Análisis de Algoritmos

El ***análisis de algoritmos*** es un término introducido por Donald Knuth, que 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 así poder decidir cuál es mejor y cuál es peor.

Las principales métricas para medir la complejidad de algoritmos que ejecutan en máquinas secuenciales (un core) son:

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

1. Tiempo de Ejecución

1.A. Tiempo de Ejecución Empírica

La idea de usar la métrica *"tiempo de ejecución calculada empíricamente"* para rankear algoritmos tiene varias dificultades:

- Como los algoritmos tardan diferente dependiendo de los datos con los que operan (input), para probar realmente con datos grandes, habría que generar esos valores.
- Si mi *algoritmoA* lo ejecuto en mi pc y tarda x ms, y otro propone un *algoritmoB* que ejecuta en su pc y tarda x/2 ms, ¿Cómo saber cuál realmente tarda menos si las computadoras son diferentes?.

1.B. Tiempo de Ejecución Teórica

El ***tiempo de ejecución teórica*** consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde ejecute. Se lo describe con una fórmula matemática.

Idea básica

"Contar la cantidad de operaciones primitivas" que ejecuta el algoritmo. No importa cuánto tarda, sino que se establece en "*unidades de tiempo genéricas"*.

Las operaciones más costosas de ejecutar son:

- *Comparaciones*.
- *Operaciones* aritméticas.
- *Transferencia de control desde una función a otra*.

Las *asignaciones* llevan tiempo despreciable, se pueden ignorar.

Como el tamaño del input afecta la performance del algoritmo, entonces la "fórmula" se realiza contando la cantidad de operaciones primitivas que se realiza expresada en términos del tamaño de dicha entrada.

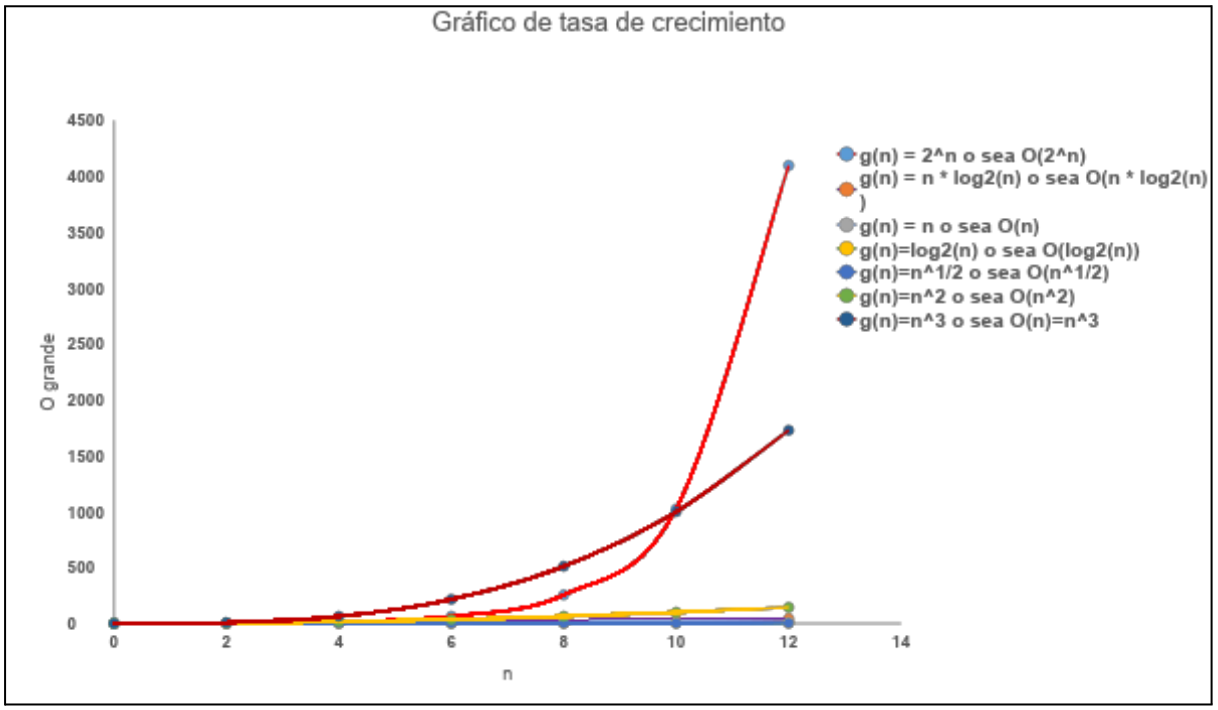
Comportamiento Asintótico, Cota Superior u O Grande

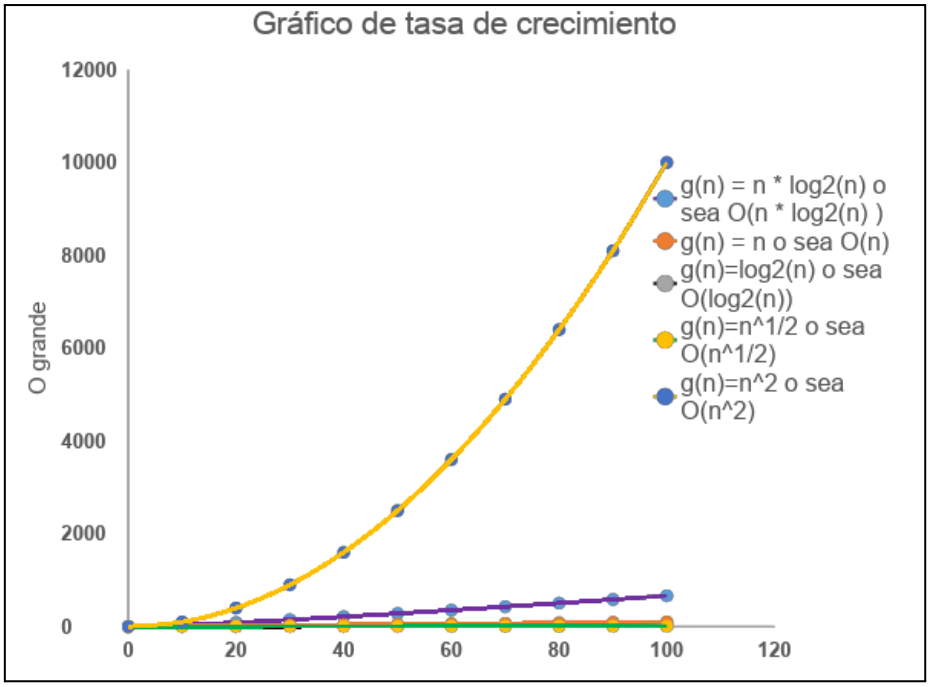
La descripción que buscamos para comparar algoritmos es una *"asíntota"* (cota) expresada en términos de n que nos permita caracterizar la tasa de crecimiento u orden de crecimiento de la fórmula.

Comportamiento Asintótico Superior - O grande *(Asymptotic upper bound running time - O-notation):*

Sean $T(n)$ y $g(n)$ funciones con $n > 0$.

Se dice que $T(n)$ es $O(g(n))$ sii $\exists\ c > 0$ (constante no dependiente de n) y $\exists n_0 > 0$ tal que $\forall\ n \geq n_0$ se cumple que $0 \leq T(n) \leq c\ *\ g(n)$.





Es decir, para $n \to \infty$ tenemos que

$$\ldots < O(log_2(n)) < O(\sqrt{n}) < O(n) < O(nlog_2(n)) < O(n^2) < O(n^3) < O(2^n) < O(n!) <\ldots$$

La ***mejor complejidad*** de todas es $O(1)$, es decir constante.

Importante

La performance del algoritmo también puede depender de cómo vienen los datos.

Se puede hacer un análisis del ***mejor caso***, ***caso promedio*** o ***peor caso*** de input.

2. Espacio de RAM

Para que un algoritmo ejecute algo en el procesador, los datos deben estar en RAM. O sea, puede ser que los datos se almacenen en el disco, pero el procesador no va directo al disco. Va a disco, lo carga en RAM y lo ejecuta.

Consiste en usar una descripción de alto nivel del algoritmo para evaluar cuánto ***espacio extra precisa*** para sus variables (parámetros formales, invocaciones a otra funciones, variables locales). Se lo describe con una fórmula en términos del tamaño de entrada del problema.

La idea es la misma, buscar una corta (***O grande***) para el espacio RAM (***stack y heap***). Busca independizarse de software y hardware, es decir no se tiene en cuenta si una computadora es de 32 o 64 bits, etc. Se expresa a través de ***"n"***.

