Ordenes de crecimiento:

Constante:

-El orden de crecimiento constante tiene una complejidad O(1).

-Ejemplo: Una función que suma un número y una constante:

Suma(n)=n+c

-Se clasifica así debido a que la operación no depende del tamaño de los datos, es decir, independientemente de la entrada (en este caso un número) solo se le sumará una constante. Es el caso más favorable, pero también uno de los menos frecuentes.

-Un algoritmo entra en esta clasificación si la salida no depende del tamaño de la entrada. Un algoritmo de este tipo no tiene llamadas recursivas ni ciclos que se deban ejecutar más veces si crece la entrada

Logarítmico:

-El orden de crecimiento logarítmico tiene complejidad O(log(n))

-Ejemplo: Algoritmo de búsqueda binaria

-En este algoritmo se encuentra un valor en un array, dividiéndolo y descartando la mitad en la que el valor no puede estar

-Este algoritmo se clasifica así ya que no utiliza la entrada completa, sino que la va dividiendo a conveniencia.

-Un algoritmo suele tener este orden de crecimiento si divide el problema en “trozos” cada vez menores para abordarlo. Supongamos que en el ejemplo anterior n=16(el array tiene 12 elementos). En el peor caso, se dividirá el array de la siguiente manera

Primera ejecución:8 elementos

Segunda ejecución:4 elementos

Tercera ejecución:2 elementos

Cuarta ejecución: 1 elemento

Sin embargo, si tuviera 32 elementos tendría una ejecución más, y si tuviera 8 una ejecución menos. Podemos observar que C=⌊log(L)⌋ donde C=cantidad de ejecuciones y L=cantidad de elementos del array. Como el algoritmo divide a la mitad el problema, se puede decir que 2^C=L. Esto debido a que para aumentar

en uno la cantidad de ejecuciones, la entrada debe duplicar su tamaño, ya que se descartará la mitad de la entrada, sean 2 o 2000 valores. Este cambio drástico en la cantidad de elementos “troceados” es la que explica el crecimiento logarítmico

Lineal:

-El orden de crecimiento lineal tiene una complejidad O(n).

-Ejemplo: una función que suma todos los números desde 1 hasta n

-Se clasifica así debido a que el tiempo necesario para ejecutar el algoritmo es directamente proporcional al tamaño de la entrada. Cada aumento en el input implica un aumento constante en el output

-Un algoritmo entra en esta clasificación cuando hay una relación lineal entre el tiempo de ejecución y el tamaño de la entrada. Se pueden identificar estos algoritmos si el número de operaciones elementales por ejecutar aumenta una dada cantidad por cada aumento en el tamaño del input. Por ejemplo, en el algoritmo anterior, si n=3 se ejecutarán dos sumas, si n=4 se ejecutarán tres sumas y así sucesivamente.

Cuadrático:

-El orden de crecimiento logarítmico tiene complejidad $O(n^2)$

-Ejemplo: algoritmo selection sort

Se trata de un algoritmo de ordenamiento que puede ser descrito de esta manera:

```
para i=0 hasta n-1
    para j=i+1 hasta n
        si lista[j] < lista[i] entonces
            intercambiar(lista[i], lista[j])
        fin si
    fin para
fin para
```

-Este algoritmo se clasifica así debido a que el output es proporcional al cuadrado del input. Esto se debe al ciclo anidado. Para cada entrada primero se ejecuta el bloque externo, por cada ejecución del bloque externo se dan n ejecuciones del bloque interno. Luego de ejecutar n veces el bloque externo, se habrán ejecutado n*n=n^2 veces el bloque interno, lo que explica que tenga crecimiento cuadrático

-Un algoritmo entra dentro de esta clasificación si para una entrada de tamaño n el algoritmo debe ejecutar dos subrutinas donde por cada ejecución de una se hacen n ejecuciones de la otra. Estos algoritmos son comunes en ciclos anidados de segundo orden y operaciones relacionadas con ciertas estructuras, por ejemplo, matrices bidimensionales.

Exponencial:

-El orden de crecimiento exponencial tiene complejidad O(2^n)

-Ejemplo: cálculo de raíces por el método de Heron.

