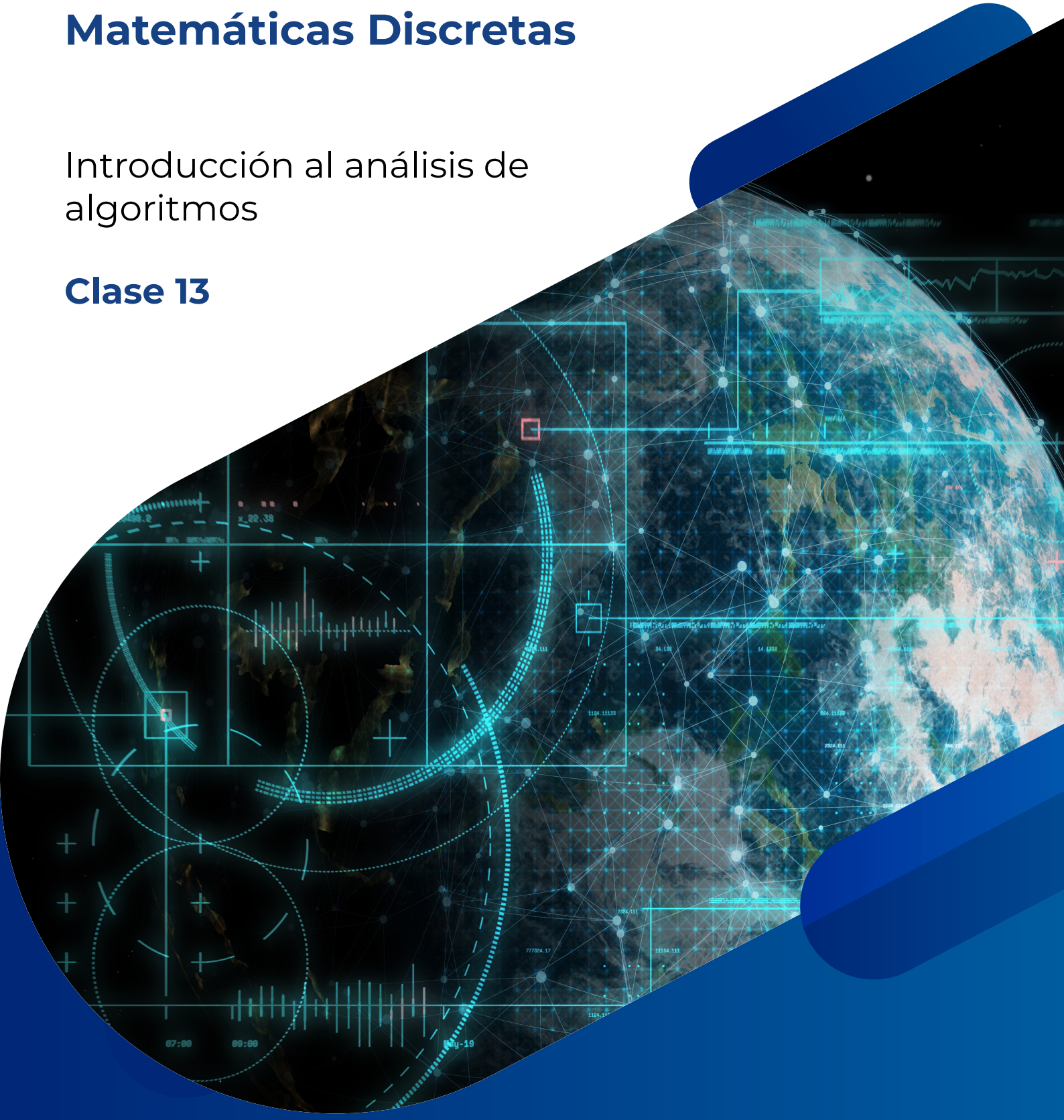
Matemáticas Discretas

Introducción al análisis de algoritmos

Clase 13



La excelencia no se improvisa

Introducción de la clase

¿Alguna vez te has preguntado por qué algunos programas informáticos responden de inmediato, mientras que otros tardan una eternidad en completar una tarea? La diferencia no siempre está en el hardware, sino en la forma en que fueron diseñados los algoritmos que los hacen funcionar. En esta clase, comenzaremos a explorar el análisis de algoritmos: una herramienta para evaluar cuán eficiente es una solución antes incluso de ejecutarla.