No siempre un algoritmo es mejor que otro tanto en la parte temporal como en la espacial. Muchas veces sucede que es mejor en lo temporal y peor en lo espacial o viceversa. Es decir, evaluar si un algoritmo es mejor que otro puede ser un tradeoff entre espacio vs tiempo.

Para caracterizar el espacio en nuestro algoritmos escritos en Java, tenemos que tener en cuenta el espacio que se aloca para:

Heap

Cada vez que hacemos *"new"* reservamos lugar en esta zona. El ***Garbage Collector*** es el proceso que libera esa zona cuando detecta que una zona ya no es más referenciada por ninguna variable.

Configuración del Heap

Java permite configurar el heap con parámetros: La cantidad inicial de heap prealocada y la cantidad máxima posible de alocar.

```
$ java   -Xms512m -Xmx1G  -jar  HeapOverflow.jar
```

Stack

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 mismo y el lugar de la próxima sentencia que falta ejecutar (así, cuando se retorne, continúa la ejecución).

Es decir, no resulta gratis invocar funciones, se generan *stack frames*.

Configuración del Stack

```
$ java   -Xss1024K  -jar  StackOverflow.jar
```

En ***intelliJ*** estos parámetros se cambian en "Run/ Edit configuration/ VM config"

Maven

Maven es una utilidad para crear y administrar proyectos basados en Java.

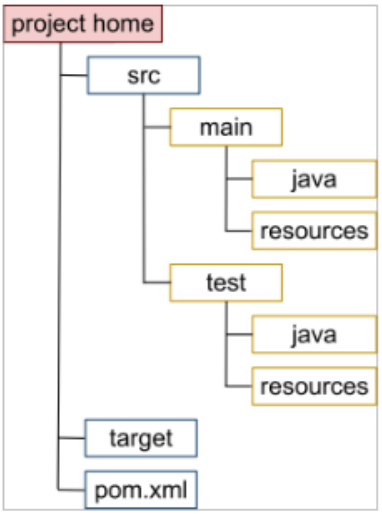
Como objetivos se propone:

- Proporcionar un sistema de construcción uniforme.
- Proporcionar información de calidad del proyecto.
- Proporcionar pautas para el desarrollo de mejores prácticas.
- Permitir la migración transparente a nuevas funcionalidades.

Permite declarar dependencias para utilizar librerías externas.

http://maven.apache.org

Estructura del Proyecto



pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ar.edu.itba.eda</groupId>
  <artifactId>Timer</artifactId>
  <version>1.0</version>
</project>
 <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```

```
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

Dependencias

En el *pom.xml* se declaran las dependencias a utilizar.

Maven busca primero localmente a la dependencia, y en caso de no encontrarla, la descarga del repositorio correspondiente.

Build Phases

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

JUnit

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.

Un caso de prueba es abortado ni bien falla alguna verificación o se lanza una excepción no esperada.

https://junit.org/junit5/

Comparando resultados

El **test unitario** más simple consiste en comparar el resultado obtenido con el resultado esperado.

Para ello, se pueden utilizar los siguiente métodos estáticos:

```
Assertions.assertEquals(valorEsperado, valorObtenido);
Assertions.assertArrayEquals(valorEsperado, valorObtenido);
Assertions.assertTrue(valorObtenido);
```

Si un método lanza una excepción, el mismo se considera que falló. Para aquellos casos en que se espera que se lance esta excepción, se indica de la siguiente manera:

```
Assertions.assertThrows(RuntimeException.class, () -> ...);
```

El inicio de un test unitario se indica de la siguiente manera

```
@Test
void Test() {
}
```

Ambiente de Prueba

Frecuentemente se desea tener un ambiente de prueba prefijado, por ejemplo ciertas variables inicializadas.

Para evitar repetir este código de inicialización en cada uno de los test unitarios se cuenta con varios anotaciones útiles:

```
@BeforeAll: Se ejecutará antes de todos los casos de prueba de la clase.
@BeforeEach: Se ejecutará antes de cada @Test.
@AfterEach: Se ejecutará después de todos los casos de prueba de la clase.
@AfterEach: Se ejecutará después de cada @Test.
```

Coverage

El Code ***Coverage*** es una métrica utilizada en el desarrollo de software que determina el número de líneas de código que fueron ejecutadas durante nuestros tests unitarios. Utilizándolo podemos determinar si nuestras aplicaciones se prueban de forma correcta y qué porcentaje de escenarios no cuentan con pruebas automatizadas.

Algoritmos para Textos

Alfabeto Σ: Conjunto de símbolos o caracteres.

Dado un alfabeto Σ y $k_{\geq 0} \in \aleph$, un ***string*** ***S*** es un elemento $\in \Sigma^k$.

Para $S \in \Sigma^k$, se dice que $|S|$ es k, y denota su ***longitud***. Si $k = 0$, S se dice que es el string vacío y se lo denota con λ.

Dado un alfabeto Σ, se define el conjunto de todos los strings posibles sobre cierto alfabeto, $\Sigma^* = \cup \Sigma^k$, con $k_{\geq 0} \in \aleph$.

Dado un alfabeto Σ y $u \in \Sigma^*$, $w \in \Sigma^*$. Se llama ***concatenación***, al string definido como uw (un elemento a continuación del otro, sin símbolos extra entre ellos).

Dado un alfabeto Σ y los strings $x \in \Sigma^*$, $w \in \Sigma^*$, $z \in \Sigma^*$. Sea $p = xwz$. Se dice que x es un ***prefijo*** de p. Se dice que w es un ***substring*** de p. Se dice que z es un ***sufijo*** de p. Si $x = z$, entonces x es ***borde*** de p.

Data Quality - Matching

El tópico de "*búsqueda ampliada"* fue estudiado ampliamente. Los buscadores usan alguna estrategia, en el caso de que la búsqueda lanzada no sea reconocida en el "corpus" que poseen sobre búsquedas y documentos indizados.

Las ***mínimas reglas*** que deberían aplicarse son:

- Sacar blancos del comienzo y final (***trim***). Si la palabra es compuesta, habría que sacar blancos internos.
- Buscar pasando todo a mayúscula o minúscula.
- Si se conocen abreviaturas, usarlas.
- Eliminar los símbolos de puntuación.
- Usar sinónimos, incluso entre diferentes idiomas.

Reglas específicas que deberían aplicarse para los tipos de datos:

- Fechas y sus formatos.
- Las horas y sus formatos.
- Números.
- Número decimales.
- Strings correspondientes a nombre y apellido.

Algoritmos de String Matching

Soundex

Soundex es un algoritmo fonético, es decir codifica una palabra según "suena". Intenta solucionar problemas de pronunciación.

Aquellas palabras que "suenan 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

Soundex siempre devuelve un código OUT de exactamente 4 caracteres formados por: primero una letra y luego 3 dígitos (pesos fonéticos).

Si hace falta, para completar el código de 4 caracteres, se completan con ceros al final.

Algoritmo

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 la variable $last$ 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:

> 4.1. Calcular variable $current$ con peso fonético de $iter$. Si es diferente de 0 y no coincide con $last$, appendear $current$ en OUT.
>
> 4.2. Independiente del paso 4.1, hacer $last = current$.

Paso 5. Si hace falta, completar con ceros y retornar OUT.

Soundex en Java

```
public class Soundex {
    // a b c d e f g h i j k l m n o p q r s t u v w x y z
    public static char [] pesos = {'0', '1', '2', '3', '0', '1', '2', '0', '0',
'2', '2', '4', '5', '5', '0', '1', '2', '6', '2', '3', '0', '1', '0', '2', '0',
'2'};

    public static String encode(String toCodify) {
        char[] word = toCodify.toUpperCase().toCharArray();
        char[] out = {'0', '0', '0', '0'};

        if (word.length == 0) {
            return new String(out);
        }

        out[0] = word[0];
        char last = getMapping(word[0]);

        for (int i = 1, j = 1; i < word.length && j < 4; i++) {

            char aux = getMapping(word[i]);
            if (aux != '0' && aux != last) {
                out[j++] = aux;
            }
            last = aux;
        }

        return new String(out);
    }

    public static double similarity(String s1, String s2) {
        char[] code1 = encode(s1).toCharArray();
        char[] code2 = encode(s2).toCharArray();
        System.out.println(code1);
        System.out.println(code2);
        double similarity = 0;
        for(int i = 0; i < 4; i++) {
            if(code1[i] == code2[i]) {
                similarity += 0.25;
            }
        }
        return similarity;
    }

    private static char getMapping(char toMap) {
        return pesos[toMap - 'A'];
    }
}
```

Soundex no es una métrica. Hay que definir una métrica a partir de Soundex. La ***similitud*** para Soundex es la proporción de caracteres coincidentes entre los encodings respecto a la longitud del encoding. Los valores posibles de similitud son 0.25, 0.5, 0.75, y 1.

Importar Soundex para Maven

