3Análisis de algoritmos

Tengo un cierto problema para resolver, y 2 algoritmos que lo resuelven: algoA y algoB. ¿Cuál usaríamos? ¿Cómo saber cuál es **mejor**?

El término “análisis de algoritmos” fue introducido por Donald Knuth. Nos permite caracterizar la cantidad de **recursos computacionales** que usará el mismo cuando se aplique a ciertos datos y evaluar así su **performance**.

Nos permite tener una **métrica** para rankear algoritmos y poder decidir cuál es mejor y cuál es peor en determinado contexto. De ahora en adelante, asumimos que los algoritmos ejecutan en máquinas secuenciales (un core).

Las principales métricas para medir la **complejidad** de algoritmos son:

1) El tiempo de ejecución (time complexity).
2) El espacio que utilizan (space complexity).

El **tiempo de ejecución** puede medirse tanto empíricamente como teóricamente.
Si se hace de forma **empírica**, hay que tener cuidados:
Como los algoritmos tardan tiempo diferente dependiendo de los datos con los que operan (input), para realizar el testeo empírico con datos muy grandes, habría que primero generar esos valores, y luego realizar el chequeo. Estas tareas en conjunto podrían tardar días.

Además, diferencias de **arquitectura**/hardware podrían hacer que un algoritmo en principio corriera más rápido que otro. Sin embargo ¿cómo puede saber realmente cuál tarda menos si las computadoras son diferentes?

Analizarlo de forma **teórica** consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde se ejecute.
La idea básica es contar la cantidad de operaciones primitivas que ejecuta el algoritmo. No importa cuánto tardan. Dichas operaciones son las más costosas de ejecutar en cualquier computadora: comparaciones, operaciones, transferencia de control, etc. Las asignaciones llevan tiempo despreciable así que las ignoramos. Como el tamaño del input afecta la performance del algoritmo, la “fórmula teórica” se realiza contacto la cantidad de operaciones primitivas que se realiza en función del tamaño de entrada.

Pero en realidad, la descripción que buscamos para caracterizar tiempos de ejecución y comparar algoritmos es una asíntota (cota) expresada en términos de N que nos permita identificar la tasa de crecimiento de la fórmula.

Definición de **O grande**:
Sean T(N) y g(N) funciones con N > 0. Se dice que T(n) es O(g(n)) si y solo si existen c > 0 (constante no dependiente de N) y n > 0 tal que para todo N >= n se cumple que
0 <= T(N) <= c * g(N).
Ejemplo: supongamos que nuestro algoritmo tiene T(N)=3*N+1.
Si g(N) = N, quiero que 0 <= 3*N+1 <= c*N. ¿Cuánto debe valer c?
Como N >= 1, 3*N+1 <= 3*N+N = 4*N.
De esta forma, tomando un c >= 4, tenemos que 0 <= 3*N+1 <= 4*N.
Entonces el algoritmo es O(N).

O(1) < O(log2(N)) < O(sqrt(N)) < O(N) < O(N*log2(N)) < O(N^2) < O(N^3) < O(2^N) < O(N!)

Se dice que un programa es O(1) cuando el algoritmo no depende del tamaño de entrada de los datos.

Aclaración: para realizar el cálculo de la complejidad temporal de un algoritmo, no basta con analizar el tamaño de los datos de entrada. La performance del algoritmo también puede depender de **cómo vienen los datos** (por ejemplo, si me dan un vector, puede ser que me convenga que me den el arreglo ya ordenado).
Se puede hacer un análisis del mejor caso, caso promedio, y peor caso. Nosotros trabajaremos con el **peor caso**.

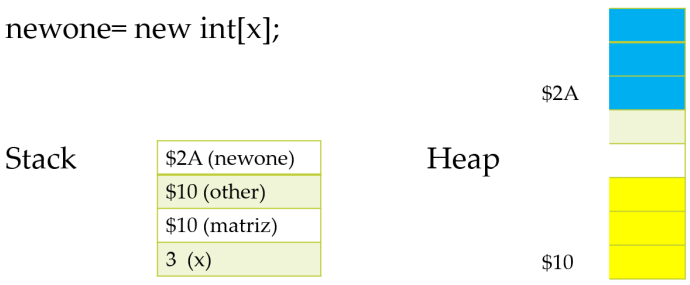
Más ejemplos:
T(N) = 6*N+2 → O(N)
T(N)=2*N^3+100*N^2 → O(N^3)
T(N)=2^N+N^3 → O(2^N)
T(N)=N+6*log10(N) → O(N)