Aprenderás a estimar cuánto tiempo toma un algoritmo en función del tamaño de los datos que procesa, y a distinguir entre los escenarios más favorables, los más exigentes y los más comunes. Además, conocerás una forma matemática de describir la eficiencia de un algoritmo usando notaciones como la O-grande. Estos conceptos no solo son esenciales para construir programas eficientes, sino que también forman parte del lenguaje común en áreas como la programación, la ciberseguridad y el desarrollo de software.

Clase 13: Introducción al análisis de algoritmos

RDA 2: Resolver problemas prácticos y teóricos mediante la aplicación de técnicas y herramientas de la Matemática Discreta, como lógica matemática, análisis de conjuntos y teoría de grafos en el manejo de estructuras discretas.

13. Introducción al análisis de algoritmos

Cuando diseñamos un algoritmo, no basta con que funcione correctamente. También debemos preguntarnos: ¿es la mejor opción disponible? En muchos problemas, existen varias formas de llegar a una misma solución, pero no todas son igual de eficientes. Un buen algoritmo no es solo aquel que resuelve el problema, sino aquel que lo hace de la manera más rápida y utilizando la menor cantidad posible de recursos computacionales. Aquí es donde entra en juego el análisis de algoritmos.

El análisis de algoritmos nos permite estimar cuánto tiempo y cuánta memoria requiere un algoritmo antes incluso de ejecutarlo. Esto es esencial en contextos como la ciberseguridad, donde trabajar con grandes volúmenes de datos y buscar soluciones

rápidas puede marcar la diferencia. Incluso si dos algoritmos devuelven exactamente el mismo resultado, uno puede demorar segundos y otro, minutos, dependiendo de su diseño interno.

En términos generales, existen dos aspectos principales que analizamos en un algoritmo: el **tiempo de ejecución** y el **uso de espacio en memoria** (White and Ray 2021).

- **Tiempo de ejecución:** mide cuánto tarda el algoritmo en completarse como función del tamaño de la entrada. Por ejemplo, si sumar dos bits toma c segundos, entonces sumar dos números de n bits tomará $F(n) = n \cdot c$ segundos. En este caso, el crecimiento es lineal. Esta medida es teórica: cuenta cuántos pasos se ejecutan, asumiendo que cada paso toma el mismo tiempo.
- **Espacio requerido:** se refiere a la cantidad de memoria que necesita el algoritmo durante su ejecución. Esta memoria puede dividirse en:
  - *Parte fija:* espacio que no depende del tamaño de la entrada, como variables, constantes o el tamaño del propio programa.
  - *Parte variable:* espacio que sí depende del tamaño de la entrada, como estructuras dinámicas, pilas de recursión o arreglos temporales.

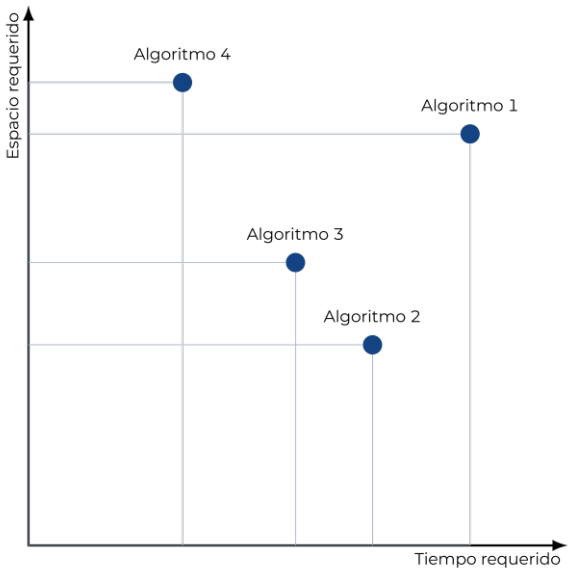
Ejemplo 1. *Supongamos que tienes una lista de 1.000.000 contraseñas y necesitas verificar si una en particular está en esa lista. Un algoritmo de búsqueda lineal comparará cada elemento uno por uno hasta encontrarlo (o no), lo que en el peor de los casos requiere un millón de comparaciones. En cambio, si las contraseñas están ordenadas y usamos búsqueda binaria, podemos encontrar la contraseña correcta (o confirmar su ausencia) en aproximadamente 20 pasos. Ambos algoritmos resuelven el mismo problema, pero de manera muy distinta.*