```
 <!-- https://mvnrepository.com/artifact/commons-codec/commons-codec -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.14</version>
</dependency>
```

Metaphone

Metaphone es un algoritmo fonético, desarrollado como respuesta a las deficiencias de Soundex. Es más exacto que Soundex porque "entiende" las reglas básicas de la pronunciación en inglés.

El encoding genera símbolos de longitud arbitraria.

Reglas

1. Drop duplicate adjacent letters, except for C.
2. If the word begins with 'KN', 'GN', 'PN', 'AE', 'WR', drop the first letter.
3. Drop 'B' if after 'M' at the end of the word.
4. 'C' transforms to 'X' if followed by 'IA' or 'H' (unless in the latter case, it is part of '-SCH-', in which case it transforms to 'K'). 'C' transforms to 'S' if followed by 'I', 'E', or 'Y'. Otherwise, 'C' transforms to 'K'.
5. 'D' transforms to 'J' if followed by 'GE', 'GY', or 'GI'. Otherwise, 'D' transforms to 'T'.
6. Drop 'G' if followed by 'H' and 'H' is not at the end or before a vowel. Drop 'G' if followed by 'N' or 'NED' and is at the end.
7. 'G' transforms to 'J' if before 'I', 'E', or 'Y', and it is not in 'GG'. Otherwise, 'G' transforms to 'K'.
8. Drop 'H' if after a vowel and not before a vowel.
9. 'CK' transforms to 'K'.
10. 'PH' transforms to 'F'.
11. 'Q' transforms to 'K'.
12. 'S' transforms to 'X' if followed by 'H', 'IO', or 'IA'.
13. 'T' transforms to 'X' if followed by 'IA' or 'IO'. 'TH' transforms to '0'. Drop 'T' if followed by 'CH'.
14. 'V' transforms to 'F'.
15. 'WH' transforms to 'W' if at the beginning. Drop 'W' if not followed by a vowel.
16. 'X' transforms to 'S' if at the beginning. Otherwise, 'X' transforms to 'KS'.
17. Drop 'Y' if not followed by a vowel.
18. 'Z' transforms to 'S'.

19. Drop all vowels unless it is the beginning.

Metaphone en Java

Características de Metaphone

La similitud entre dos textos puede pensarse como la proporción de caracteres coincidentes entre los encodings respecto de la máxima longitud de los encodings obtenidos, ya que son de longitud variable.

Importar Metaphone para Maven

```
<!-- https://mvnrepository.com/artifact/commons-codec/commons-codec -->
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.14</version>
        </dependency>
```

Q-Grams

Q-Grams (ó ***N-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*.

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

$$Q\ -\ Gram(str1,\ str2)\ =\ \frac{\#QG(str1)\ +\ \#QG(str2)\ -\ QGNoShared(str1,\ str2)}{\#QG(str1)\ +\ \#QG(str2)}$$

Donde:

$\#QG(str1)$ es la cantidad de QGramas que se generaron de str1.

$\#QG(str2)$ es la cantidad de QGramas que se generaron de str2.

$QGNoShared(str1,\ str2)$ es la cantidad de QGramas que no matchearon.

Si hay QGramas repetidos, éstos sólo se pueden usar una vez, por ejemplo si str1 tiene 2 QGramas iguales y str1 tiene una vez ese mismo QGrama, se dice que sólo uno matchea. (Para más detalle ver clase 6).

Qgrams, a diferencia de los demás métodos no tiene en cuenta la posición. Para frases conviene Qgrams. Por ejemplo Metaphone y Phonemeta, y Big Data con Data big.

Qgrams en Java

```java
import java.util.HashMap;
import java.util.Map;

public class QGram {

    private final int n;

    public QGram(int n) {
        this.n = n;
    }

    public Map<String, Integer> generateGrams(String string) {
        string = addHashtag(string);

        Map<String, Integer> stringMap = new HashMap<>();
        for(int i = 0; i < string.length() - n + 1; i++) {
            String key = string.substring(i, i + n);
            int value = stringMap.getOrDefault(key, 0);
            stringMap.put(key, value + 1);
        }
        return stringMap;
    }

    private String addHashtag(String string) {
        StringBuilder aux = new StringBuilder();
        StringBuilder ans = new StringBuilder();
        for(int i = 1; i < n; i++) {
            aux.append("#");
        }
        ans.append(aux).append(string).append(aux);
        return ans.toString();
    }

    public void printTokens(String word) {
        Map<String, Integer> tokens = generateGrams(word);
        for(Map.Entry<String, Integer> entry : tokens.entrySet()) {
            System.out.printf("%s : %d\n", entry.getKey(), entry.getValue());
        }
    }

    public double similarity(String str1, String str2) {
        Map<String, Integer> token1 = generateGrams(str1);
        Map<String, Integer> token2 = generateGrams(str2);

        int inCommon = 0;
        int total = 0;

        for(Integer value : token1.values()) {
```

```
            total += value;
        }

        for(Integer value : token2.values()) {
            total += value;
        }

        for(Map.Entry<String, Integer> entry1 : token1.entrySet()) {
            for(Map.Entry<String, Integer> entry2 : token2.entrySet()) {
                if(entry1.getKey().equals(entry2.getKey())) {
                    inCommon += Math.min(entry1.getValue(), entry2.getValue()) *
2;
                }
            }
        }
        return (double)inCommon / total;
    }
}
```

Importar QGram de String-Similarity

```
<!-- https://mvnrepository.com/artifact/info.debatty/java-string-similarity -->
<dependency>
    <groupId>info.debatty</groupId>
    <artifactId>java-string-similarity</artifactId>
    <version>2.0.0</version>
</dependency>
```

Búsqueda Exacta

Hemos analizado algunos de los algoritmos que manejan "similitud" entre strings o textos. Es decir, algoritmos aproximados.

Pero sobre el procesamiento de textos hay muchos más desafíos. Por ejemplo, "Búsqueda exacta".

Búsqueda Exacta: Ideal por ejemplo para cuando realizamos una búsqueda (opcionalmente para realizar un reemplazo) en un editor de texto.

ExactSearch

Para este algoritmo se usa Fuerza Bruta o Naïve.

El algoritmo Naïve no aprovecha lo que aprendió durante el recorrido cuando encuentra un mismatch. Hace backtracking en el query y en el target.

ExactSearch en Java

```
public static int indexOf(String str1, String str2) {
        return indexOf(str1.toCharArray(), str2.toCharArray());
    }

    public static int indexOf(char[] str1, char[] str2) {
        int index = -1;
        if(str1.length < str2.length) {
            return index;
        }

        for(int i = 0; i < (str1.length - str2.length + 1) && index == -1; i++)
{
            boolean flag = true;
            for(int j = 0; j < str2.length && flag; j++) {
                if(str1[i + j] != str2[j]) {
                    flag = false;
                }
            }
            if(flag) {
                index = i;
            }
        }
        return index;
    }
```

Otra implementación es:

```
public static int indexOf(char[] query, char[] target)
{
```

```
        int idxTarget= 0;
        int idxQuery 0;

        while(idxTarget < target.length &&  idxQuery < query.length) {
                if (query[idxQuery] == target[idxTarget])   {
                        idxQuery++;
                        idxTarget++;
                        if (idxQuery == query.length)
                                return idxTarget-idxQuery;
                }
                else  {
                        idxTarget= idxTarget - idxQuery + 1;
                        idxQuery = 0;
                }
        }
        return -1;
}
```

¿Cuál es el peor caso? Que el query no se encuentre en el target

¿Complejidad temporal? O (n * m) Sea |target|= n y |source| = m

¿Complejidad espacial? O(1)

Algoritmo Knuth-Morris-Pratt (KMP)

El algoritmo ***Knuth-Morris-Pratt*** no vuelve a chequear un caracter que ya sabe que matcheo. No hace backtracking en el target.

Escanea el target de izquierda a derecha, pero usa conocimiento 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 del mismo tamaño del query.

La ***tabla de Next*** tiene en cada posición i la longitud del borde propio más grande para el substring query desde 0 hasta i.

Algoritmo para Next en Java

Sabemos que $N[i] = N[i-1] + 1$. Pero puede ser que $N[i] = 0$.

