

¿Cuantas formas hay de acomodar K elementos de un conjunto de tamaño n en una permutacion? Por ejemplo tomemos un $n=4$ y un $m=2$ es decir tenemos $1,2,3,4$

- $1,2$
  - $[1,2]$
  - $[2,1]$
- $1,2$
  - $[1,2]$
  - $[2,1]$
- $1,3$
  - $[1,3]$
  - $[3,1]$
- $1,4$
  - $[1,4]$
  - $[4,1]$
- $2,3$
  - $[2,3]$
  - $[3,2]$
- $2,4$
  - $[2,4]$
  - $[4,2]$
- $3,4$
  - $[3,4]$

- [4, 2]

Nuevamente si generalizamos para n elementos si tenemos un arreglo de tamaño k tenemos:

$$P_k^n = n(n-1)(n-2)\dots(n-k+1)\frac{(n-k)}{(n-k)}\frac{(n-k-1)}{n-k-1}\dots\frac{1}{1} = \frac{n!}{(n-k)!}$$

6.12.3 Permutación con repeticiones

Es un ordenamiento de n elementos donde hay n_1 del tipo 1, n_2 del tipo 2, ... n_k del tipo k, y $n_1+n_2+\cdots+n_k = n$
Tomemos un ejemplo
n = 5, $n_1 = 3$, $n_2 = 2$
$1,2,1,1,2$
$[1a,2a,1a,1a,2a]$
¿Cuantas permutaciones hay? Como solución tenemos 3! x 2! que es el producto del factorial de los grupos que existen

Formalmente podemos decir que este tipo de permutación la obtenemos de la siguiente manera:

$$P^n_{n_1,n_2,\dots n_k} n_1!n_2!\dots n_k! = n!$$

$$P^n_{n_1,n_2,\dots n_k} = \frac{n!}{n_1!n_2!\dots n_k}$$

6.12.4 Combinaciones

Dada una colección de n elementos, queremos saber de cuantas formas podemos elegir k elementos (no importa el órden)
Dado $1,2,3,4$ y $n = 4$, $k = 2$

- 1,2
- 1,3
- 1,4
- 2,3
- 2,4
- 3,4

6.12.5 Coeficiente binomial

Podemos elegir k elementos de un conjunto de n

De manera formar la obtenemos con la siguiente ecuación:

$$P_k^n = \frac{n!}{k!(n-k)!}$$

En código

```
long long Combinatorics(int n, int k){
    return factorial[ n ] * inverse[ k ] % mod * inverse[ n - k ] % mod
}
```

La ecuación anterior solo funciona si y solo si $0 \le k \le n$ en el caso de que $k > n$ el resultado es cero.

Ahora vamos a tratar de deducir la siguiente fórmula

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

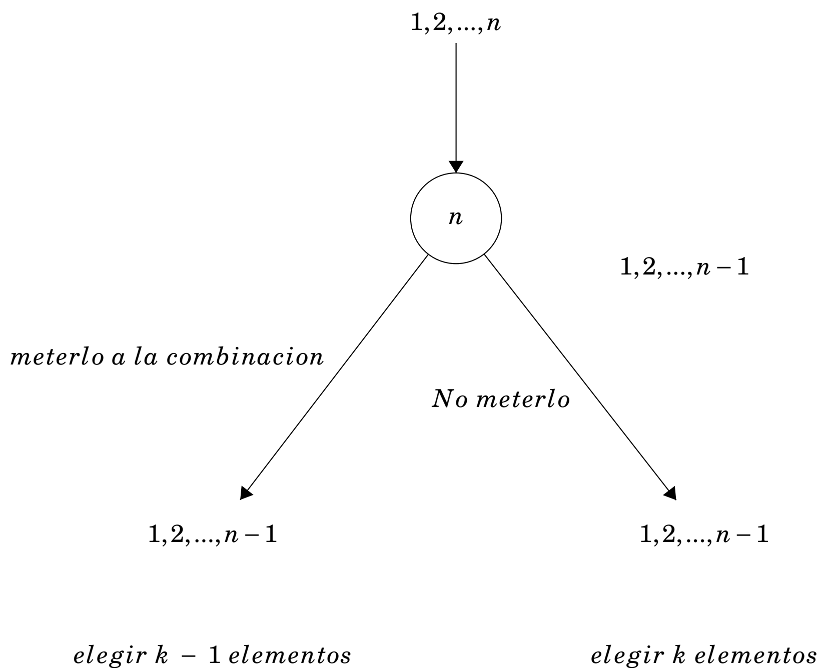
Si y solo si $0 \le k \le n$

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

Si y solo si $n = 0$ o $k = n$

La deducción es la siguiente:

Tenemos un conjunto de tamaño n y queremos elegir k de ellos, si tomamos el iésimo elemento tenemos como opciones elegir k - 1 elementos o bien elegir k elementos



[H]

En código tenemos lo siguiente:

```
long long mem[MAXN][MAXN]; //Inicializado en -1
long long comb( int n, int k ){
    if( n == k || k == 0 )
        return 1;
    if( mem[ n ][ k ] != -1 )
        return mem[ n ][ k ];
    mem[ n ][ k ] = ( comb( n - 1, k ) + comb( n - 1, k - 1 ) ) % mod
    return mem[ n ][ k ];
}
```

Algunas identidades utilizadas en el coeficiente binomial

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

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

$$\binom{n}{n-k} = \frac{n!}{(n-k)!(n-(n-k)!)} = \frac{n!}{(n-k)!k!}$$

6.12.6 Triángulo de pascal

$$\begin{array}{ccccccccccc} & & & & & 1 & & & & & \\ & & & & 1 & 2 & 1 & & & & \\ & & & 1 & 3 & & 3 & 1 & & & \\ & & 1 & 4 & & 6 & & 4 & 1 & & \\ & 1 & 5 & & 10 & & 10 & & 5 & 1 & \end{array}$$

$$\binom{n}{0} \quad \binom{n}{1} \quad . \quad . \quad . \quad \binom{n}{n-1} \quad \binom{n}{n}$$

Identidad del palo de Hockey (Hockey stick)

Usando el triángulo de pascal podemos obtener gráficamente varias propiedades, por ejemplo la identidad del palo de Hockey (Hockey Stick). Formalmente se define como:

$$\sum_{i=0}^{r} \binom{k+i}{i} = \binom{k+r+1}{r}$$

Propiedades

La suma de la n-ésima fila es igual a 2^n
$\sum_{i=0}^{n} \binom{n}{i} = 2^n$

6.12.7 Binomio de Newton

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$$

6.12.8 Biyecciones

Dados dos conjuntos A y B, existe una biyección entre A y B si para cada elemento en A se puede establecer una relación con un y solo un elemento en B y para conjunto en B se puede establecer una relación con un y solo un elemento en A.

Ej:
Sea $A = 0,1,2,3$
Sea $B = 5,6,7,8$
para cada elemento

6.13 Problema de las barras y las estrellas

Si tengo n estrellas y m barras ¿De cuantas formas puedo ordenarlas?, ejemplo:
Tengo 2 estrellas y 2 barras, la solución es 6, es decir:

- **| |

- * | * |
- * | | *
- | ** |
- | * | *
- | | **

O bien:

$$\binom{4}{2} = \frac{4!}{2!2!} = \frac{24}{4*4} = 6$$

Es decir que de forma general tenemos:

$$\binom{n+m}{n} = \binom{n+m}{m}$$

En otras palabras de un arreglo de n + m espacios elegimos n espacios para poner las estrellas

Otra posible solución es considerar esto como una permutacion con repeticiones, donde tenemos dos grupos, uno para barras y otro para estrellas. De forma general es:

$$\frac{(n+m)!}{n!m!}$$

Ahora supongamos que queremos resolver lo siguiente: ¿Cuántas formas hay de asignarle valores enteros no negativos a k variables $(x_1, x_2, \ldots, x_k)$ tal que la suma $x_1 + x_2 + x_3 + \cdots + x_k = n$, primero analicemos un caso más sencillo:
K = 2, n = 6, solución = 7

- 6 + 0 = 6
- 5 + 1 = 6
- 4 + 2 = 6
- 3 + 3 = 6
- 2 + 4 = 6
- 1 + 5 = 6
- 0 + 6 = 6