Ejemplo 2. *En ciberseguridad, cuando se analiza un tráfico de red en busca de patrones sospechosos, puede que un algoritmo simple evalúe cada paquete uno por uno, mientras que un algoritmo más sofisticado use estructuras como árboles de decisión o hash tables. El primero puede saturar el sistema ante ataques masivos; el segundo puede responder rápidamente, incluso bajo presión. Este tipo de decisiones no son triviales: exigen un análisis cuidadoso del comportamiento del algoritmo.*

En la siguiente figura, presentamos cuatro algoritmos comparados gráficamente según el tiempo de ejecución y el espacio requerido. Por ejemplo, el Algoritmo 1 es costoso tanto en tiempo como en memoria, mientras que el Algoritmo 2 parece ser más equilibrado. Este tipo de análisis gráfico es útil para visualizar rápidamente qué algoritmos son más convenientes bajo restricciones específicas: algunos priorizan rapidez, otros, menor uso de memoria, y otros buscan un punto medio.

Figura 1

Comparación de requerimientos de tiempo y espacio en distintos algoritmos



Nota: creación propia (Merino, A., 2025).

Para ayudarte a visualizar mejor estos conceptos y entender cómo se hace un análisis básico de algoritmos, te recomendamos el siguiente video.

- **Título del enlace:** Cómo ANALIZAR tus ALGORITMOS
- **Descripción del enlace:** En este video se explica de forma sencilla y visual cómo se puede analizar el tiempo de ejecución y espacio que usa un algoritmo.

Enlace: https://youtu.be/IZgOEC0NIbw?si=i3gEjMpadDMX9ne8

13.1. Tiempo de ejecución

Cuando medimos el tiempo que tarda un algoritmo en ejecutarse, podríamos pensar en segundos, minutos o días. Sin embargo, esta medida puede variar entre computadoras,

dependiendo del procesador, la cantidad de memoria, el sistema operativo, la carga del sistema, o incluso procesos en segundo plano (White and Ray 2021). Por ello, medir el tiempo real no siempre es útil ni objetivo. En lugar de eso, se mide el número de *operaciones fundamentales* que realiza el algoritmo, que son independientes del hardware.

Las operaciones que se consideran elementales incluyen comparaciones, asignaciones, operaciones aritméticas (suma, multiplicación), llamadas a funciones, accesos a elementos de una estructura de datos, entre otras. Estas operaciones se cuentan como si tomaran el mismo tiempo, permitiéndonos estudiar el comportamiento del algoritmo de forma abstracta y general.

Definición 1.

Dado un algoritmo A, cuyo conjunto de datos es D, se define la función $T: D \to \mathbb{N}$, denominada **tiempo de ejecución** (o costo computacional), de tal forma que para un dato $x \in D$, $T(x)$ es el número de *operaciones elementales* que realiza el algoritmo, con el dato x, hasta finalizar.

En muchos casos, nos interesa cómo cambia el tiempo de ejecución cuando cambia el *tamaño de la entrada*, más allá de los datos específicos. Por ejemplo, el tamaño puede medirse como:

- el número de elementos en una lista,
- la cantidad de vértices en un grafo,
- o el número de bits necesarios para representar un número.

Definición 2.

Sea un algoritmo A cuyo conjunto de datos es D y, para $x \in D$ tomemos $n_x \in \mathbb{N}$ una característica del dato x. Si para todo $x, y \in D$ tal que $n_x = n_y = n \in \mathbb{N}$ se tiene que $T(x) = T(y)$, se dice que el tiempo de ejecución no depende de la naturaleza de los datos sino únicamente del **tamaño de la entrada** n y se redefine la función $T: \mathbb{N} \to \mathbb{N}$ de tal forma que para $n \in \mathbb{N}$, $T(n)$ es el número de *operaciones elementales* que realiza el algoritmo con datos de tamaño n, hasta finalizar.