```
private static int[] nextComputation(char[] query) {
    int next[] = new int[query.length];
    next[0] = 0;
    int border = 0;
```

```
    for(int i = 1; i < query.length; i++) {
        while((border > 0) && (query[border] != query[i]))
            border = next[border - 1];
        if(query[border] == query[i])
            border++;
        //else border=0;//Redundant
        next[i] = border;
    }
    return next;
}
```

Otra implementación

```
private static int[] nextComputation(char[] query) {
    int next[] = new int[query.length];
    int border = 0;
    int rec = 1;
    while(rec < query.length) {
        if(query[rec] != query[border] {
            if(border != 0)
                border = next[border - 1];
            else
                next[rec++] = 0;
        }
        else {
            border++;
            next[rec] = border;
            rec++;
        }
    }
    return next;
}
```

¿Qué complejidad temporal tiene? ¿espacial?

Sea *m* la longitud de query y *n* la de target.

Complejidad Temporal:

En la construcción de Next lleva O(m)

Complejidad Espacial:

En la construcción de Next se usa O(m)

Una vez calculada la tabla Next, ¿Cómo se usa para calcular search sin hacer backtracking en el texto?

Idea:

Supongamos que *rec* apunta al caracter en **target** y que **pquery** que apunta a un caracter en **query**.

Mientras haya coincidencia, avanzo en ambos.

Cuando no la haya, se "shiftea" query a *next[pquery-1]*, salvo que *pquery* sea cero, en cuyo caso hay que avanzar *rec* en **target**.

Algoritmo KMP en Java

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class KMP {

    private static int[] nextComputation(char[] query) {
        int next[] = new int[query.length];
        next[0] = 0;
        int border = 0;

        for(int i = 1; i < query.length; i++) {
            while((border > 0) && (query[border] != query[i])) {
                border = next[border - 1];
            }
            if(query[border] == query[i]) {
                border++;
            }
            next[i] = border;
        }
        return next;
    }

    //Target: Palabra a consultar
    //Query: Palabra a ver si está contenida en target
    //Query C Target ? Es la pregunta
    public static int indexOf(String target, String query) {
        return indexOf(target.toCharArray(), query.toCharArray());
    }

    public static int indexOf(char[] target, char[] query) {
        int[] next = nextComputation(query);
        int rec = 0, pquery = 0;

        while(rec < target.length && pquery < query.length) {
            if(target[rec] == query[pquery]) {
                rec++;
                pquery++;
            } else if(pquery == 0) {
```

```
                rec++;
            } else {
                pquery = next[pquery - 1];
            }
        }

        if(pquery == query.length) {
            return rec - pquery;
        }
        return -1;
    }

    public static ArrayList findAll(String target, String query) {
        return findAll(target.toCharArray(), query.toCharArray());
    }

    private static  ArrayList findAll(char[] target, char[] query) {
        ArrayList<Integer> indexOf = new ArrayList<>();
        int[] next = nextComputation(query);
        int rec = 0, pquery = 0;

        while(rec < target.length && pquery < query.length) {
            if(target[rec] == query[pquery]) {
                rec++;
                pquery++;
                if(pquery == query.length) {
                    indexOf.add(rec-pquery);
                    pquery = next[pquery - 1];
                }
            } else if(pquery == 0) {
                rec++;
            } else {
                pquery = next[pquery - 1];
            }
        }

        return indexOf;
    }
}
```

El ***algoritmo KMP*** busca en forma eficiente la aparición exacta de un query en un texto, tomando ventaja del procesamiento del "query".

Pero si quisiéramos buscar un query en un "grupo de documentos", nos conviene generar un índice de dicha colección para luego buscar a través del índice. Es decir, preprocesar los documentos sobre los que se va a realizar la búsqueda (el corpus). Los términos que integran al corpus se denomina ***Vocabulario***.

Eso es lo que hacen las bibliotecas de ***Fulltext Retrieval***. Un buscador como Google o Bing generan un índice de los documentos para facilitar la búsqueda. La búsqueda se realiza a través del índice.

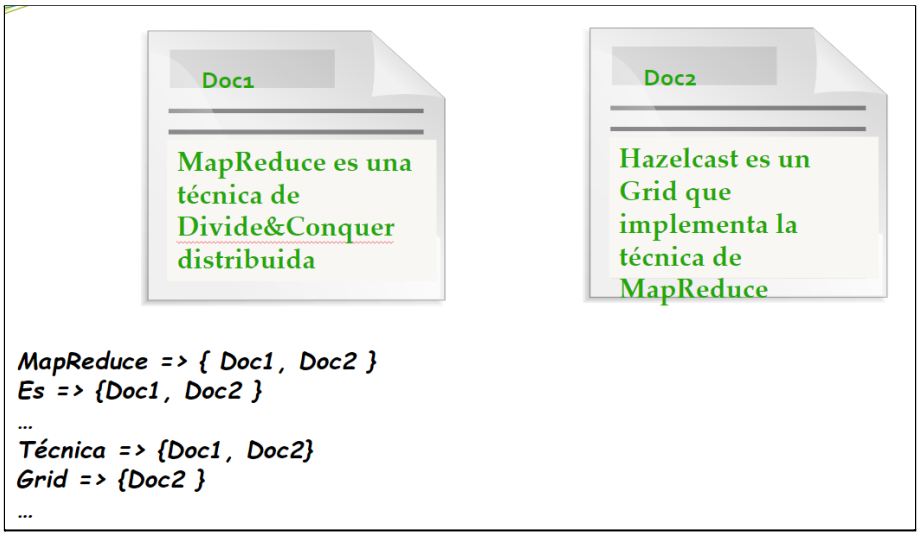
Lucene

Existe una biblioteca de código abierto para esto: ***Lucene***. Está escrita en Java (Lucene) por Douglass Cutting en 1999. Doug es uno de los fundadores de Apache Hadoop.

Nos permite armar nuestro propio "***Search Engine***". Puede incluirse en proyectos. Así, por ejemplo, otras bibliotecas como Solr y Elasticsearch, embeben Lucene para ofrecer una aplicación web que permite buscar texto en documentos.

Un ***índice*** es una estructura que permite llegar rápidamente al dato buscado. Si se agrega/elimina/actualiza un documento en la colección debe actualizarse. Su ventaja está en la búsqueda.

Lucene usa un ***Archivo Invertido***: Conjunto de términos que dicen a qué documentos pertenecen. Es un mapping: $término \rightarrow documento$.



Lucene indiza documentos. Un ***documento*** se identifica por un número y es una colección de campos (fields). Cada campo está compuesto por: nombre, tipo y valor. El tipo puede ser binario, numérico o texto.

Hay dos formas de definir un campo de tipo texto indexable:

- ***StringField***: No se separa en tokens (o sea, maneja un solo token). Puede o no almacenarse. Si elijo almacenarlo ocupa el doble de espacio: En el índice y en el almacenamiento pedido.
- ***TextField***: Se separa en tokens. Si el texto proviene de un stream no se almacena, si en cambio proviene de un string podría almacenarse. Si elijo almacenarlo ocupa el doble de espacio: En el índice y en el almacenamiento pedido.

El valor del field se indexa. Si es tipo StringField es ese token el que se indiza. Si es de tipo TextField se lo separa en tokens y esos tokens son los que van al índice.

- Si se pidió no almacenar, una vez construido el índice, el valor del field se descarta.
- Si se pidió almacenar, se lo guarda tal cual (el valor original provisto),

El método `getField(fieldName)` devuelve el valor almacenado.

Respecto de la búsqueda: Me permite consultar el índice y obtener un resultset de los documentos "más similares" rankeados por similitud,

La forma de consultar es variada: por un sólo término, expresiones booleanas, expresiones regulares, etc.