El método de Herón fue desarrollado por el matemático Herón como una forma de calcular el n-ésimo término de raíces cuadradas d. Se calcula mediante la fórmula:

Raiz (0)=1

Raíz(n)= (x / (Raíz x (n-1)) + (Raíz x (n-1))) / 2

Donde n es el n-ésimo término de la raíz y x es el número al cual se le está calculando la raíz

-Este algoritmo se debe calcular mediante una función recursiva. Debido a que la llamada recursiva tiene más de una llamada recursiva dentro, evidentemente conforme aumente n la cantidad de llamadas aumentará de forma exponencial. Es decir, si n=2, la cantidad de llamadas (E) será

E= 2*Raíz (1)

Raíz (1): 2*raíz (0)=2

=>E=2*2=4,

Pero si n=3:

E=2*Raíz (2)

Raíz (2) =2**Raíz (1)

Raíz (1): 2*raíz (0)=2

=>E=2*2*2=8

Se puede apreciar un crecimiento exponencial

-Un algoritmo tiene un crecimiento exponencial si conforme aumenta el tamaño de la entrada el tiempo de ejecución aumenta de manera exponencial. Es bastante ineficiente para grandes entradas, por lo que es poco recomendable usar algoritmos con este crecimiento. Este crecimiento es común en funciones

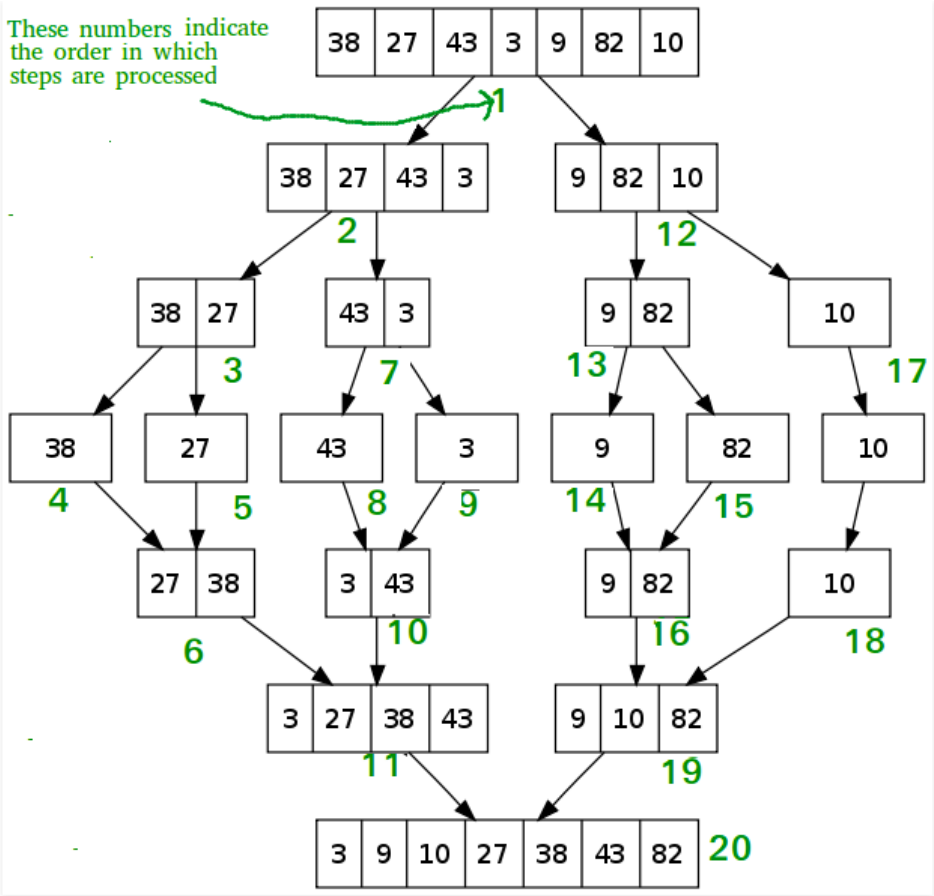
recursivas donde se hace más de una llamada recursiva por cada ciclo. Otro ejemplo de un algoritmo de crecimiento exponencial sería la sucesión de Fibonacci calculada de manera recursiva, ya que está tiene dos llamadas recursivas.