El **espacio de RAM** se mide teóricamente. Para que un algoritmo ejecute algo en el procesador, los datos deben estar en RAM. Puede ser que los datos residan en el disco, pero el procesador va a disco, los carga en RAM y ejecuta.
Consiste en usar una descripción de alto nivel del algoritmo para evaluar cuánto espacio extra necesita para sus variables (parámetros formales, invocaciones a otras funciones, variables locales). Se lo describe con una fórmula en términos del tamaño de entrada del problema (al igual que con el tiempo de ejecución). La idea también es esencialmente la misma: buscar una cota para el espacio en RAM (stack y heap), y se independiza de software y hardware.

No siempre un algoritmo es mejor que otro tanto temporal como espacialmente. Evaluar si un algoritmo es mejor que otro suele ser un tradeoff entre espacio y tiempo.

En Java, cada vez que hacemos new reservamos lugar en el **heap**. El garbage collector es el proceso que libera esa zona cuando detecta que una zona ya no es más referenciada por ninguna variable. Cada vez que se invoca un método se genera un **stack frame** para el mismo, conteniendo: los parámetros formales con sus valores, variables auxiliares declaradas dentro del método y el lugar de la próxima sentencia a ejecutar (así, cuando se retorne, continúa la ejecución).

```
x = 3;
matriz= new int [x];
other = matriz;
newone= new int[x];
```



Java permite configurar al heap con parámetros: la cantidad inicial de heap prealocada y la cantidad máxima posible de alocar: java -Xms512m -Xmx4G -cp HeapOverflow-1.jar space.Generate
También se puede modificar el tamaño del stack: -Xss(tam) donde tam puede ser 10k, 1024k, 2048k, 512m o 1G.

Maven

Si usamos en las aplicaciones bibliotecas propias, crear un “peoyectoJava” sirve. Pero si usamos bibliotecas externas (otros jars), es complicado mantener actualizaciones y versiones de esta manera, dado que importamos jars estáticamente.
Maven es una utilidad para crear y administrar proyectos basados en Java. Permite declarar dependencias para utilizar librerías externas.

En pom.xml se declara el nombre del proyecto, la versión, el paquete en el que se encuentra, la versión de Java con la que compilamos y las dependencias. Maven busca estas primero localmente en nuestro repositorio local. En caso de no encontrarla la descarga del repositorio correspondiente al repositorio local de nuestra computadora.

Test driven development

TDD es una metodología que comienza por los tests y luego pasa a la implementación. Permite enfocarse en la definición de la interfaz y del comportamiento y no en los detalles internos de implementación. Ve al sistema que se va a desarrollar como una caja negra ya que todavía no existe. Apunta a que todas las funcionalidades tengan algún test que las controle y las defina.

Unit testing

Consiste en testear pequeñas unidades de código. Estos se corren automáticamente y pueden ser ejecutados cada vez que se hacen cambios para comprobar que la funcionalidad anterior siga funcionando correctamente. Son esenciales para las metodologías ágiles y para refactorizar código.

¿Qué testear en un unit test?
En el caso de una clase: secuencias de llamadas válidas, secuencias de llamadas inválidas, chequeo de invariantes.
En el caso de un método: casos típicos, casos de borde, casos de error, casos de excepción.