Consideremos el siguiente ejemplo tomado de Johnsonbaugh (2018):

Ejemplo 3. *Suponga que se tiene un conjunto X de n elementos, algunos etiquetados con «rojo» y otros con «negro», y se desea encontrar el número de subconjuntos que contienen al menos un elemento rojo. Si se construye un algoritmo que examina todos los subconjuntos de X y cuenta los que contienen al menos un rojo, entonces ese algoritmo generará los 2^n subconjuntos posibles y verificará cada uno. El tiempo de ejecución, entonces, será proporcional a 2^n, lo cual crece tan rápidamente que hace inviable su ejecución para valores grandes de n.*

La siguiente tabla ilustra este fenómeno. Si asumimos que cada operación elemental toma 1 microsegundo (10^{-6} segundos), se muestra el tiempo total requerido para diferentes funciones de crecimiento, al variar el tamaño de la entrada n.

Tabla 1

Tiempo para ejecutar un algoritmo si un paso toma un microsegundo de ejecución

Número de pasos	**Tiempo de ejecución si $n =$**					
2-7	3	12	50	100	1000	10^5
1	1×10^{-6} s	1×10^{-6} s	1×10^{-6} s	1×10^{-6} s	1×10^{-6} s	1×10^{-6} s
$\lg(\lg(n))$	1×10^{-6} s	2×10^{-6} s	2×10^{-6} s	3×10^{-6} s	3×10^{-6} s	4×10^{-6} s
$\lg(n)$	2×10^{-6} s	4×10^{-6} s	6×10^{-6} s	7×10^{-6} s	1×10^{-5} s	2×10^{-5} s
n	3×10^{-6} s	1×10^{-5} s	5×10^{-5} s	1×10^{-4} s	1×10^{-3} s	0.1 s
$n\lg(n)$	5×10^{-6} s	4×10^{-5} s	3×10^{-4} s	7×10^{-4} s	1×10^{-2} s	2 s
n^2	9×10^{-6} s	1×10^{-4} s	3×10^{-3} s	0.01 s	1 s	3 hr

Número de pasos	**Tiempo de ejecución si $n =$**					
2-7	3	12	50	100	1000	10^5
n^3	3×10^{-5} s	2×10^{-3} s	0.13 s	1 s	16.7 min	32 años
2^n	8×10^{-6} s	4×10^{-3} s	36 años	4×10^{16} años	3×10^{287} años	3×10^{30089} años

Nota: adaptado de Johnsonbaugh (2018).

Como se observa en la tabla, algoritmos con número de pasos n o $n\log(n)$ siguen siendo eficientes incluso para entradas grandes, mientras que algoritmos con número de pasos como 2^n, se vuelven imprácticos muy rápidamente. Por ejemplo, para $n = 1000$, una función 2^n requeriría más tiempo que la edad del universo.

Veamos algunos ejemplos prácticos del cálculo del tiempo de ejecución de algunos algoritmos.

Ejemplo 4. *Consideremos una función que accede al primer elemento de un arreglo.*

- ***Tamaño de la entrada:*** *Un arreglo A con $n \geq 1$ elementos.*
- ***Análisis de la dependencia:*** *El tiempo de ejecución no cambia, independientemente de los datos que tenga el arreglo. Podemos tomar como tamaño de los datos, el número de elementos del arreglo.*

Código 1

Acceder al primer elemento de un arreglo

```
def primero(A):
    x = A[0]
    return x
```

Cálculo del costo:

- ***Línea 2:*** *La operación de acceso al arreglo ($A[1]$) y la asignación a la variable x son dos operaciones elementales.*

- ***Línea 3:*** *El retorno de la función es una operación.*

El número total de operaciones es $2 + 1 = 3$. *Como este valor es fijo y no depende de* n, *la función de tiempo es* $T(n) = 3$.

Ejemplo 5. *Analicemos una función que copia todos los elementos de un arreglo a otro.*

- ***Tamaño de la entrada:*** *Un arreglo* A *de longitud* n.
- ***Análisis de la dependencia:*** *El algoritmo no depende de la naturaleza de los datos que contenga el arreglo, solo debe recorrer todos los elementos del arreglo una única vez, por lo que el número de operaciones dependerá del número de elementos de la entrada,* n.

