[Γα=Ω5]

Big O (Complejidad de Algoritmos)

Por Ariel Parra

[Γα=Ω5]

¿Por qué analizar un algoritmo?

La razón más sencilla para analizar un algoritmo es descubrir sus características para evaluar su uso para diversas aplicaciones o compararlo con otros algoritmos para la misma aplicación. Además, el análisis de un algoritmo puede ayudarnos a comprenderlo mejor y puede sugerir mejoras informadas. Los algoritmos tienden a volverse más cortos, más simples, elegantes y más eficientes.

Eficiencia

La eficiencia de los algoritmos consiste en el tiempo de ejecucion y la cantidad de recursos consumidos, en especial el uso de la memoria.

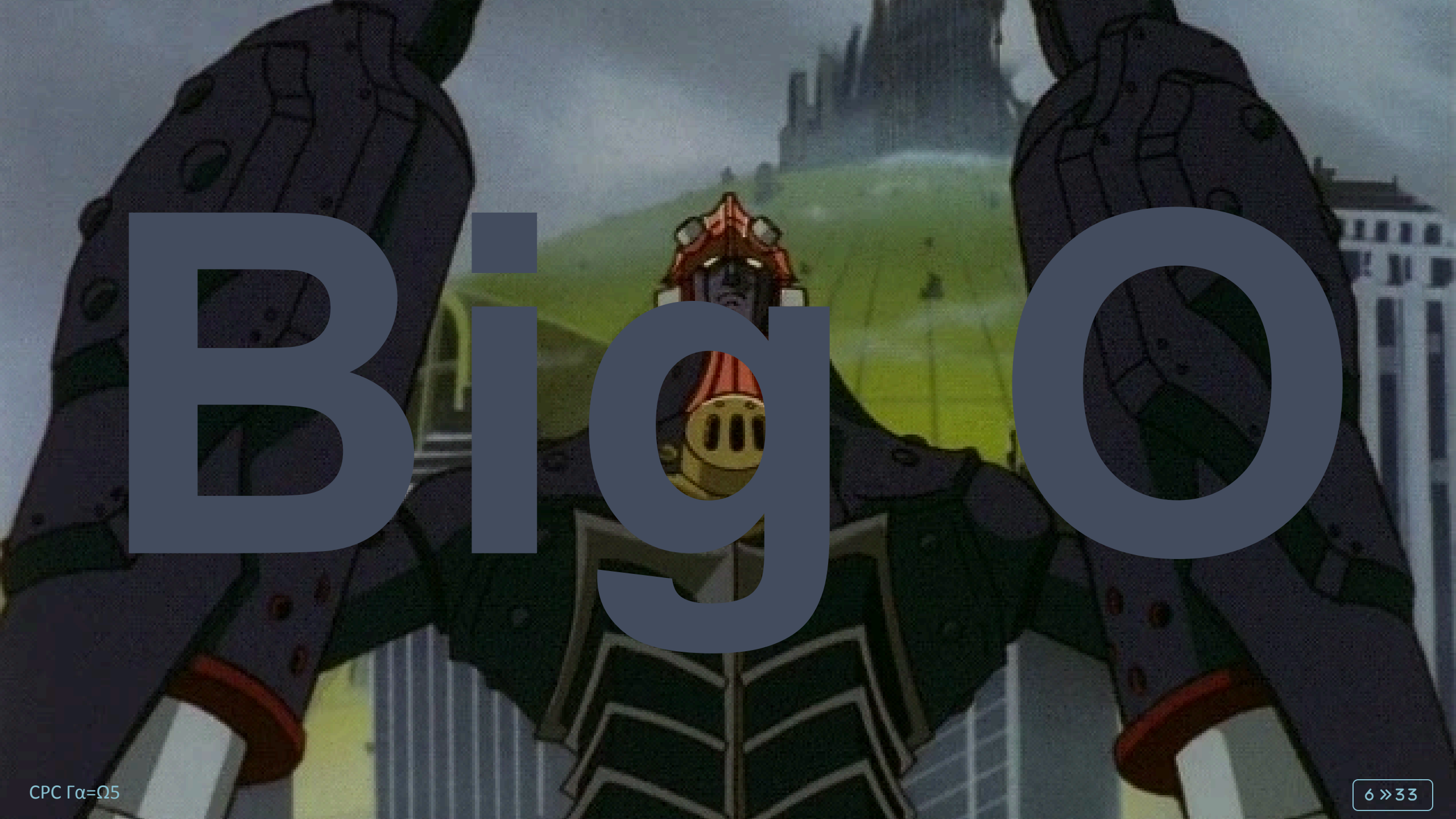
Esta se puede medir en funcion de su eficiencia, el costo de escribirlo, leerlo y modificarlo.