Si tengo un ordenamiento con n estrellas y $k-1$ barras, consideramos que las barras dividen k bloques, y el valor de x_i será la cantidad de estrellas del i-ésimo bloque, por ejemplo

	|	*	|	*
0		1		1

Es decir:

1. $x_1 = 0$
2. $x_2 = 1$
3. $x_3 = 1$

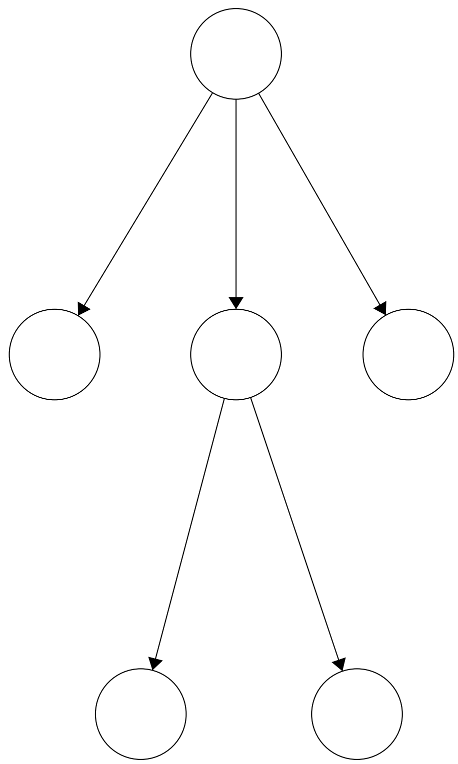
4. $\therefore 0+1+1=2$

Colocamos x_1 extrellas en el primer bloque, luego una barra, luego x_2 estrellas, luego una barra y así sucesivamente hasta haber colocado $k-1$ barras y finalmente x_k estrellas. De forma general:

$$\binom{n+k-1}{n} = \binom{n+k-1}{k-1} = \frac{(n+k-1)!}{n!(k-1)!}$$

Problema

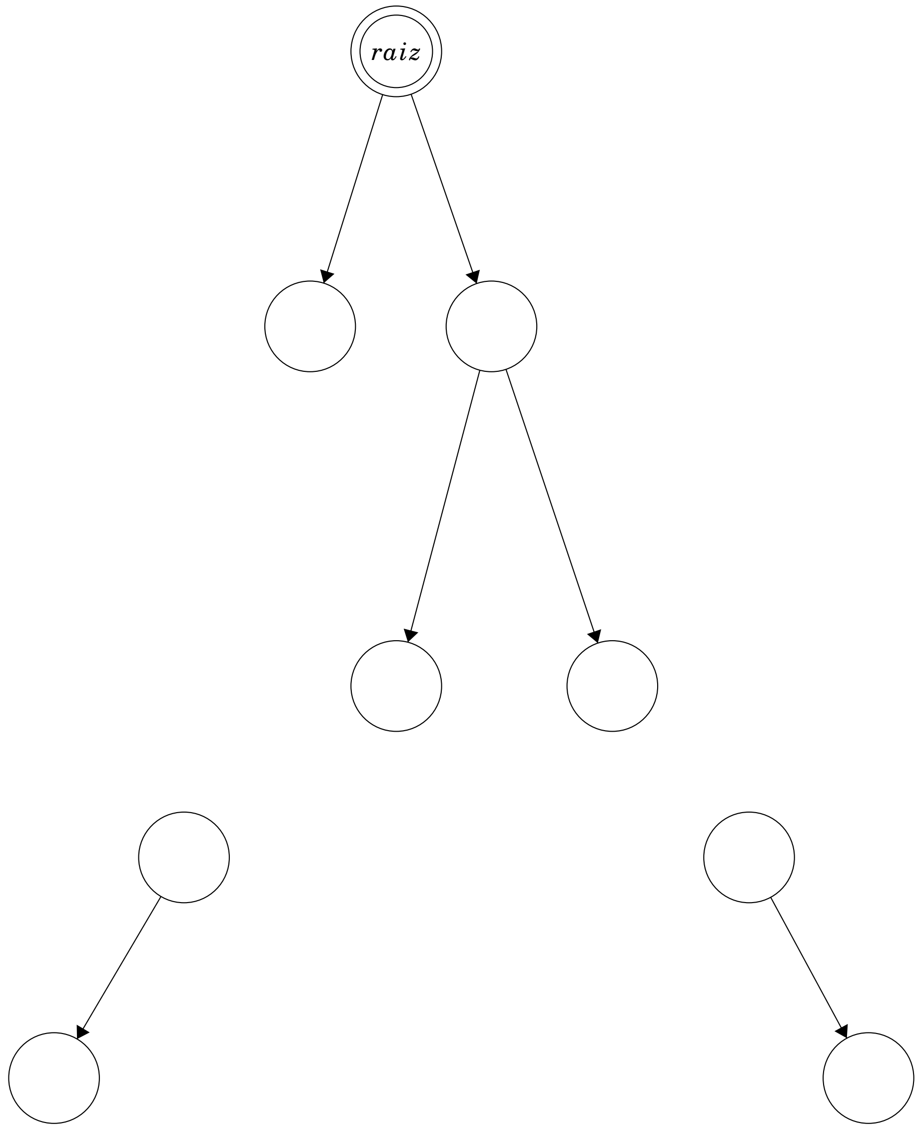
¿Cuántos árboles binarios con n nodos existen?
Primero recordemos que es un árbol:
Un árbol es un tipo especial de grafo, el cual no contiene ciclos además es totalmente conexo, por ejemplo:



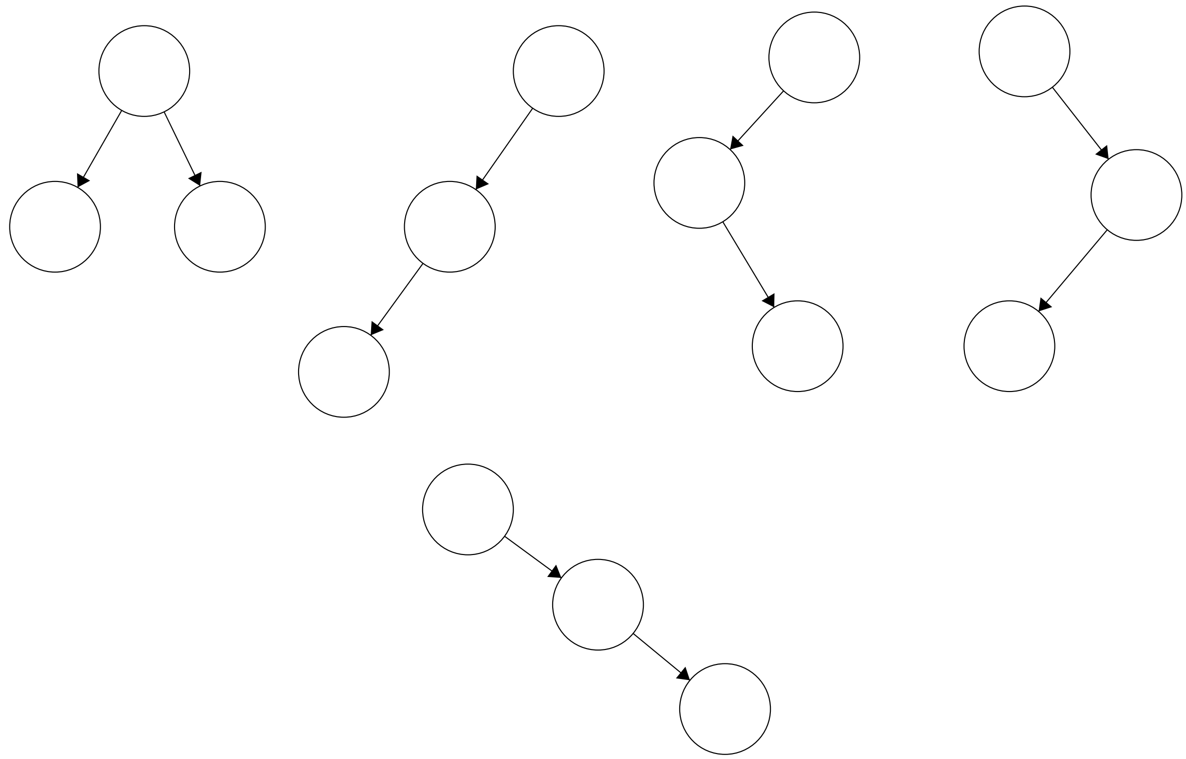
[H]
Árbol binario: es un árbol con raiz y cada nodo tiene a lo más 2 hijos (hijo izquierdo y/o hijo derecho), por ejemplo