Código 2

Copiar un arreglo

```
def copiar(A,n):
    B = []
    for i in range(n):
        B[i] = A[i]
    return B
```

Cálculo del costo:

- ***Línea 2:*** *La creación de un nuevo arreglo y la asignación toman un tiempo constante. Contabilizamos una operación.*
- ***Líneas 3-4 (Bucle):*** *El bucle se repite* n *veces, una por cada elemento del arreglo. En cada iteración:*
  - *Se accede a* $A[i]$ *y* $B[i]$.
  - *Se realiza una asignación (*$B[i] = A[i]$*).*

  En total, cada iteración realiza 2 operaciones. Como el bucle se ejecuta n *veces, las operaciones de este bloque son* $2n$.
- ***Línea 5:*** *El retorno de la función es una operación.*

Sumando todas las operaciones, obtenemos la función de tiempo: $T(n) = 1 + 2n + 1 = 2n + 2$.

Ejemplo 6. *Consideremos un algoritmo que cuenta los elementos positivos en una matriz cuadrada.*

- ***Tamaño de la entrada:*** *Una matriz* M *de* $n \times n$. *El tamaño de la entrada se mide por* n.
- ***Análisis de la dependencia:*** *Para resolver el problema, no depende del tipo de datos que tenga la matriz ni de su distribución, solo debemos visitar cada una de las entradas, por lo que podemos considerar a* n *como el tamaño de los datos.*

Código 3

Contar elementos positicos en una matriz

```
def contar_positivos(M,n):
    c = 0
    for i in range(n):
        for j in range(n):
            if M[i,j] > 0 :
                c = c + 1
            else:
                c = c + 0
    return c
```

Cálculo del costo:

- ***Línea 2:*** *La asignación inicial de* c *es una operación.*
- ***Líneas 3-8 (Bucles anidados):*** *El bucle exterior se ejecuta* n *veces, y el bucle interior se ejecuta* n *veces para cada iteración del bucle exterior. En total, el bloque interior se ejecuta* $n \cdot n = n^2$ *veces. En cada una de estas* n^2 *iteraciones:*
  - *La comparación (*$M[i, j] > 0$*) es una operación. Esto se hace para las* n^2 *celdas, sumando* n^2 *operaciones.*

– *Sin importar el resultado del condicional, la suma y la asignación ($c = c + 1$ o $c = c + 0$) son dos operaciones. Esto se hace para cada una de las n^2 celdas, sumando $2n^2$ operaciones.*

El costo de los bucles anidados es $n^2 + 2n^2 = 3n^2$.

- ***Línea 9:*** *El retorno de la función es una operación.*

La función de tiempo total es:

$$T(n) = 1 + 3n^2 + 1 = 3n^2 + 2.$$

En este último ejercicio, notemos que se puede simplificar el código, eliminado la asignación $\boldsymbol{c = c + 0}$, la cual no es necesaria y es redundante, pero con esto, el número de operaciones que ejecuta el algoritmo dependería de la naturaleza de los datos, por ejemplo, si todas las entradas fueran negativas, el algoritmo dejaría de hacer $\boldsymbol{2n^2}$ operaciones, es decir, realizaría $\boldsymbol{n^2 + 2}$ operaciones en total. Esto lo analizaremos en la siguiente sección.

13.2. Mejores, peores y promedio casos

En el ejemplo anterior, observamos que al eliminar la operación redundante $c = c + 0$, el número total de operaciones que realiza el algoritmo puede variar según la naturaleza de los datos. Si todos los elementos son negativos, se ejecutan menos operaciones que si hay elementos positivos. Esto nos lleva a la idea de que, para un mismo tamaño de entrada n, el tiempo de ejecución de un algoritmo puede no ser único, sino depender de los datos específicos que recibe como entrada.

Para formalizar esta idea, introducimos las siguientes tres nociones: el tiempo de ejecución en el mejor caso, en el peor caso y en el caso promedio.

Definición 3.

Dado un algoritmo cuyo conjunto de datos es D y, para $x \in D$ tomemos $n_x \in \mathbb{N}$ una característica del dato x. Para $n \in \mathbb{N}$, se define:

- **Tiempo del mejor caso:**

$$T_m(n) = \min\{T(x) : n_x = n\};$$

representa la menor cantidad de operaciones que puede ejecutar el algoritmo para entradas de tamaño n.

- **Tiempo del peor caso:**

$$T_M(n) = \max\{T(x): n_x = n\};$$

representa la mayor cantidad de operaciones que puede ejecutar el algoritmo para entradas de tamaño n.