Complejidad en el Tiempo

En la mayoría de los algoritmos, el tiempo de ejecución depende de la cantidad de elementos y no de su magnitud.

Por ejemplo: **[1000000000,20000000000000,3000000000000] < [1,2,3,4,5,6,7,8,9,10]**

Todos los algoritmos trabajan con datos de entrada de distintos tamaños. Para evaluar la eficiencia o "velocidad" de un algoritmo en función del número de entradas, utilizamos una función que describe el tiempo de ejecución, $T(n)$, que representa el tiempo en función de las operaciones elementales realizadas por el algoritmo y se expresa sin unidades.


Big O

The Big O es un anime japones de 1999 creada por Keiichi Satō, dirigida por Kazuyoshi Katayama y producida por los estudios Sunrise. El equipo de guionistas de la serie fue dirigido por Chiaki J. Konaka, famoso por su trabajo en Serial Experiments Lain y Hellsing.

La historia de la serie está ambientada cuarenta años después de que un extraño suceso provocara que los habitantes de Ciudad Paradigma perdieran la memoria. El protagonista de la serie es Roger Smith, el mejor Negociador de la ciudad. Roger presta este "tan solicitado servicio" con la ayuda de una androide llamada Dorothy Wayneright y de su Mayordomo, Norman Burg. Cuando surge la necesidad, Roger invoca al Big O, una reliquia gigantesca relacionada con el pasado...

Notación Big O

La notación Big O define una cota superior para la función de tiempo (T(n)) a partir de una función (f(n)). La función (f(n)), también conocida como (O(g(n))), es una función que siempre es mayor o igual que (T(n)) para valores suficientemente grandes de (n). En la imagen, la cota superior es representada por (c \cdot g(n)) y (f(n)) representa nuestra función de tiempo (T(n)). En otras palabras, definimos una función (f(n)) que sirve como una cota superior asintótica para (T(n)), indicando que (T(n)) no crecerá más rápido que (f(n)) a medida que (n) se hace grande.

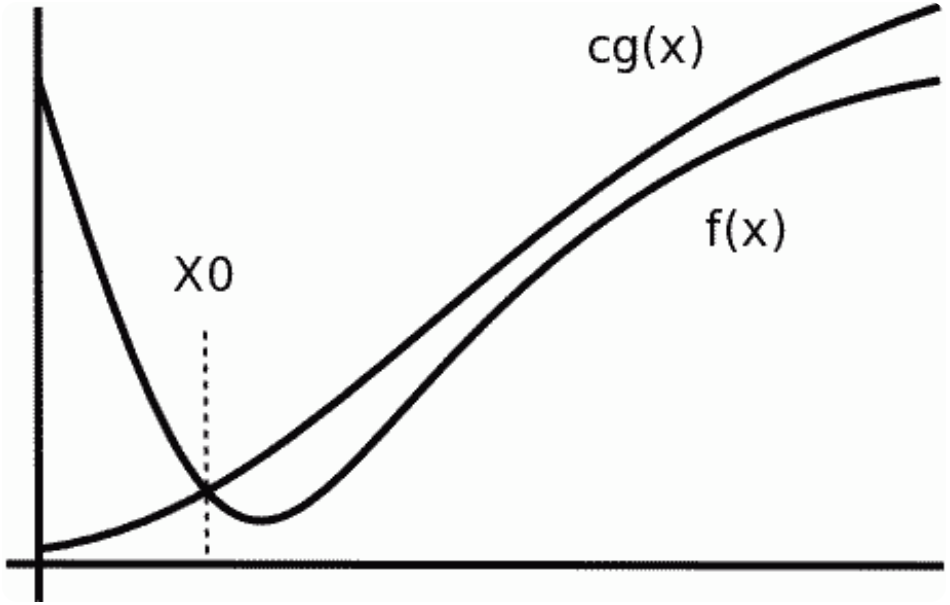


Tabla de funciones

notación	nombre de la función
O(1)	Constante
O(log n)	Logarítmica
O(n)	lineal
O(n · log n)	Lineal logarítmica o casi-lineal
O(n^2)	Cuadrática
O(n^k)	Potencial (`k` siendo constante y `k` > 1)
O(k^n)	Exponencial (`k` usualmente siendo `2` y `n` > 1)
O(n!)	Factorial