Regresando a nuestro problema, hagamos el análisis por casos, primero, ¿Cuántos árboles de cero nodos tenemos? solo 1:

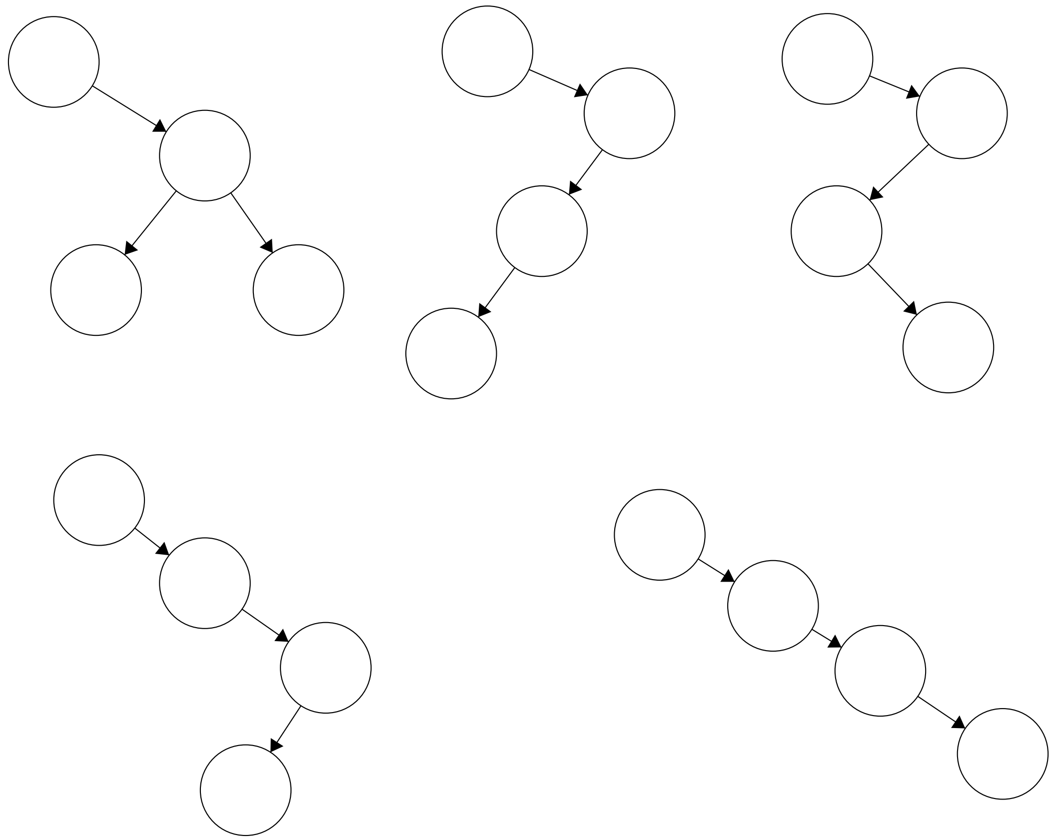
¿Cuántos árboles de 2 nodos, tenemos?