- **Tiempo esperado:**

$$T_e(n) = E(T(x) \mid n_x = n).$$

representa el número promedio de operaciones que ejecuta el algoritmo para entradas de tamaño n, considerando alguna distribución de probabilidad sobre los datos.

Ejemplo 7 (Búsqueda de un dato en una lista). *Supongamos que tenemos una lista L de n elementos y queremos buscar si el valor a se encuentra en ella. Consideremos el algoritmo de búsqueda lineal, que recorre la lista elemento por elemento hasta encontrar a o hasta agotar la lista.*

- ***Mejor caso:*** *el dato buscado a está en la primera posición. En este caso, basta con una comparación, por lo que $T_m(n) = 1$.*
- ***Peor caso:*** *el dato a no está en la lista, o está en la última posición. En este caso, se deben realizar n comparaciones, por lo que $T_M(n) = n$.*
- ***Caso promedio:*** *si suponemos que a está presente en la lista con probabilidad uniforme y su posición es igualmente probable entre las n posibles, en promedio se necesitarán $n + 1/2$ comparaciones. Si a no está en la lista, se requerirán n comparaciones. Considerando ambos escenarios, el valor esperado depende de la probabilidad asumida de que a esté o no en la lista.*

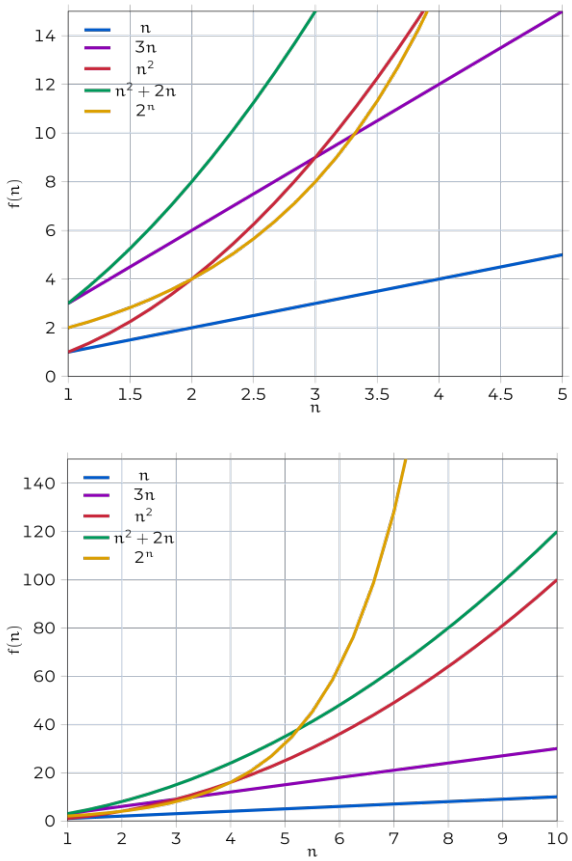
Este ejemplo muestra que el análisis de algoritmos no se limita a una única medida de tiempo, sino que debemos considerar escenarios extremos (mejor y peor caso) y, cuando es posible, un escenario intermedio que refleje el comportamiento típico (caso promedio).

13.3. Notación O-grande y similares

En las secciones anteriores vimos cómo calcular el número de operaciones que realiza un algoritmo en función del tamaño de la entrada. Sin embargo, no todas las funciones crecen de la misma manera. En la siguiente figura se muestra la comparación entre funciones como n, $3n$, n^2, $n^2 + 2n$ y 2^n. Al inicio, $n^2 + 2n$ parece crecer más rápido que 2^n, pero al ampliar la vista se observa que 2^n termina superando ampliamente a todas las demás.

Figura 2

Comparación de crecimiento de funciones



Nota: creación propia (Merino, A., 2025).

Para formalizar estas ideas, introducimos tres notaciones fundamentales que comparan el crecimiento de funciones, que se conoce como notación asintótica.

Definición 4.