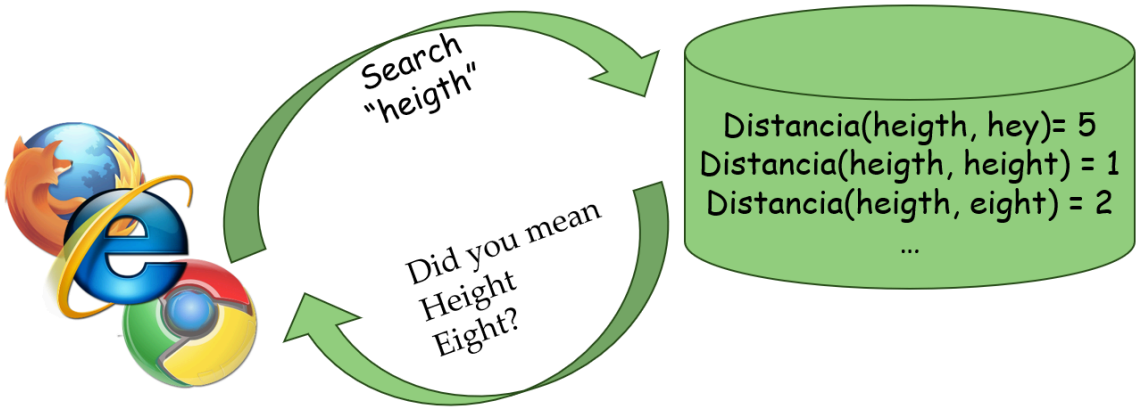
JUnit es un framework para realizar casos de prueba en aplicaciones Java. Se pueden comparar resultados de las invocaciones de métodos con los valores esperados, o verificar si una excepción fue lanzada o no.
Una clase es **testable** si fue diseñada para poder realizar correctamente tests sobre ella. Los métodos no deben tener efectos secundarios ni dependencias con componentes externos.

Algoritmos para textos

Borde: todo substring que es prefijo y sufijo a la vez.
Ejemplo: s = “01230”
Prefijos: “”, “0”, “01”, “012”, “0123”, s
Sufijos: “”, “0”, “30”, “230”, “1230”, s
Bordes: “”, “0”, s
Todo string tiene como mínimo dos bordes: “” y s

Los buscadores usan alguna estrategia en el caso de que la búsqueda lanzada no sea reconocida en el “corpus” que posee sobre búsquedas y documentos indizados.
Las estrategias pueden ser muy variadas (combinaciones de una o más de estas):

1) Buscar las palabras, y si las palabras no están en el corpus de los documentos indizados, encontrar las que mayor similitud posean y sugerirlas.



2) Tomar el idioma que tiene configurado el browser para saber en qué corpus buscar las palabras usadas. Hay reglas conocidas para el idioma en cuestión (por ejemplo, heigth vs height, widht vs width, etc).
3) Sabiendo que los usuarios buscan palabras y cuando fue un error de typo/ortografía no clickear nada del resultado e inmediatamente intentan realizar la búsqueda arreglada, almacenan esas búsquedas erróneas con la que sí arrojó resultados correctos. Es decir, hay un matching entre errores viejos y soluciones que los mismos usuarios hicieron.

Mínimas reglas que deberían aplicarse para búsqueda de texto:

1) Sacar blancos del comienzo y del final (trim), y blancos internos extra (ejemplo “ yogurt bebible “ → “yogurt bebible”).
2) Pasar todo a mayúscula o minúscula.
3) Si se conocen abreviaturas, usarlas.
4) Eliminar símbolos de puntuación.
5) Si se conocen sinónimos, utilizarlos.

Soundex

Soundex es un algoritmo **fonético**. Codifica una palabra según como suena. Intenta solucionar problemas de pronunciación, pero fue creado para el alfabeto inglés. Aquellas palabras que suenen igual, aunque no se escriban igual, deben ser codificadas de la misma manera.

26 Letras	Pesos fonéticos
A, E, I, O, U, Y, W, H	0 -- no se codifica
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

Siempre devuelve un código OUT de 4 caracteres formados por una letra y 3 dígitos (pesos fonéticos). Si hace falta para completar el código, se rellena con ceros.

Paso 1 (opcional): Pasar a mayúsculas y dejar sólo las letras (dígitos, símbolos de puntuación, espacios, etc. se eliminan).
Paso 2: Colocar OUT[0]=IN[0].
Paso 3: Se calcula vble. **last** como el peso fonético de IN[0]
Paso 4: Para cada letra **iter** siguiente en IN y hasta completar 3 dígitos o terminar de procesar IN, hacer
3.1) calcular vble **current** con peso fonético de **iter**. Si es diferente a 0 y no coincide con **last**, appendear **current en OUT**.
3.2) **independiente del paso anterior,** tapar **last** = **current**.
Paso 5: si hace falta completar con '0's y devolver OUT.