CreateMiniIndex en Java

```
import java.io.IOException;
import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.*;
import org.apache.lucene.index.*;
import org.apache.lucene.search.*;
import org.apache.lucene.store.*;
import org.apache.lucene.util.BytesRef;


public class CreateMiniIndex {

	public static void main(String[] args) throws IOException {
		IndexWriter idx = CreateIndex();
		addDoc(idx, "1234", "Leticia Gomez",
				"Estructura de datos y algoritmos",
				"Un arreglo es una estructura indexada , bla bla bla"
);

		addDoc(idx, "5678", "Leticia irene gomez",
				"algoritmos en Java",
				"Hashing es una estructura que bla bla bla" );

		SearchIndex();
	}

	static Directory directory;

	static IndexWriter CreateIndex() throws IOException
	{
		StandardAnalyzer standardAnalyzer = new StandardAnalyzer();
		IndexWriterConfig config = new IndexWriterConfig(standardAnalyzer);

		//Create a writer
		directory = new ByteBuffersDirectory();
		IndexWriter idxWriter = new IndexWriter(directory, config);
```

```
            return idxWriter;
      }

      static void addDoc(IndexWriter idxWriter, String IsbnValue, String
authorValue, String titleValue, String contentValue) throws IOException
      {
            Document aDoc;

            aDoc= new Document();

            // no tokenizable: StringField
            // tokenizable: TextField

            aDoc.add( new StringField("isbn", IsbnValue, Field.Store.NO));
            aDoc.add( new StringField("author", authorValue, Field.Store.YES));
            aDoc.add( new TextField("title", titleValue, Field.Store.NO));
            aDoc.add( new TextField("content", contentValue, Field.Store.YES));

            idxWriter.addDocument(aDoc);
            idxWriter.commit();
            // Add Second document, etc.
      }

      static void SearchIndex() throws IOException
      {
            Query query = new TermQuery(new Term("fieldName", "value"));

            //Create a reader
            IndexReader idxReader = DirectoryReader.open(directory);
            IndexSearcher searcher = new IndexSearcher(idxReader);

            TopDocs topDocs = searcher.search(query, 10);

            ScoreDoc[] x = topDocs.scoreDocs;

            System.out.println("Query=> " + query);
            System.out.println("Resultset=> ");
            for (ScoreDoc aD : x) {
                  // print info about finding
                  int docID= aD.doc;
                  System.out.println(aD);

                  // print stored document
                  Document aDoc = searcher.doc(docID);
                  System.out.println(aDoc);
            }
      }
}
```

Análisis del Código: Básicamente siempre hacemos lo siguiente:

1. `IndexWriter` para crear y mantener el índice. Puede grabarse o no en disco.
2. Una vez creado el índice, usar `IndexReader` para accederlo y usar `IndexSearcher` para buscar a través de él a través de queries. `Query` se usa para acceder a los top documentos `TopDocs` y sus hits. Search para un field que no fue indexado retorna 0.

Lucene viene equipado con la posibilidad de preguntar por diferentes ***tipos de Query*** (subclases):

- `TermQuery()`: El más básico. Acepta un único término.
- `PhraseQuery()`: Buscar frases, palabras consecutivas.
- `TermRangeQuery()`: Espera buscar dentro de un rango de términos que puede ser un intervalo [], (), [), (]. Se le indica extremo inferior, extremo superior, y 2 booleanos para indicar el tipo de intervalo.
- `BooleanQuery()`, etc.

Además, se puede elegir la ***forma de separar en tokens***:

- SimpleAnalyzer().
- StandardAnalyzer().
- WhitespaceAnalyzer().
- StopAnalyzer() ->

```
sw = new ArrayList<>();
sw.add("de");
sw.add("com");
sw.add("y");
CharArraySet stw = new CharArraySet(sw,
false);
Analyzer a= new StopAnalyzer(stw);
```

- EnglishAnalyzer().
- SpanishAnalyzer().
- CustomAnalyzer().

```
Analyzer analyzer = CustomAnalyzer.builder()
          .withTokenizer("standard")
          .addTokenFilter("lowercase")
          .addTokenFilter("stop")
          .addTokenFilter("porterstem")
          .addTokenFilter("capitalization")
          .build();
```

Ahora vamos a grabar el índice en disco. Además, vamos a usar para algunos de los TextField información que reside en el disco (Archivo de texto).

Lucene precisa saber:

- Donde queremos grabar la estructura de índice en disco.
- Si algunos de los fields de un documento son archivos de disco, dónde se almacenan para ir a buscarlos.

Programación Dinámica

La ***Programación Dinámica*** es una técnica que consiste en reutilizar valores previamente calculados para no tener que recalcularlos repetidamente. Sirve, si para cierto cálculo, pueden reusarse valores previos.

Así, los valores calculados deben almacenarse en una estructura de datos con el objetivo de "buscarlos" (lookup) y no calcularlos nuevamente cuando se los precise.

Desde el punto de vista de complejidad algorítmica computacional, una operación lookup podrías tener un costo bajísimo.

Levenshtein Distance

La ***Distancia de Levenshtein*** es un algoritmo que calcula la cantidad mínima de operaciones necesarias para transformar un string en otro. Las operaciones válidas son: ***insertar***, ***borrar*** y ***sustituir un caracter***.

Aquellos strings que son iguales deben tener distancia cero porque no hace falta transformar uno en otro.

A diferencia de Soundex, que se adoptó para proponer una medida de similitud, a partir de su cálculo, Levenshtein ES una métrica de distancia. Por lo tanto, cumple con propiedades:

1. Simetría. $Levenshtein(str1,\ str2)\ = Levenshtein(str2,\ str1)$.
2. Desigualdad.
   $Levenshtein(str1,\ str2)\ = Levenshtein(str2,\ str3) \geq Levenshtein(str1,\ str3)$

Es un algoritmo que se suele implementar con programación dinámica.

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

Se puede normalizar para que el número obtenido esté entre 0 y 1. El valor 1 implica coincidencia.

$$LevenshteinNormalized(str1,\ str2) = 1 - \frac{Levenshtein(str1, str2)}{max(str1.len, str2.len)}$$

Levenshtein en Java

Versión con ***matriz***:

```
public class Levenshtein {

    private static int minimum(int a, int b, int c) {
        return Math.min(a, Math.min(b, c));
    }

    public static int distance(String str1, String str2) {
        return distance(str1.toCharArray(),
                str2.toCharArray());
    }

    private static int distance(char [] str1, char [] str2) {
        int [][]distance = new int[str1.length+1][str2.length+1];

        for(int i = 0; i <= str1.length; i++) {
            distance[i][0]=i;
        }
        for(int j = 0; j <= str2.length; j++) {
            distance[0][j]=j;
        }
        for(int i = 1; i <= str1.length; i++){
            for(int j = 1;j <= str2.length; j++){
                distance[i][j]= minimum(distance[i-1][j]+1,
                        distance[i][j-1]+1,
                        distance[i-1][j-1]+
                                ((str1[i-1]==str2[j-1])?0:1));
            }
        }
        return distance[str1.length][str2.length];
    }

    public static double normalizedSimilarity(String str1, String str2) {
        return 1 - (double)distance(str1, str2)/Math.max(str1.length(),
str2.length());
    }
}
```

Versión con ***dos vectores***:

```
import java.util.Arrays;

public class LevenshteinEnhanced {
    private static int minimum(int a, int b, int c) {
```

```
        return Math.min(a, Math.min(b, c));
    }

    public static int distance(String str1, String str2) {
        return distance(str1.toCharArray(),
                str2.toCharArray());
    }

    private static int distance(char[] str1, char[] str2) {
        if (str1.length > str2.length) {
            char[] swap = str1;
            str1 = str2;
            str2 = swap;
        }

        int[] previous = new int[str1.length + 1];
        int[] current = new int[str1.length + 1];

        for (int i = 0; i <= str1.length; i++) {
            previous[i] = i;
        }
        for (int i = 1; i <= str2.length; i++) {
            current[0] = i;
            for (int j = 1; j <= str1.length; j++) {
                current[j] = minimum(previous[j] + 1,
                        current[j - 1] + 1,
                        previous[j - 1] + ((str1[j - 1] == str2[i - 1]) ? 0 :
1));
            }
            previous = Arrays.copyOf(current, current.length);
        }
        return current[str1.length];
    }

    public static double normalizedSimilarity(String str1, String str2) {
        return 1 - (double) distance(str1, str2) / Math.max(str1.length(),
str2.length());
    }
}
```

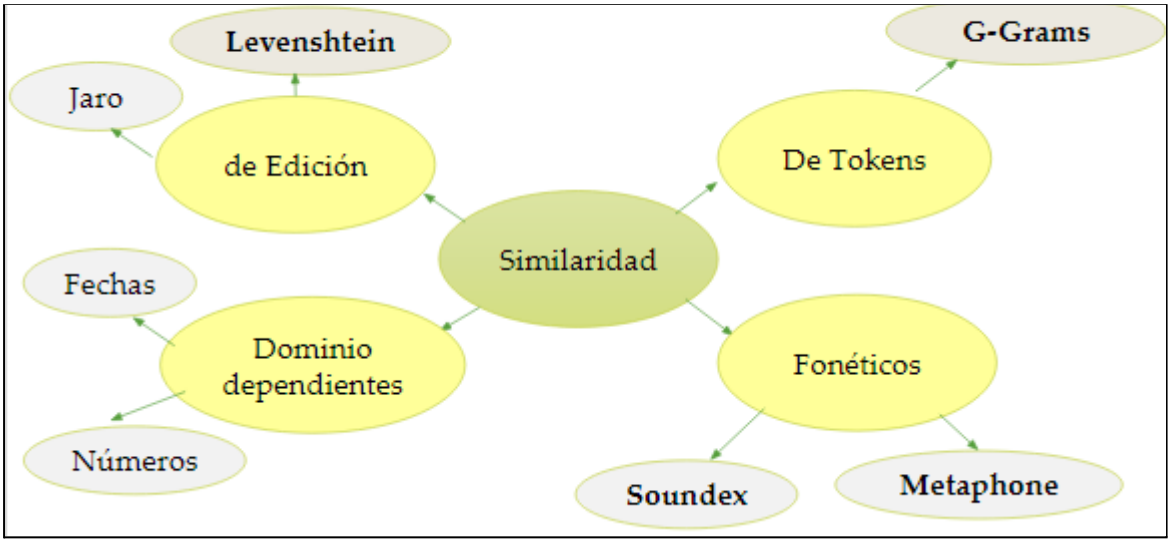
Existen variantes, por ejemplo:

- Damerau-Levenshtein: Las operaciones no son solo borrado, inserción y sustitución. También se agrega transposición.
- Otras variantes no consideran que las operaciones valen todas igual. Alguna es más cara que otra y cambia la fórmula de distancia.