Sean f y g funciones reales cuyos dominios son el mismo subconjunto de los reales no negativos. Decimos que:

1. f es de orden al menos g, que se escribe $f(x)$ es $\Omega\big(g(x)\big)$, si y sólo si existen $A > 0$ y $a \geq 0$ tales que, para todo $x > a$

$$A|g(x)| \leq |f(x)|.$$

2. f es de orden a lo más g, que se escribe $f(x)$ es $O\big(g(x)\big)$, si y sólo si existen $B > 0$ y $b \geq 0$ tales que, para todo $x > b$

$$|f(x)| \leq B|g(x)|.$$

3. f es de orden g, que se escribe $f(x)$ es $\Theta\big(g(x)\big)$, si y sólo si existen $A, B > 0$ y $k \geq 0$ tales que, para todo $x > k$

$$A|g(x)| \leq |f(x)| \leq B|g(x)|.$$

En la práctica, la notación que más se utiliza es la **O-grande**, ya que describe una cota superior del tiempo de ejecución en función del tamaño de la entrada. Esto nos permite comparar algoritmos de manera sencilla y anticipar su comportamiento cuando el tamaño de los datos crece.

- Por ejemplo, cualquier polinomio de grado n, como

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0,$$

  es $O(x^n)$, ya que el término de mayor grado domina el crecimiento.

- En los ejemplos anteriores: el acceso al primer elemento de un arreglo fue $O(1)$, la copia de un arreglo fue $O(n)$ y el conteo de elementos positivos en una matriz $n \times n$ fue $O(n^2)$.

Tabla 2

Complejidades comunes en algoritmos

Complejidad	Ejemplo
$O(1)$	Acceso a un elemento de un arreglo
$O(\log n)$	Búsqueda binaria en una lista ordenada
$O(n)$	Búsqueda lineal en una lista no ordenada
$O(n\log n)$	Algoritmos eficientes de ordenamiento (QuickSort promedio)
$O(n^2)$	Ordenamiento por inserción o burbuja
$O(n^3)$	Multiplicación de matrices clásica
$O(n^{2.81})$	Multiplicación de matrices con el algoritmo de Strassen
$O(2^n)$	Problemas de fuerza bruta
$O(n!)$	Resolver el problema del viajante por enumeración de rutas

Nota: creación propia (Merino, A., 2025).

Ejemplo 8 (Multiplicación de matrices). *La multiplicación de matrices es una operación fundamental en áreas como gráficos por computadora o entrenamiento de redes neuronales.*

- *Con el algoritmo clásico, multiplicar dos matrices de tamaño $n \times n$ requiere $O(n^3)$ operaciones.*
- *Con algoritmos más avanzados, como el de Strassen, se logra $O(n^{2.81})$.*
- *Existen algoritmos aún más rápidos, pero complejos, que se utilizan en computación de alto rendimiento.*

Esto ilustra que, incluso para el mismo problema, pueden existir algoritmos con distintas complejidades, lo que tiene un impacto directo en aplicaciones reales.

Para profundizar en este tema, puedes revisar el siguiente recurso:

- **Título del enlace:** Notación Big O
- **Descripción del enlace:** Explicación introductoria sobre cómo funciona la notación Big O y cómo se aplica al análisis de algoritmos.
- **Enlace:** https://youtu.be/MyAiCtuhiqQ?si=0T5K883w7Mj4IThk

Finalmente, cabe señalar que, además del análisis matemático, también es posible estudiar el comportamiento de un algoritmo de forma experimental o simulada. Estos métodos de análisis estarán disponibles en los recursos de profundización.

Referencias

Johnsonbaugh, R. (2018). Discrete Mathematics (8.a ed.). Pearson Educación.

White, R. y Ray, A. (2021). Practical Discrete Mathematics. Packt Publishing.

Términos

- **Tiempo de ejecución:** Es la cantidad de operaciones elementales que realiza un algoritmo desde que inicia hasta que termina.
- **O-grande:** Es una notación asintótica que describe una cota superior para el crecimiento del tiempo de ejecución de un algoritmo.

Profundización

Recurso 1

- **Título del recurso:** Análisis de algoritmos por simulación
- **Descripción del enlace:** Este recurso presenta ejemplos de análisis de algoritmos por simulación para estimar tiempos de ejecución.
- **Enlace:** https://colab.research.google.com/github/andres-merino/MatematicasDiscretas-Virtual-05-N0279/blob/main/pages/clases/clase13/AnalisisAlgoritmos.ipynb

PUCE Virtual

La excelencia no se improvisa

síguenos