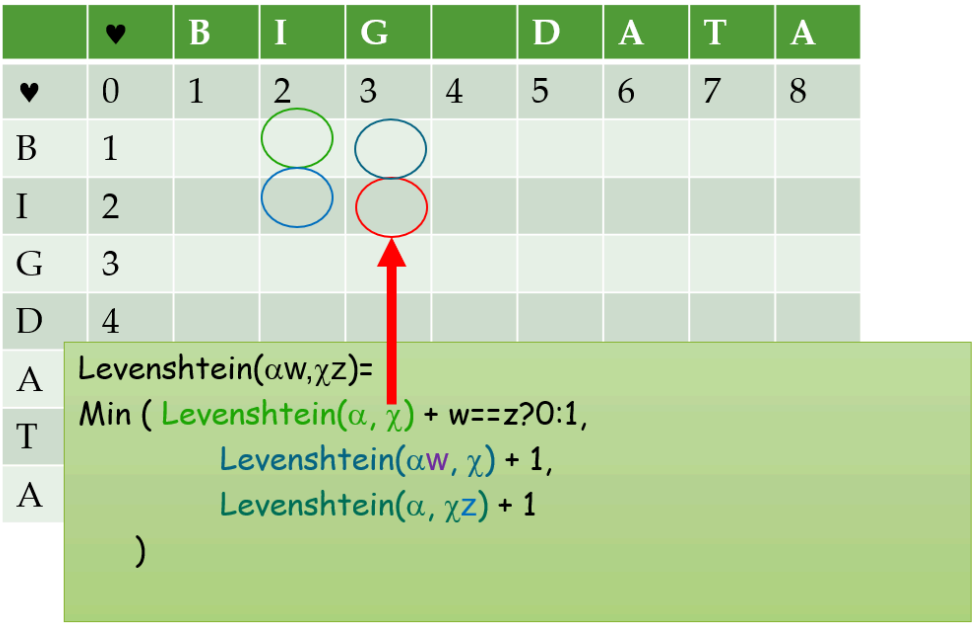
Ejemplo:
Soundex("LuXUry") = "L260"
Soundex(""SZLLOYDTIRUL")="S436"

Soundex **no es una métrica**. Hay que definir una forma de obtener una métrica a partir de Soundex.
En Soundex, la definición de similitud será la proporción de caracteres coincidentes entre los encodings respecto de la longitud del encoding (es decir, los únicos valores posibles de similitud son 0, 0.25, 0.5, 0.75 y 1).

Levenshtein distance

Es un algoritmo que calcula la **mínima cantidad de operaciones** necesarias para transformar un string en otro, donde las operaciones válidas son: insertar, borrar, o sustituir un caracter por otro. Lógicamente, strings iguales deben tener distancia 0 ya que no hace falta transformar uno en otro.
A diferencia de Soundex que se adaptó para proponer una medida de similitud a partir de un cálculo, Levenshtein es una métrica de distancia. Por lo tanto, cumple con las propiedades de simetría (Levenshtein(str1, str2) = Levenshtein(str2, str1)) y desigualdad (Levensthein(str1, str2) <= Levenshtein(str1, str3) + Levenshtein(str3, str2)).
Se implementa con la técnica de **programación dinámica**. Esta técnica se basa en reutilizar valores calculados previamente para no tener que recalcularlos. Los valores calculados deben almacenarse en una estructura de datos (vector, matriz, hashmap) con el objetivo de buscarlos (lookup) y no calcularlos nuevamente cuando se los precise. La idea

es que la complejidad de este lookup sea menor que la complejidad de calcular ese valor, sino la técnica no sería útil.



Esta distancia se puede normalizar para que el valor esté entre 0 y 1. El valor 1 significa coincidencia.
LevenshteinNormalized(str1, str2) = 1 - (Levenshtein(str1, str2) / max(str1.len, str2.len))
Existen variantes de Levenshtein. Por ejemplo, en Damearu-Levenshtein también se agrega la operación de transposición. También puede ser que cada operación tenga un costo distinto.