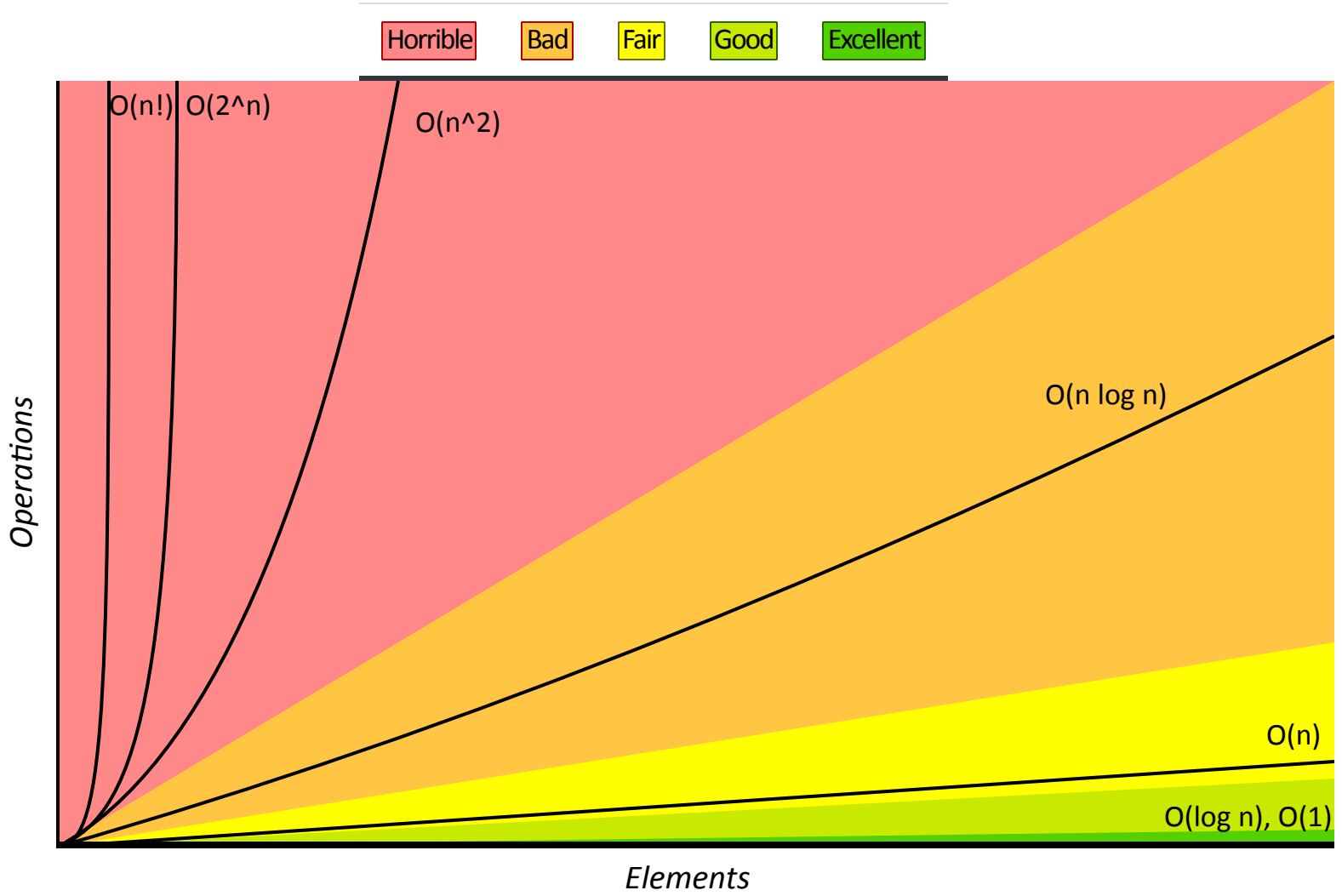

Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Tabla de notaciones

Notación	Tipo de cota	Descripción
O (Big O)	Cota asintótica superior (≥)	Describe el **peor de los casos** de un algoritmo.
o (small o)	Límite superior estricto (>)	Una función crece más lentamente que otra función .
Ω (Big Omega)	Cota asintótica inferior (≤)	Describe el **mejor de los casos** de un algoritmo.
ω (small omega)	Límite inferior estricto (<)	Una función crece más rápidamente que otra función.
Θ (Big Theta)	Cota asintótica exacta (==)	Describe el **caso promedio** de un algoritmo.