Importar Levenshtein de Apache

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-text -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-text</artifactId>
    <version>1.9</version>
</dependency>
```

Resumiendo. los algoritmos representantes que hemos analizado sobre procesamiento de Strings



Estructuras Lineales (En construcción)

Cuando se busca una o varias apariciones de un elemento en un conjunto se puede proceder de diferentes formas:

1. Dejar la colección como está y navegar en ella. Ese fue el caso de los algoritmos Naïve y de KMP. En ese caso, el texto es una colección de caracteres y no se modifica para que no se pierda su semántica.
2. Generar una estructura auxiliar, llamada índice, que facilite la búsqueda. Ese fue el caso del archivo invertido. Los documentos no se modifican para que no pierdan su semántica, pero se crea un índice. Se busca sobre el índice. Si la búsqueda resulta exitosa en él, entonces se llega al documento.

Algoritmos sobre índices

Se llama ***índice*** a la estructura de datos que facilita la búsqueda (lookup). Un índice está compuesto por elementos que representan la información que indizan.

Siendo el índice la estructura auxiliar que se utiliza para encontrar un elemento, entonces la búsqueda sobre el mismo debe ser **muy eficiente**.

Características de los índices:

- La clave de búsqueda puede o no, tener repetidos. Si es única no tiene sentido hablar de compactación. En caso contrario, podremos tener información adicional compactada.
- La clave de búsqueda debe permitir buscar rápidamente la información adicional.

Ahora bien, el índice **no sólo se utiliza para buscar**. Hay otras operaciones necesarias sobre él:

- **Búsqueda**: Para ello se lo construye.
- **Inserción**: El índice debe reflejar los datos de la colección. O sea, si inserto un documento en la colección, preciso que el índice lo refleje.
- **Borrado**: Si borro un documento en la colección, preciso que el índice lo refleje.

Algunos ejemplos para índices pueden ser:

	Búsqueda	**Inserción**	**Borrado**
Arreglo	☐ O(n)	✔ O(1)	☐ O(n)
Arreglo ordenado por clave de búsqueda	✔ O(log2 n)	◎ O(n)	◎ O(n)
Hashing	✔ ??	✔ ??	✔ ??

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

El problema del hashing es que tiene que resolver colisiones. Si no tiene colisiones es $O(1)$.

No hay estructuras de datos perfectas. Dependen de los casos de uso necesarios. En general los objetivos se contraponen y hay que buscar un trade-off.

Arreglos Ordenados

Para la búsqueda se usan arreglos ordenados y se aprovecha la "***Búsqueda Binaria***". Su complejidad es $O(log_2 n)$.

Búsqueda Binaria en Java

Versión Iterativa:

```
public static boolean iterative(int arr[], int x) {
        int L = 0;
        int R = arr.length - 1;
        while(L <= R) {
            int i = L + (R-L) / 2;
            if(arr[i] < x) {
                L = i + 1;
            }
            else if(arr[i] > x) {
                R = i - 1;
            } else {
                return true;
            }
        }
        return false;
    }
```

Versión Recursiva:

```
public static boolean recursive(int arr[], int x) {
        return recursive(arr, x, 0, arr.length - 1);
    }

    private static boolean recursive(int arr[], int x, int l, int r) {
        if(l <= r) {
            int i = l + (r-l) / 2;
            if(arr[i] == x) {
                return true;
            }
            if(arr[i] > x) {
                return BinarySearch.recursive(arr, x, l, i - 1);
            }
            if(arr[i] < x) {
                return BinarySearch.recursive(arr, x, i + 1, r);
            }
```

```
        }
        return false;
    }
```

IndexService en Java

IndexService es una interface que permite la búsqueda, inserción, remoción y contar ocurrencias en un arreglo.

Contrato de la interface:

```
public interface IndexService {

    //elements serán los valores de
    void initialize(int[] elements);

    boolean search(int key);

    void insert(int key);

    void delete(int key);

    int ocurrences(int key);
}
```

La implementación de la interface es la siguiente:

La complejidad de estos algoritmos es

Algoritmos de Ordenación

Lenguaje Tipado y Generics de Java

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

Con la introducción de **Java Generics** ese tipo puede parametrizarse. Técnicamente hablando, cuando hay Generics, el compilador aplica **Erasure**: Reemplaza todo tipo de parámetro en Generics con su "bound/restricción" y si no lo hay lo reemplaza por Object. Si precisa, agrega casteos.

En Java los Generics son invariantes, no se puede asignar un subtipo generics a un supertipo generics.

Hay muchas restricciones que se establecieron al diseñar en Java Generics y Erasure. Por ejemplo:

- No puede un built-in sustituir un tipo paramétrico.
- No puedo crear dinámicamente un arreglo de tipo paramétrico (en tiempo de ejecución) porque su tipo ya no se conoce ya que en compilación se hizo erasure.

Restricciones de Generics

IndexParametricService en Java

Stack (En construcción)

Un **Stack**, **LIFO** o **Pila** es una colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de un elemento distinguido: *Tope* que es el último elemento en llegar.

Las operaciones que debe ofrecer son:

- **push**: Agrega un elemento a la colección y se convierte en el nuevo tope.
- **pop**: Quita un elemento de la colección y cambia el tope de la pila. Es una operación destructiva y solo puede usarse si la colección no está vacía.
- **isEmpty**: Devuelve true/false según la colección tenga o no elementos.
- **peek**: Devuelve el elemento tope pero sin removerlo. Es una operación de lectura y solo puede usarse si la colección no está vacía.

Si se lo implementa con un **arreglo** la contigüidad está garantizada. Nunca quedarían huecos internos y no haría falta mover elementos para garantizar contigüidad ya que las operaciones solo se acceden por el tope.

Sin embargo, si se acaba el espacio sí debemos buscar espacio contiguo en otro lugar y copiar componentes. Es decir, nos conviene hacer crecer/decrecer la estructura de a “chunks”.

Si se lo implementa con una **lista lineal simplemente encadenada**, como los elementos sólo se acceden por el tope, es sólo cuestión de apuntar el tope al elemento conveniente. Es decir, el primer elemento de la lista. Así, jamás tenemos que recorrer para push/pop.

Resulta más conveniente la **lista lineal**.

Java viene equipado con una clase para el stack. En Java se cometió un importante error de diseño, pues se extiende de un arreglo.

Stack - Java

Discusión sobre Stack en Java

La implementación correcta de la clase Stack es encapsulando una LinkedList (o arreglo), no especializando alguno de ellos.