Q-Grams

Es un algoritmo que consiste en generar los pedazos que componen un string. La distancia entre 2 strings estará dada por la cantidad de componentes que tengan en común.
Si Q es 1, se generan componentes de longitud 1, si Q es 2 se generan bi-gramas, si Q es 3 se generan tri-gramas.
Por ejemplo, para el string “JOHN” si se quiere generar hasta tri-gramas (Q <= 3), puede completarse al comienzo y al final con Q-1 “símbolos especiales” que no pertenezcan al alfabeto. De esta forma, deslizando la ventana imaginaria de tamaño Q, se van generando los Q-gramas. Nuestro string queda “JOHN”, “#JOHN#” y “##JOHN##”, y los Q-grams son {“J”, “O”, “H”, “N”, “#J”, “JO”, “OH”, “HN”, “N#”, “##J”, “#JO”, “JOH”, ”OHN”, “HN#”, “N##”}.

Ejemplo: ¿Qué tan distinto es JOHN de JOE?
Q-Grams de JOE: {“J”, “O”, “E”, “#J”, “JO”, “OE”, “E#”, “##J”, “#JO”, “JOE”, “OE#”, “E##”}.
Q-Grams de JOHN: {“J”, “O”, “H”, “N”, “#J”, “JO”, “OH”, “HN”, “N#”, “##J”, “#JO”, “JOH”, ”OHN”, “HN#”, “N##”}
Los que tienen en común son “J”, “O”, “#J”, “JO”, “##J”, “#JO”.
Distancia(“JOHN”, “JOE”)=6.

La fórmula para normalizarlo a un número en [0, 1] es la siguiente:

Q-Gram (str1, str2) =

$$\frac{\#TG(str1) + \#TG(str2) - \#TGNoShared(str1, str2)}{\#TG(str1) + \#TG(str2)}$$

Donde:
#TG(str1) es la cantidad de trigramas que se generaron de str1.
#TG(str2) es la cantidad de trigramas que se generaron de str2.

#TGNotShared(str1, str2) son la cantidad que no matchearon.

Ejemplo, Q-Gram('JOHN', 'JOE') para Q=3, sabiendo que

Q-grams (John) ={'##J', '#JO', 'JOH', 'OHN', 'HN#', 'N##'}
Q-grams (Joe) = {'##J', '#JO', 'JOE', 'OE#', 'E##' }

Y los Q-gramas que tienen en común son: '##J', '#JO'

Q-Gram(John, Joe) = (6 + 5 - 7) / (6 + 5) = 0.3636

Q-grams(salesal)= { #s, sa, al, le, es, sa, al, l#)
Q-grams(vale)= { #v, va, al, le, e#}.
En común 2 (ojo con los repetidos!).
Q-Gram(salesal, vale) = (8 + 5 - 9) / (8 + 5) = 0.3076

KMP

El algoritmo naive de string matching no aprovecha lo que aprendió durante el recorrido cuando encuentra un mismatch. Hace backtracking en el query y en el target. El algoritmo Knuth-Morris-Pratt no vuelve a chequear un caracter que ya sabe que matcheó. No hace backtracking en el target.

KPM escanea el target de izquierda a derecha, pero aprovecha sus conocimientos sobre los caracteres comparados antes de determinar la próxima posición del patrón a usar. Preprocesa el query antes de la búsqueda una vez con el objetivo de analizar la estructura (las características del patrón query). Para ello construye una tabla Next de query. Esta tabla Next tiene en cada posición i la longitud del borde propio más grande para el substring de query que va de 0 hasta i.

query	A	B	R	A	C	A	D	A	B	R	A
Next	0	0	0	1	0	1	0	1	2	3	4

query	S	A	S	S
Next	0	0	1	1

Estructuras lineales

Cuando se busca una/múltiples apariciones de un elemento en un conjunto se puede proceder de diferentes formas:

1) Se deja la colección como está y nos la ingeniamos para navegar en ella. Este es por ejemplo el caso del algoritmo KMP. En ese caso, el texto es una colección de caracteres y no se modifica para que no se pierda su semántica.
2) Se genera una estructura auxiliar, llamada **índice**, que facilita la búsqueda. Siendo el índice la estructura que se utiliza para encontrar un elemento, la búsqueda sobre el mismo debe ser muy **eficiente**. Un índice está compuesto por elementos que representan la información que indizan.

Características de índices: la clave de búsqueda puede o no tener repetidos. Si se repite, podemos tener la información asociada compactada o no. Además, la clave de búsqueda debe permitir buscar rápidamente la información adicional.
Pero además de requerir poder realizar búsquedas sobre el índice, necesitamos poder insertar y borrar elementos.