Lineal Logarítmico:

-El orden de crecimiento lineal logarítmico tiene complejidad O(n*log(n))

-Por ejemplo, el algoritmo de ordenamiento Merge Sort, tiene este tipo de complejidad. Además de Heap Sort, Quick Sort, y ciertas optimizaciones de algoritmos que se basan en el principio de “Divide y Conquistarás “y poseen una complejidad o(n^2).

El algoritmo Merge Sort es un algoritmo recursivo que se basa en el principio de Divide y vencerás. De esta manera recibe un array de números y lo primero que procede a hacer es dividir en dos partes este array. Luego cada uno funciona como input para volver a aplicar el algoritmo, es decir que se va a ir dividiendo el array en dos partes continuamente hasta que posean un solo elemento, luego de esto se retrocede acomodándolos de acuerdo si son mayores o menores y uniéndolos con base a esto. Se puede visualizar en el siguiente ejemplo.



El primer paso de dividir los arrays en mitades, toma un tiempo de O (1).

El tercer paso, que conlleva agrupar los datos, toma un tiempo de O(n).

El “árbol”, ya que como se puede notar en la figura semeja a un árbol, tiene un tamaño de más de log(n), es decir que los pasos se tienen que repetir log(n) veces. Lo que resulta en que el algoritmo tenga una complejidad de O(n*log(n)).

Por lo general, los algoritmos que poseen este tipo de complejidad son porque se realiza algún tipo de división y se aplica de nuevo sobre estas divisiones. Esta complejidad es bastante eficiente, y muchos algoritmos que son modificados para ser más eficientes poseen este tipo de complejidad.

Factorial:

-El orden de crecimiento factorial tiene una complejidad O(¡n!)

-Ejemplo: El algoritmo para calcular todas las posibles permutaciones dada una cantidad de caracteres. Es decir que se ingresan n cantidad de caracteres y se calcula todas las combinaciones que se pueden hacer con ellos.

Si se cuentan la cantidad de llamadas al realizar estas combinaciones, se puede notar que para cada nivel siguiente es el resultado de n*(n-1). Esto es así porque de n caracteres ingresados, no se puede tomar en cuenta de nuevo los caracteres ya utilizados, por eso para el primer nivel de combinaciones para cada uno solo pueden hacer (n-1) veces, ya que se resta el carácter ya utilizado. Para el segundo nivel para cada grupo solo se pueden hacer (n-2) combinaciones ya que ya se utilizaron 2 caracteres. Esto se hace recursivamente hasta que ya se hayan hecho todas las combinaciones.

En resumen se puede decir que para el primer nivel hay n*(n-1) llamadas totales, para el nivel 2 hay n*(n-1) *(n-1) llamadas totales. Y así sucesivamente. Esto describe un comportamiento de n!

Por lo general los algoritmos que impliquen realizar combinaciones de datos conllevan una complejidad de O(n!). Estos algoritmos son muy ineficientes ya que para una cantidad pequeña de datos se tarda una gran cantidad de tiempo. Así que los algoritmos con esta complejidad no son usualmente utilizados.

Cúbico:

- El orden de crecimiento cúbico tiene una complejidad $O(n^3)$.

- Ejemplo: Además de la multiplicación de matrices de tamaño n x n está el algoritmo Floyd-Warshall es un algoritmo de análisis sobre grafos que permite encontrar el camino mínimo en grafos dirigidos ponderados. El algoritmo encuentra los caminos más cortos entre dos entre todos los pares de nodos.

```
 int cn; //cantidad de nodos

 vector< vector<int> > ady; // [[matriz de adyacencia]]

 // Devuelve una matriz con las distancias mínimas de cada nodo al
resto de los vértices.

 vector< vector<int> > Grafo :: floyd(){

 vector< vector<int> > path = this->ady;

 for(int i = 0; i < cn; i++)

   path[i][i] = 0;

 for(int k = 0; k < cn; k++)

    for(int i = 0; i < cn; i++)

     for(int j = 0; j < cn; j++){
```

```
      int dt = path[i][k] + path[k][j];

      if(path[i][j] > dt)

      path[i][j] = dt;

    }

 return path;

 }
```

Como se puede ver en el código que se puede encontrar en "https://www.ecured.cu/Floyd-Warshall" se encuentran tres "for" anidados. Lo que implica que la complejidad del algoritmo es de $O(n^3)$.

Por lo general los algoritmos realizando operaciones sobre grafos o matrices implican una complejidad $O(n^3)$.