```
public class Stack<T> {
   private LinkedList<T> data = new LinkedList<>();

   public void push(T v) { data.addFirst(v); }
   public T peek() { return data.getFirst(); }
   public T pop() { return data.removeFirst(); }
   public boolean isEmpty() { return data.isEmpty(); }
}
```

Evaluador de Expresiones

Una **expresión** es una combinación de operadores y operandos. En esta discusión vamos a considerar un subconjunto de operados: los binarios.

Las expresiones se pueden clasificar según la notación que utilizan:

- **Prefija**: El operador se encuentra **antes** de los operandos sobre los que aplica.
- **Infija**: El operador se encuentra **entre** los operandos sobre los que aplica.
- **Postfija**: El operador se encuentra **detrás** de los operandos sobre los que aplica.

Por ejemplo, dados dos operandos A y B, el operador binario * puede tener las siguientes posibilidades:

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

En las notaciones prefija y postfija el uso de paréntesis es **innecesario** debido a que el orden de los operandos determina el orden real de las operaciones en la evaluación de expresiones.

Algoritmo para evaluar una expresión que ya esté en notación postfija

- Cada operador en una expresión postfija se refiere a los operandos previos a la misma.
- Cuando aparece un operando hay que postergarlo porque no se puede hacer 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 expresión.
- Cuando se termine de analizar la expresión de entrada el resultado de su evaluación es el único valor que quedó en la **pila**.

Es un buen diseño utilizar una pila porque, como se observó, la única forma de acceso a la estructura de datos fue a través de su tope. Jamás se necesitó navegar por dentro de la estructura en busca de otras componentes. Siempre se respetó el orden de llegada de los elementos a la estructura.

Acá tenemos dos problemas:

a. Cómo parsear un string de entrada para separarlo en tokens válidos (dónde terminan los números y los operadores)

b. La evaluación en sí de la expresión postfija. Eso incluye el manejo de errores.

Empecemos analizando qué clase de Java simplifica el análisis de tokens. La clase **Scanner** permite leer de standard input/archive/string información, indicarle cuáles son los separadores y navegar por sus tokens (iterador).

La clase **Evaluator** lee de entrada estándar el input que se supone corresponde a una expresión aritmética en notación **postfija** formada por: constantes numéricas, y {+, -, *, /}.

Ahora bien, no vamos a esperar que el usuario la ingrese en forma postfija. Analizaremos cómo transformar una expresión infija en una postfija.

Algoritmo para transformar expresión en notación infija a expresión en notación postfija

Cada vez que aparezcan varios operadores se consultará 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.

En la siguiente tabla se representa si el tope de la pila tiene mayor precedencia que el elemento actual:

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

El **algoritmo** es el siguiente:

Cada **operando** de la expresión infija se copia en la expresión postfija.

Cuando aparece un **operador** hay que analizar la precedencia respecto del resto de los previos operadores, por lo tanto los casos se reducen a chequear la precedencia entre el **tope de la pila** y el **operador current**:

- Si la pila está vacía, **se pushea el operador current** ya que no se lo puede comparar con nada, porque es el primero de la expresión.
- Si la pila no está vacía:

    - Si el tope tiene mayor precedencia que el operador current, entonces se realiza el pop del operador en la pila y se lo copia en la expresión postfija hasta que se acabe la pila o quede en ella uno de menor precedencia que el operador current. Se pushea al operador current, ya que hay que postergar su acción hasta que aparezca otro operador.
    - Si el tope de la pila tiene menor precedencia que el operador current no se puede ir todavía. Se pushea al operador current, ya que hay que postergar su acción hasta que aparezca otro operador.
- Cuando se terminó de analizar la expresión infija, se popean todos los operadores de la pila y se copian en la expresión postfija.

Clase 14B

Listas (Faltan colocar implementaciones)

Lista Lineal Simplemente Encadenada

Una **Lista Lineal Simplemente Encadenada** es una estructura de datos compuesta por 0 o más nodos. Cada **nodo** (elemento) almacena 2 datos: Su información y la referencia al siguiente elemento.

Una variante es una **Lista Lineal Simplemente Encadenada Ordenada**, que mantiene los elementos ordenados con algún criterio de ordenación.

Una **Lista Lineal Simplemente Encadenada 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.
- Cada **nodo**/elemento (común) almacena 2 datos: Su información y la referencia al siguiente elemento.

Es decir, hay 2 tipos de nodos: header y comunes.

- El nodo header no tiene que ser comparable. Hay uno solo de ese tipo de nodo.
- Los nodos comunes tienen que poder compararse entre sí.

Uso de Iteradores

Para borrar o insertar un elemento hay 2 situaciones que se dan:

a. Borrar/insertar un elemento recorriendo **desde el header**: Tenemos que la búsqueda se hace en O(n). Aunque no hay movimiento de datos porque no se precisa contigüidad, la operación es entonces **O(n)**.

b. Borrar/insertar un elemento **sin buscarlo**: Estamos parados en el elemento y como no hay movimiento de datos para garantizar contigüidad, entonces sería **O(1)**. Para estar "apuntando al elemento a borrar" vamos a colocar el `delete()` en el **iterador**.

La interface `Iterator` permite implementar el método `remove()`, es opcional.

Posibles problemas al usar iteradores

Si el iterador es con `remove()` hay cosas que no puede chequearse y pueden producir un problema en tiempo de ejecución: uso de cursores anidados, donde uno de ellos elimina un elemento (el otro que había chequeado que había elementos obtiene un error porque el elemento ya no está).

```java
public static void main(String[] args) {

SortedListService<Integer> a= new SortedLinkedListWithHeaderAllowsRemoves<>();
a.add(10);

for (Iterator<Integer> iter1 = a.iterator(); iter1.hasNext();) {
    Integer nro1;
    nro1=  iter1.next();
        Iterator<Integer> iter2 = a.iterator();
        if ( iter2.hasNext() )
        {
            iter1.remove();
            Integer nro2 =  iter2.next();
        }
    }
}
```

```java
public static void main(String[] args) {

SortedListService<Integer> a= new SortedLinkedListWithHeaderAllowsRemoves<>();
a.add(10);