	Búsqueda	Inserción	Borrado
Arreglo (cualquiera, desordenado)	✗ O(n)	✓ O(1)	✗ O(n)
Arreglo ordenado por clave de búsqueda	✓ $O(\log_2 n)$	✗ O(n)	✗ O(n)
Hashing	✓ ??	✓ ??	✓ ??

El problema del arreglo es que tiene que garantizar la contigüidad de sus elementos.

Arreglos ordenados

La búsqueda puntual es O(log2(N)) realizando búsqueda binaria.

Teorema maestro

T(N) = a * T(N/b) + c * N^d
N es el tamaño de entrada del problema, a es el número de invocaciones recursivas que realiza ese paso, b mide en qué tasa se reduce el input, c cuenta la cantidad de operaciones que se realizan dentro de la función y d es la complejidad interna del algoritmo.

Entonces la complejidad O grande está dada por los siguientes 3 casos (c no cuenta):

- Si $a < b^d$ entonces el algoritmo es $O(N^d)$
- Si $a = b^d$ entonces el algoritmo es $O(N^d * \log N)$
- Si $a > b^d$ entonces el algoritmo es $O(N^{\log_b a})$

El Teorema Maestro puede utilizarse en problemas que sean resueltos con la técnica **Divide and Conquer**. Esta consiste en dividir el problema en **subproblemas** de un mismo tamaño, resolver cada subproblema en forma independiente por recursión, y combinar los resultados parciales para dar la solución final.

Ejemplo:

```
public static int surprise(int N) {
    if (N < 4)
        return 16;

    for (int i = 0; i < N; i++) {
        System.out.println(i);
    }

    int auxi1= surprise( N / 3);
    int auxi2= surprise( N / 3);

    return auxi1 + auxi2;

}
```

Times(N) = 1 si N < 4 y 2 * T(N / 3) + O(N) si N >= 4
Es decir, tenemos a=2, b=3, y d=1.
Por lo tanto, como a < b^d, el algoritmo es O(N^d)=O(N).

Quicksort

Este método opera in-place y aplica la técnica Divide and Conquer. Particiona en subarreglos y en cada uno elige un pivot y ordena para que todos los elementos a la izquierda sean menores que él y los de la derecha sean mayores que él (es decir, posiciona al pivot en su posición correcta). Si un subarreglo tiene 0 ó 1 elementos, está ya ordenado y se corta la recursión.
El peor caso de quicksort es que esté todo ordenado (O(N^2)). No se puede usar teorema maestro pues las dos llamadas recursivas no parten en tamaños iguales.
En el mejor de los casos, quicksort parte cada mitad en partes iguales en cada iteración. En este caso el algoritmo es O(Nlog2(N)).

Generics

Java es un lenguaje estáticamente tipado, es decir, hay que declarar el tipo de una variable antes de usarla. Sin Generics, los casteos son una fuente de errores que se detectan en

tiempo de ejecución. Con la introducción de Generics, pueden parametrizarse tipos y así minimizar errores. Generics fue implementado usando la **técnica de erasure**. Esta consiste en reemplazar todo tipo de parámetro con su bound/restricción y si no lo hay lo reemplaza por Object. De ser necesario realiza casteos.

Stack

Es una colección de datos ordenada por orden de llegada. La única forma de acceso es mediante un elemento distinguido llamado tope, que es el último elemento que llegó.
Si se implementa con un arreglo, la continuidad de datos está garantizada. Nunca quedan huecos internos y no hace falta mover objetos para garantizar contigüidad ya que las operaciones solo se acceden mediante el tope.
Sin embargo, si se acaba el espacio debemos buscar espacio contiguo en otro lugar y copiar componentes. Es decir, hacemos crecer o decrecer la estructura de a chunks.
Si se lo implementa con una lista lineal simplemente encadenada, es solo cuestión de apuntar el tope al elemento correspondiente (es decir, al primer elemento de la lista). Así, jamás tenemos que recorrer para hacer push o pop.