Analisis de complejidad por función

Para determinar la complejidad temporal de un algoritmo, se analiza el tiempo de ejecución en función del tamaño de la entrada n sumando las complejidades de cada operación principal. Luego, se simplifica la expresión combinando términos y eliminando constantes y términos de menor orden, enfocándose en el término dominante que crece más rápido respecto a `n`.

Los pasos para realizar un análisis de la complejidad son:

1. **Identificar Operaciones Principales** Detecta las operaciones clave del algoritmo (bucles, llamadas recursivas).
2. **Determinar Complejidad de Cada Operación** Asigna notación Big O a cada operación principal (ej. (O(1)), (O(n)), (O(n^2))).
3. **Suma de Complejidades** Suma las complejidades de las operaciones principales.
4. **Simplificación de la Expresión** Combina términos y elimina constantes y términos de menor orden.
5. **Enfocar en el Término Dominante** Identifica el término que crece más rápido con respecto a (n).
6. **Escribir la Complejidad Asintótica** Expresa la complejidad en notación Big O.

Ejemplo de complejidad O(1):

```
int n=1000;          //es O(1), ya que n ya esta declarada
if(n%2==0)           //O(1)
    cout<<"par";     //O(1)
else                 //O(1)
    cout<<"impar";   //O(1)
```

Complejidad = 1 + 1 + 1 + 1 + 1 = O(5) = O(1);

Ejemplo de complejidad O(n):

```
int n=0;                        // O(1)
cin>>n;                         // O(1)
for(int i=0;i<n;i++){           //es O(n) ya que n es el limite
    if(i%2==0)                  // O(n)
        cout<<i<<" es par";     // O(n)
    else                        // O(n)
        cout<<i<<" es impar";// O(n)
}
```

Complejidad = 1 + 1 + n + n + n + n + n = O(2 + 5n) = O(n);

Otro ejemplo de complejidad O(n):

```
int n=0, j=0;                              //O(1)
for(int i=0;i<n;i++){                      //O(n)
    if(i%2==0)                             //O(n)
        cout<<i<<" es par";                //O(n)
    else                                   //O(n)
        cout<<i<<" es impar";              //O(n)
}
while(j<n){                                //O(n)
    if(i%3==0)                             //O(n)
        cout<<i<<" es mutliplo de 3";      //O(n)
    else                                   //O(n)
        cout<<i<<" no es multimplo d";     //O(n)
    j++;                                   //O(n)
}
```

Complejidad = 1 + n + n + ... + n = O(1+ 11n) = O(n)

Ejemplo de complejidad O(log(n)):

```
int n=0;                        //O(1)
cin>>n;                         //O(1)
for(int i=0;i<n;i*=2){          //O(log (n)) ya incrementa con multiplicaciones en lugar de sumas
    if(i%2==0)                  //O(log (n))
        cout<<i<<" es par";     //O(log (n))
    else                        //O(log (n))
        cout<<i<<" es impar"; //O(log (n))
}
```

Complejidad = 1 + 1 + log(n) + log(n) + log(n) + log(n) + log(n) = O(2 + 6log(n)) = O(log(n))

Ejemplo de complejidad O(n^2):

```
int n=0;                    //O(1)
cin>>n;                     //O(1)
for(int i=0;i<n;i++){       //O(n)
    for(int j=0;j<n;j++){ //O(n^2)
        cout<<i<<j<<endl; //O(n^2)
    }
}
```

Complejidad = 1 + 1 + n + n * n + n^2 = O(2 + n + 2n^2) = O(n^2)

Ejemplo de complejidad O(n^k):

```
int n=0;                                                                    //O(1)
cin>>n;                                                                     //O(1)
for(int i=0;i<n;i++)                                                        //O(n)
    for(int j=0;j<n;j++)                                                    //O(n^2)
        for(int ca=0;ca<n;ca++)                                             //O(n^3)
            for(int cb=0;cb<n;cb++)                                         //O(n^4)
                ...                                                         //O(n^5)
                    ...                                                     //O(n^6)
                        for(int k=0;k<n;k++)                                //O(n^7)
                            cout << i << j << ca << cb << ... << ... << k; //O(n^k)
```

Complejidad = 1 + 1 + n + n * n + n^3 ... + n^k = O(2 + n + n^2 + ... n^k) = O(n^k)