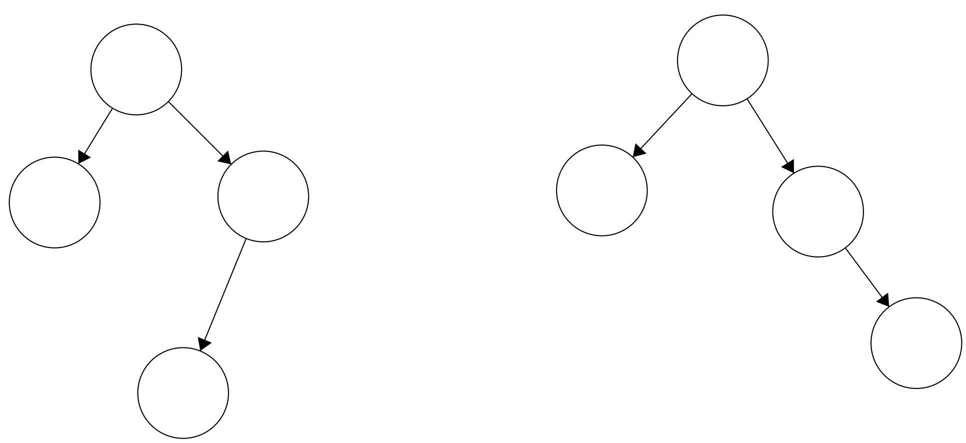
¿Cuántos árboles de 3 nodos, tenemos?



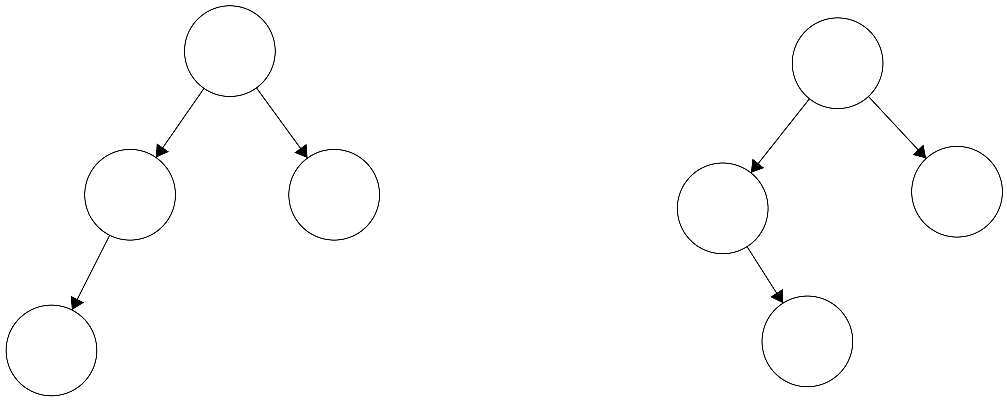
¿Cuántos árboles de 4 nodos tenemos?, de aquí tenemos dos sub casos, el primero un hijo izauierdo con cero nodos y uno derecho con 3 nodos



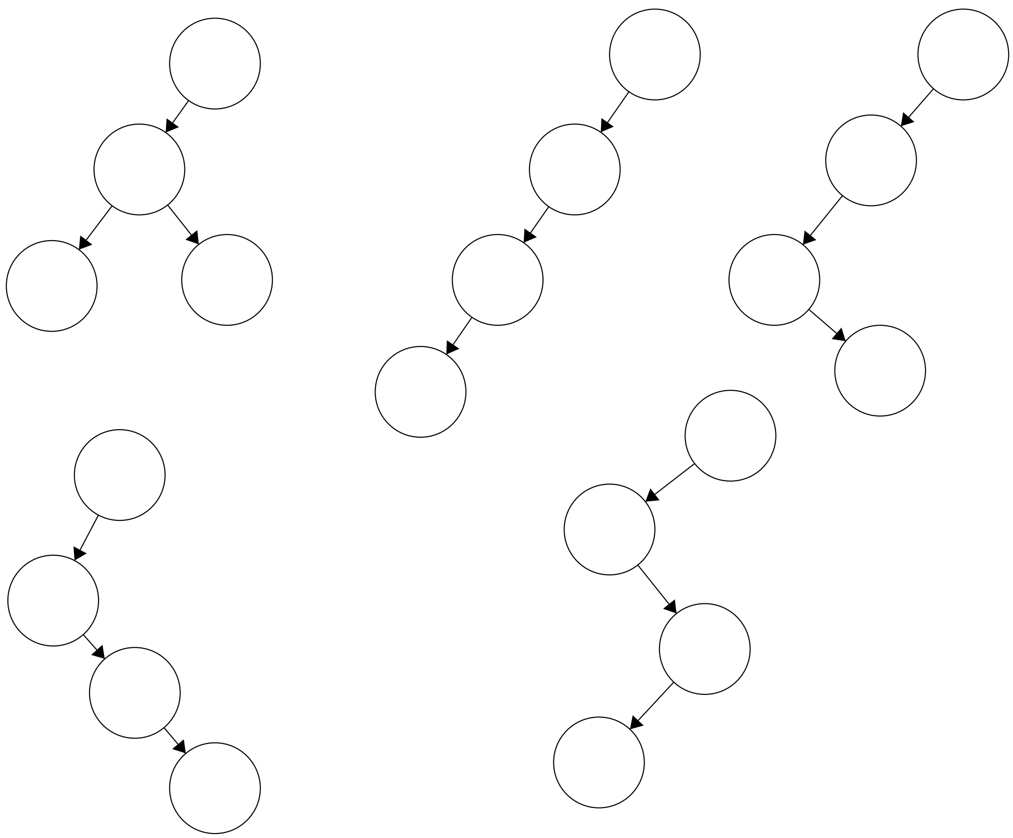
Segundo caso, la raiz tiene un hijo izquierdo con 1 nodo y uno derecho con dos nodos