Evaluador de expresiones

Una expresión es una combinación de operadores y operandos. Pero para simplificar, solo consideramos operadores binarios. Las expresiones pueden clasificarse según la notación que utilizan:

1) Prefija: el operador se encuentra **antes** de los operandos sobre los que aplica,
2) Infija: el operador se encuentra **entre** los operadores sobre los que aplica (es la que usamos normalmente).
3) Postfija: el operador se encuentra **después** de los operadores sobre los que aplica.

Prefija	Infija	Postfija	Prefija Inversa	Infija Inversa	Postfija Inversa
* A B	A * B	A B *	* B A	B * A	B A *

El problema con la evaluación de una expresión infija es que existen ambigüedades cuando dos operadores tienen la misma precedencia. Esto se resuelve mediante la asociatividad, pero se complica más aún cuando aparecen paréntesis que alteran las prioridades.
Esta es la ventaja de las notaciones prefija y postfija. El uso de paréntesis es innecesario debido a que el orden de los operadores determina el orden real de las operaciones en la evaluación de la expresión.

Algoritmo para evaluar una expression que ya esté en notación postfija:

- Cada operador en una expresión postfija se refiere a los operandos previos en la misma.
- Cuando aparece un operando hay que postergarlo porque no se puede hacer nada con él hasta que no llegue el operador, y como la notación es postfija el operador va a llegar después. Por lo tanto cada vez que se encuentre un operando la acción a tomar es "**pushearlo" en una pila**
- Cuando aparezca un operador en la expresión implica que llegó el momento de aplicárselo a los operandos que lo preceden, por lo tanto se deben "**popear" los dos elementos más recientes de la pila** , aplicarles el operador y volver a dejar el resultado en la pila porque dicho valor puede ser operando para otra subexpresión (al resultado habrá que aplicársele el próximo operador que aparezca).
- Cuando se termine de analizar al expresión de entrada el resultado de su evaluación es el único valor que quedó en la **pila**.

Ejemplo: evaluemos 3 10 + 2 - 5 4 * -
Pusheo 3
Pusheo 10
(Acá me topé con el +)
Popeo 10
Popeo 3
Pusheo 3+10=13
Pusheo 2
(Acá me tope con el -)
Popeo 2
Popeo 13
Pusheo 13-2=11
Pusheo 5
Pusheo 4
(Acá me topé con el *)
Popeo 4
Popeo 5
Pusheo 5*4=20
(Acá me tope con el último -)
Popeo 20
Popeo 11
Pusheo 11-20=-9
El resultado de evaluar la expresión es entonces -9.

Parseo de entrada

Probar el siguiente código que toma una línea de la **entrada estándard** que termina con \ n. A ese primer scanner lo llamamos inputScanner.
Luego usa otro scanner sobre la línea previamente leida y la tokeniza separando en espacios (blancos, tabulador, etc). Ese segundo scanner lo llamamos lineScanner

```
public static void main(String[] args) {
     // primer scanner: separador enter
    Scanner inputScanner = new Scanner(System.in).useDelimiter("\\n");
    System.out.print("Introduzca la expresión en notación postfija: ");

    String line = inputScanner.next; // si usan nextLine() no poner \\r

    // segundo scanner: separador espacios sobre el anterior
    Scanner lineScanner = new Scanner(line).useDelimiter("\\s+");
    while(lineScanner.hasNext())
        System.out.println(lineScanner.next());
}
```

Podemos chequear cada token con una expresión regular. En este caso, la enumeración de opciones válidas se separará con un pipe.

```
String token = lineScanner.next();
System.out.println(token);
if (token.matches("¡!|,|;|##") )
    System.out.println("OK:“: token );
else
    System.out.println("invalid: " + token);
```

Parser de presedencia de operadores

Algoritmo para transformar expresión en notación infija en expresión en notación postfija: la idea es que cada vez que aparezcan varios operadores se consulte una tabla que indique cuál se evalúa primero. Si dos operadores tienen la misma precedencia, se utiliza la regla de asociatividad para saber cuál se evalúa primero.
Ejemplo: si tenemos A + B * C / D, entre el + y el *, debe aparecer primero el * en la postfija. Y entre el * y el /, debe aparecer primero el *, pues aunque tengan igual precedencia, la asociatividad a izquierda implica que se evaluará primero el *.
Completemos la siguiente tabla (considerando que las operaciones son asociativas a izquierda) en la que cada celda indica si la precedencia del tope de la pila es mayor que la precedencia del operador actual.