Ejemplo de complejidad O(nlog(n)):

```
int n=0;                                //O(1)
cin>>n;                                 //O(1)
for(int j=0;j<n;j++){                   //O(n)
    for(int i=0;i<n;i*=2){              //O(n(log(n)))
        if(i%2==0)                      //O(n(log(n)))
            cout<<i<<" es par";         //O(n(log(n)))
        else                            //O(n(log(n)))
            cout<<i<<" es impar";       //O(n(log(n)))
    }
}
```

Complejidad = 1 + 1 + n + n * log(n) + n(log(n)) + n(log(n)) + n(log(n)) + n(log(n))

= O(2 + n + 5n(log(n))) = O(nlog(n))

Recursividad complejidad O(k^n)

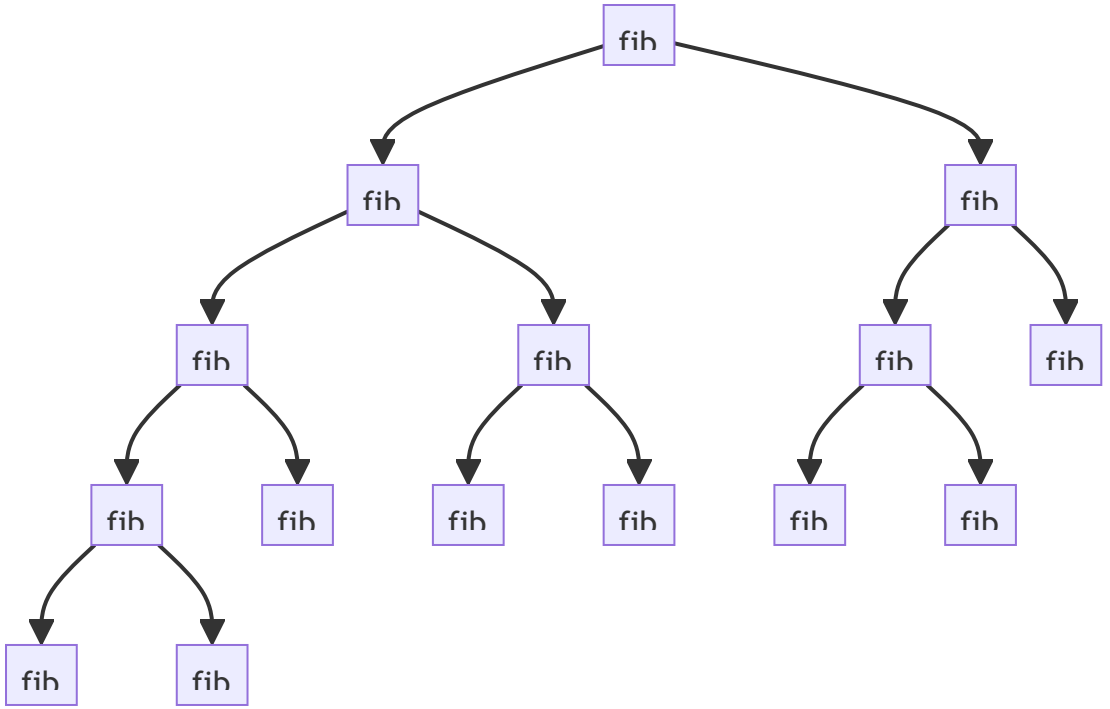
Muchos algoritmos recursivos tienen una complejidad de O(k^n), donde `k` es el número de llamadas recursivas en cada nivel de la recursión y `n` es la profundidad de la recursión. Esto es común en algoritmos que se pueden representar como un árbol de recursión, como lo seria el cálculo de la secuencia de Fibonacci.

Ejemplo:

```
int fibonacci(int n){                          //O(2^n)
    if(n<=1)                                   //O(1)
        return 1;                              //O(1)
    return fibonacci(n-1) + fibonacci(n-2); //O(2^n)
}
```

Gráfica del algoritmo recursivo de fibonacci

para `n = 5`



Complejidad en el espacio

La complejidad en el espacio mide la cantidad de memoria que un algoritmo requiere en función del tamaño de la entrada. Es un aspecto crucial del análisis de algoritmos, para evaluar la eficiencia de un algoritmo.