Otro caso es que el hijo izquierda tenga 2 nodos y el derecho tiene 1



Finalmente, si el hijo izauierdo tiene 4 nodos y el derecho tiene 0



Y con cinco noos, ¿Podemos calcular rápidamente todo?

- I = 0, D = 4 (1)(14) = 14
- I = 1, D = 3 (1)(5) = 5
- I = 2, D = 2 (2)(2) = 4
- I = 3, D = 1 (5)(1) = 5
- I = 4, D = 0 (14)(1) = 14

La suma de todo es 42

Por tanto lo que hemos hecho es

$$f(n) = f(0)f(n-1) + f(1)f(n-2) + f(2)f(n-1) + f(3)f(n-3) + \cdots + f(n-2)f(2) + f(n-1)f(0) = \sum_{i=0}^{n-1} f(i)f(n-i-1)$$

Esta fórmula nos da los **números de Catalan**, en código tenemos lo siguiente

```
long long f[ 1000 ] = { 0 };
f[ 0 ] = 1;
for( int n = 0; n < 1000; n++ )
    for( int i = 0; i <= n - 1; i++ )
        f[ n ] = ( f[ n ] + f[ i ] * f[ n - i - 1 ] ) % mod;
```

Otras definiciones de los números de Catalán son:

$$C_n = \sum_{i=0}^{n-1} C_i * C_{n-i-1}$$

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

$$C_n = \binom{2n}{n} - \binom{2n}{n+1}$$

Relaciones de recurrencia

Una relación de recurrencia para una suceción de números $a_0, a_1, a_2, \ldots, a_n$ es una expresión que relaciona cada término a_n con los términos anteriores $a_0, a_1, \ldots, a_{n-1}$ para cualquier $n \geq m$ teniendo m valores iniciales. Por ejemplo
Como ejemplos tenemos Fibonacii

- f(0) = 0
- f(1) = 1
- f(n) = f(n - 1) + f(n - 2), $n \geq 2$

También tenemos el factorial

- 0! = 1
- n! = n * (n - 1), $n \geq 1$

Números de catalán:

- $C_1 = 1$
- $C_n = \sum_{i=1}^{n-1} C_i C_{n-i}$, $n \geq 2$

Problema de las torres de Hanoi

- $T_1 = 1$
- $T_n = 2T_{n-1} + 1$, $n \geq 2$

Problema de Josefo

- $J(2) = 1$
- $J(2k) = 2J(k) - 1$
- $J(2k+1) = 2J(k) + 1$

Para las demostraciones y más información: Concrete mathematics, Donald knuth, Graham, Patashnik

Recurrencias lineales coeficientes constantes y homogeneas

$a_n = C_1 a_{n-1} + C_2 a_{n-2} + C_3 a_{n-3} + \cdots + C_d a_{n-d} + g(n), n \geq d$
dónde $C_1, \ldots, C_d$ son constantes y $C_d \neq 0$ y $g(n) = 0$