Elemento que está siendo analizado (actual)

Elemento que está en el tope de la pila (previo)	+	-	*	/
+	true	true	False	false
-	true	true	False	false
*	true	true	True	True
/	true	true	True	true

Algoritmo infija a postfija:

1) Cada operando de la expresión infija se copia tal cual en la expresión postfija.
2) Cuando aparece un operador, hay que analizar precedencia respecto de los operadores previos que están en el stack. Esto se reduce a chequear el orden de precedencia entre el tope de la pila y el operador actual.
3) Si la pila está vacía, no se puede comparar, por lo que solo se pushea el operador actual.
4) Si no está vacía y el tope de la pila tiene mayor precedencia que el actual, se realiza pop del operador en la pila y se lo copia en la postfija hasta que la pila quede vacía o bien el operador del tope tenga menor precedencia que el actual. Una vez hecho esto, se pushea el operador actual.
   Si el tope de la pila tiene menor precedencia, solo se pushea el operador actual.
   Por último, si la precedencia es la misma, se va popeando igual hasta que sea estrictamente menor, debido a que por la asociatividad izquierda queremos que operaciones de igual precedencia que aparezcan antes, se evalúen antes (esto igual queda contemplado en la tabla de arriba).
5) Cuando se terminó de analizar la expresión infija, se popean todos los operadores de la pila y se copian en la postfija.

El pasaje de infija a postfija no detecta errores. Estos se detectan cuando se intenta evaluar la expresión postfija.

Ahora, si queremos introducir el operador ^, hay que tener cuidado pues este es asociativo a derecha (a^b^c = a^(b^c)).
La tabla ahora queda asi:

Elemento que está siendo analizado (actual)

Elemento que está en el tope de la pila (previo)	+	-	*	/	^
+	true	true	false	false	False
-	true	true	false	false	false
*	true	true	true	true	False
/	true	true	true	true	false
^	True	True	True	true	false

Solo tenemos que cambiar la tabla de precedencias, agregar ^ como operador válido e implementar el método eval correspondiente a la exponenciación.
Ahora, incorporamos paréntesis en las expresiones. Estos no deben aparecer en la salida pues no son necesarios en la postfija.
Una posible solución es la siguiente: los consideramos operadores especiales y si el operador actual es un “(“, el mismo debe postergarse hasta que aparezca un “)”. Es decir, completar la tabla para que se lo pushee siempre. Si el operador actual es un “)” el mismo debe sacar a todos los operadores de la pila y concatenarlos en la postfija hasta encontrar el “(“ que aparee con él. Sin embargo, este último no se concatena. Hay que colocar que la precedencia entre “(“ y “)” sea false para manejarla como un caso especial, sino se vacía la pila.

Queue

Es una colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de dos elementos distinguidos: first es el más antiguo de los elementos de la colección y tiene prioridad para salir, y last marca el elemento más reciente que ha llegado a la colección.
Si se tiene una unbounded queue (no hay límite en la cantidad de elementos que puede manejar), la implementación de queue con LinkedList es superior a la de ArrayList.

Índices y arreglos ordenados

Ventajas: el arreglo ordenado es una buena estrategia para implementar un índice pues la información puede buscarse rápido ya que se puede usar búsqueda binaria y cada acceso es O(1), por lo que es óptimo para operaciones que requieran acceso a una componente en particular (ejemplo, getMin, getMax).
Desventajas: los datos tienen que estar contiguos. Para garantizarlo, la inserción y el borrado requieren mover componentes. Además, cuando se acaba el espacio pre-alocado, generas más espacio implica generar otro espacio contiguo (de a chunks).

Tratando de superar el problema de la contigüidad y la realocación del espacio, podemos pensar en estructuras de datos que permitan que los elementos estén físicamente aislados y lógicamente conectados. Para ellos utilizamos una **Lista lineal simplemente encadenada**.
Esta está compuesta por 0 o más nodos, y cada nodo contiene su información correspondiente y la referencia al elemento siguiente.

Linked list con header

Es una estructura de datos compuesta por un elemento distinguido llamado header que tiene la referencia del primer elemento de la lista y además información global de la lista, y a su vez, cada nodo almacena su información y la referencia al nodo siguiente (como una linked list normal).