Ejemplos de complejidad en el espacio

```cpp
int a = 2; //O(1) de espacio
int n; cin >> n; //O(1) de espacio
int* vec = new int[n]; //O(n) de espacio
vector<int> v(n); //O(n) de espacio
vector<vector<int>> mat(n,vector<int>(n)); //O(n^2) de espacio
int** matrix = new int*[n]; //O(n) de espacio
for (int i = 0; i < n; ++i)
    matrix[i] = new int[n];//O(n^2) de espacio
```

Ejercicios de Notación Big O

1

```
string entrada;
cin>>entrada;
int x=5;
if(entrada=="holi")
    cout << x;
```

2

```
int n;
cin>>n;
for (int i=1; i<=n; i++)
    cout << "hola";
```

1 Solucion

```
string entrada;      // O(1)
cin>>entrada;        // O(1)
int x=5;             // O(1)
if(entrada=="holi") // O(1)
    cout<<x;         // O(1)
```

2 solucion

```
int n;                    //O(1)
cin>>n;                   //O(1)
for (int i=1; i<=n; i++) //O(n)
    cout<<"hola";         //O(n)
```

3

```
int n, c=5; cin>>n;
for (int i=1; i<=n; i+=c)
    for (int j=1; j<=n; j+=c)
        cout<<i+j;
```

4

```
int n;in>>n;
n=2;
for(int i=0;i<n;i++)
    for(int j=0;j<n;j++)
        for(int k=0;k<n;k*=j)
            cout<<i*j*k;
```

3 solucion

```
int n, c=5; cin>>n;              //O(1)
for (int i=1; i<=n; i+=c)        //O(n)
    for (int j=1; j<=n; j+=c)    //O(n^2)
        cout<<i+j;               //O(n^2)
```

4 solucion

```
int n;                           //O(1)
cin>>n;                          //O(1)
n=2;                             //O(1)
for(int i=0;i<n;i++)             //O(1)
    for(int j=0;j<n;j++)         //O(1)
        for(int k=0;k<n;k*=j)    //O(1)
            cout<<i*j*k;         //O(1)
```

5

```
int n,c=2;
cin>>n;
for(int i=0;i<n;i*=c){
    cout<<i;
}
```

6

```
int n,c=5,i=10;
cin>>n;
for(i=0;i<n;pow(i, c))
    cout<<i;
```

5 solucion

```
int n,c=2;                  //O(1)
cin>>n;                     //O(1)
for(int i=0;i<n;i*=c){      //O(log(n))
    cout<<i;                //O(log(n))
}
```

6 solucion

```
int n,c=5,i=10;             //O(1)
cin>>n;                     //O(1)
for(i=0;i<n;pow(i, c))      //O(log(log(n)))
    cout<<i;                //O(log(log(n)))
```

Problemas

Resuelve los siguientes problemas escribiendo la complejidad de espacio y tiempo en notación de Big O

- **158A** Next Round ⮭
- **1598A** Computer Game ⮭

Referencias

- Arte de programar. (2020). ⏳ *¿Qué es la complejidad en el tiempo?*. Recuperado de https://youtu.be/CtpvpnYNNiE?si=TqTTLfa91imUG4t6
- Chio Code. (2020). *Notación Big O | Análisis de algoritmos de forma sencilla*. Recuperado de https://youtu.be/MyAiCtuhiqQ?si=bOD2ZuWLhsoZpJej
- Computerphile. (2013). *Getting Sorted & Big O Notation - Computerphile* [video]. Recuperado de https://youtu.be/kgBjXUE_Nwc?si=Q8MeZJ9QSFE2qc2d
- freeCodeCamp.org. (2021). *Big O Notation - Full Course* [video]. Recuperado de https://youtu.be/Mo4vesaut8g?si=AUGWnTML6DZFw3nz

- kumar, H. (2024). *Design and Analysis of Algorithms*. Recuperado de https://www.geeksforgeeks.org/design-and-analysis-of-algorithms/
- NeetCode. (2022). *Big-O Notation - For Coding Interviews*. Recuperado de https://youtu.be/BgLTDT03QtU?si=DQvYH0odkkJIRb2Y
- Rowell, E. (2019). *Know Thy Complexities!*. Recuperado de https://www.bigocheatsheet.com/
- Sedgewick,R. & Flajolet, P. (2022). *Analysis of Algorithms*. Recuperado de https://aofa.cs.princeton.edu/10analysis/
- Tom Scott. (2020). *Why My Teenage Code Was Terrible: Sorting Algorithms and Big O Notation* [video].Recuperado de https://youtu.be/RGuJga2Gl_k?si=OXB88anP81nx3lV5