Decimos que la recurrencia es de órden d

Resolver la recurerencia

Hallar el n-ésimo término, podemos utilizar:

- Programación dinámica:
  1. recursiva
  2. iterativa
- Hallar una fórmula cerrada, por ejemplo usando Fibonacci
  - $F(0) = 1$
  - $F(n) = 2F(n-1)$, $n \geq 2$
  - $F(n) = 2^n$
- Matrices O($d^3 log(n)$)
- Ecuaciones características O($d^2 log(n)$)
- Fourier ($O(dlog(n)log(d))$)

Para entenderlo mejor vamos a tratar de resolver con el siguiente ejeplo:

- $a(0) = 3$
- $a(1) = 2$
- $a(2) = -3$

De forma general:

$$a(n) = a(n-1) + 3a(n-3), n \geq 3, d = 3$$

```
int find_nth_term( int n ){
    vector< int > a_last( 3 );
    a_last[ 0 ] = -3;
    a_last[ 1 ] = 2;
    a_last[ 2 ] = 3;
    for( int i = 3; i <= n; i++ ){
        int new_t = a_last[ 0 ] + 3 * a_last[ 2 ];
        for( int j = 2; j > 0; j -- )
            a_last[ j ] = a_last[ j - 1 ];
        a_last[ 0 ] = new_t;
    }
    return a_last[ 0 ];
}
```

6.13.1 Matrices

Matrices cuadradas

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

Multiplicación de matrices

$$\begin{pmatrix} 3 & 0 \\ 2 & 1 \end{pmatrix} * \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} 3*0+0*2 & 3*2+0*3 \\ 2*0+1*1 & 2*2+3*1 \end{pmatrix} = \begin{pmatrix} 0 & 6 \\ 1 & 7 \end{pmatrix}$$

Debemos tomar en cuenta que:

- No es conmutativa es decir $A * B \neq B * A$
- El producto es asociativo es decir $A * B * C = (A * B) * C = A * (B * C)$
- Hay una matriz identidad tal que $A * I = A$

dónde:

$$I = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & \ddots & \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

Aquí tenemos un pequeño programa para multiplicar matrices.

```
typedef vector< vector< int > > Matrix;
Matrix multiply( Matrix &A, Matrix &B, int n ){
    Matrix C;
    C.assign(n, vector< int >(n, 0));
    for( int i = 0; i < n; i++)
        for( int j = 0; j < n; j++ )
            for( int k = 0; k < n; k++ )
                C[ i ][ j ] += A[ i ][ k ] * B[ k ][ j ];
    return C;
}
```

Ahora tratemos de resolver el ejecicio que ya se ha hecho anteriormente pero en lugar de utilizar programación dinámica utilicemos matrices. Recordemos que tenemos

- a(0) = 3
- a(1) = 2
- a(2) = -3

$$a(n) = a(n-1) + 3 * a(n-3)$$

$$\begin{bmatrix} -3 \\ 2 \\ 3 \end{bmatrix} \to \begin{bmatrix} 6 \\ -3 \\ 2 \end{bmatrix} \to \begin{bmatrix} 12 \\ 6 \\ -3 \end{bmatrix} \to \begin{bmatrix} 3 \\ 12 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} -3 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ -3 \\ 2 \end{bmatrix}$$

Si repetimos vamos a realizar esta multiplicación $n-3+1$ veces, es decir

$$\begin{bmatrix} 1 & 0 & 3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^{n-3+1} \begin{bmatrix} 3 \\ 2 \\ -3 \end{bmatrix}$$

La complejidad de esto es $O(d^3 n)$ Pero podemos optimizar utilizando la idea ade la exponenciación binaria

```
Matrix matrixPow( Matrix &b, int p ){
    Matrix ans = Identidy( n );
    while( p ){
        if( p & 1 )
            ans = multiply( ans, b );
        b = multiply( b, b );
        p /= 2;
    }
    return ans;
}
```

Este algoritmo tiene una complejidad de $O(log(n)d^3)$