for (Iterator<Integer> iter1 = a.iterator(); iter1.hasNext();) {
    Integer nro1;
    nro1=  iter1.next();
        Iterator<Integer> iter2 = a.iterator();
        if ( iter2.hasNext() )
        {
            iter1.remove();
            Integer nro2 =  iter2.next();
            iter2.remove();
        }
    }
}
```

Es importante no invocar en el `remove()` del cursor.

La implementación del iterador con `remove()` en O(1) no es algo trivial. Es más complicado que con iterador readonly. Implícitamente hay muchos casos que resolver. Es un autómata con estados por los que se va pasando frente a las operaciones invocadas sobre el iterador.

Típicamente se revuelve:

1. Tres variables: prevprev, prev y current que se van moviendo paralelamente y a un elemento de distancia.
2. Dos variables: prev y current, ya que cuando se remueve un elemento hay que “rebobinar” un elemento hacia atrás, pero como la lista es simple encadenada, volver atrás en O(1) implica haber guardado la posición anterior. Si se usan 2 variables hay que complicar levemente la lógica (chequear algunas condiciones más).

El método `remove()` lanza una `IllegalStateException()` si:

- Se invocó a `remove()` sin haber invocado a `next()`.
- Se invocó a `remove()` dos veces seguidas.

	Búsqueda	Inserción desde el header	Inserción desde iterador	Borrado desde el header	Borrado desde iterador
Arreglo ordenado por clave de búsqueda	✔ O(log n)	× O(n)	× O(n)	× O(n)	× O(n)
Lista lineal simplemente encadenada ordenadad por clave de búsqueda	× O(n)	× O(n)	✔ O(1)	× O(n)	O(1)

LinkedList en Java

Lista Lineal Doblemente Encadenada

Una **Lista Lineal Doblemente Encadenada con Header** es una estructura de datos compuesta por:

- Un elemento distinguido llamado **header** que tiene la referencia del primer elemento y además información global de la lista.
- Cada nodo/elemento almacena 3 datos: Su información, y la referencia a los elementos previo y siguiente.

Una variante es una **Lista Lineal Doblemente Encadenada Ordenada con Header**, que mantiene los elementos ordenados con algún criterio de ordenación.

Double Linked List en Java

Listas Circulares

Existen las **Listas Circulares**, que pueden ser encadenadas simples o dobles, en las que el el último elemento, posee la referencia al primer elemento de la misma.

CircularList en Java

Grafos (Falta colocar implementaciones)

Un ***Grafo*** es un conjunto de nodos (o vértices) y ejes (o aristas). Cada arista conecta dos nodos.

Características

- Cada arista puede tener un valor asociado, lo que lo convierte en un ***grafo con pesos***.
- Si las aristas pueden recorrerse en un único sentido, se representan con una flecha y el grafo se llama ***digrafo*** o ***grafo dirigido***.
- Si entre dos nodos puede haber más de una arista, el grafo se llama ***multigrafo***.
- Cuando una arista conecta a un nodo con sí mismo, se llama un ***lazo***.

Los grafos pueden ser representados a través de:

- Matriz de adyacencia
- Lista de adyacencia
- Matriz de incidencia
- Lista de incidencia.

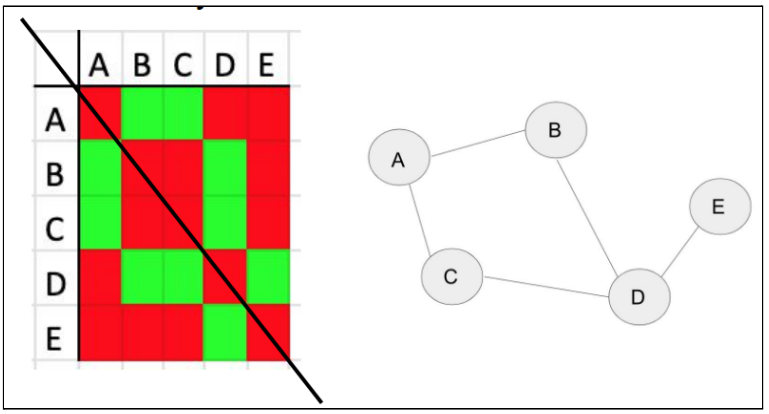
Matriz de adyacencia

La ***Matriz de Adyacencia*** es muy útil para representar grafos **densos**. Es muy ineficiente en memoria para el resto de los grafos.

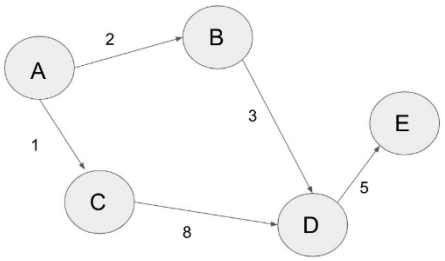
Si el grafo tiene pocas aristas por nodo (sparse en inglés), la mayoría de las celdas de la matriz quedan vacías, por lo que se desperdicia mucho espacio.

Agregar o leer ejes es muy simple. Agregar nodos no es eficiente, ya que hay que recrear la matriz para hacer espacio.

Se le asigna un índice a cada nodo que le corresponde una fila y una columna de una matriz cuadrada. Si el nodo *i* y el nodo *j* tienen una arista en común, se representa en la posición *i,j* de la matriz.



	A	B	C	D	E
A	0	2	1	0	0
B	0	0	0	3	0
C	0	0	0	8	0
D	0	0	0	0	5
E	0	0	0	0	0



Matriz de incidencia

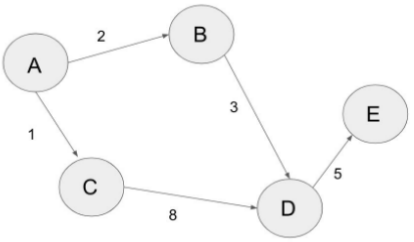
Cada fila representa un nodo y cada columna una arista. En la posición *[i,j]* se indica si la arista *j* conecta al nodo *i*.

Cada columna contiene solo dos valores, por lo que no es muy eficiente en espacio.

Lista de Adyacencia

Los nodos se guardan en una lista, y cada uno contiene a su vez otra lista con sus nodos adyacentes.

A	B	C	D	E
B - 2	D - 3	D - 8	E - 5	
C - 1				

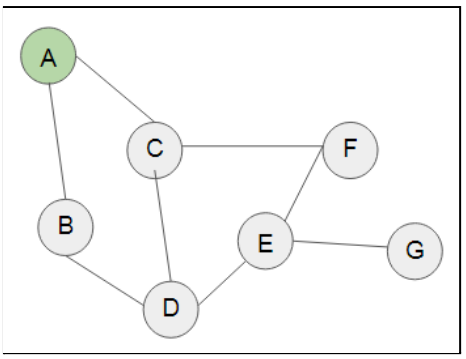


Formas de recorrer un grafo

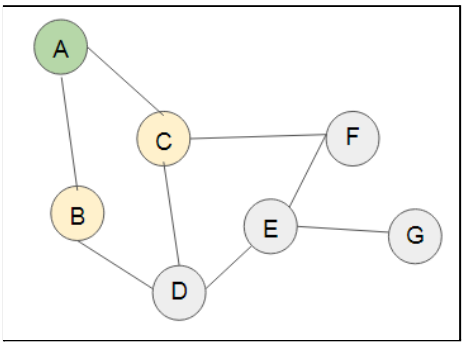
BFS

En ***BFS*** (***Breadth-First Search***) se recorre "de a niveles". Se empieza marcando el nodo inicial y luego en cada paso se recorren y marcan los nodos no marcados que son adyacentes a un nodo marcado.

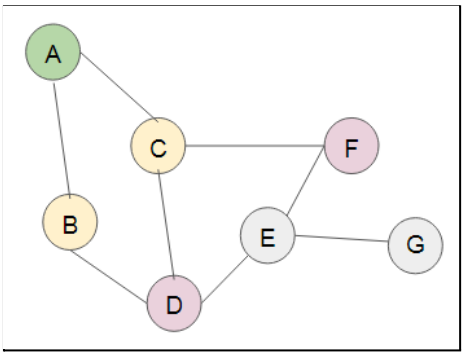
Por ejemplo, empezando por la A.



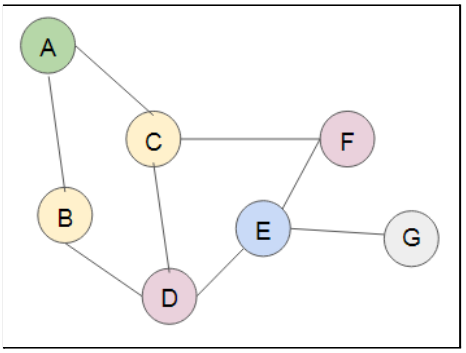
El siguiente nivel es B-C.

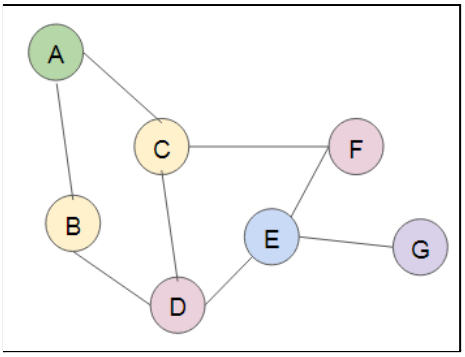


El siguiente nivel es D-F.



El siguiente nivel es la E.





BFS en Java

En código, esto se puede representar con una cola. La cola mantiene los nodos que quedan por visitar y por cada paso se toma un nodo y añaden a la cola todos sus vecinos que no están marcados.

La complejidad de este algoritmo es **O(N+E)** siendo *N* la cantidad de nodos y *E* la cantidad de aristas. Esto es porque cada nodo se recorre una única vez y cada eje se recorre 2 veces (1 por cada nodo que conecta).

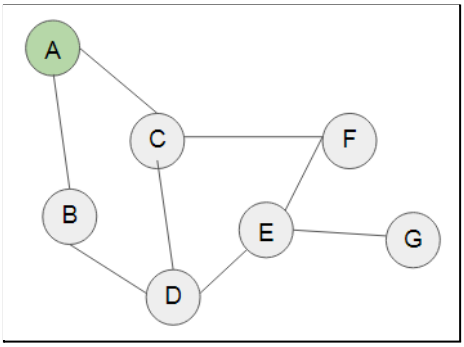
BFS con Matriz de Adyacencia en Java

DFS

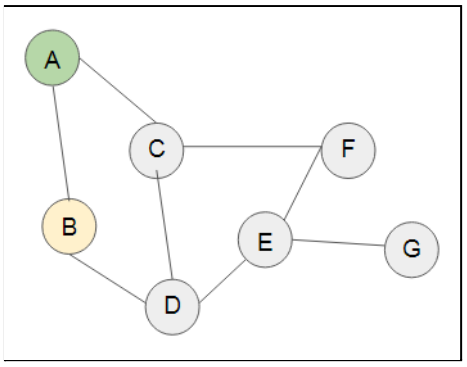
En ***DFS*** (***Depth-First Search***) se recorre "por profundidad". Se siguen los siguientes pasos, empezando por el nodo inicial:

1. Dado un nodo, se marca como visitado y elige uno de los vecinos no visitados al azar.
2. Nos paramos en ese nodo vecino y hacemos lo mismo que en el paso anterior.
3. Después de terminar de visitar un vecino en profundidad, visitamos el resto de ellos de igual forma.

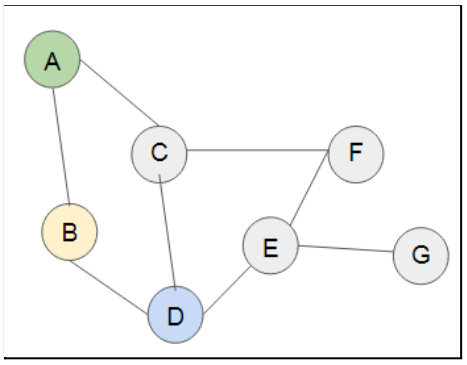
Por ejemplo, empezando por la A.



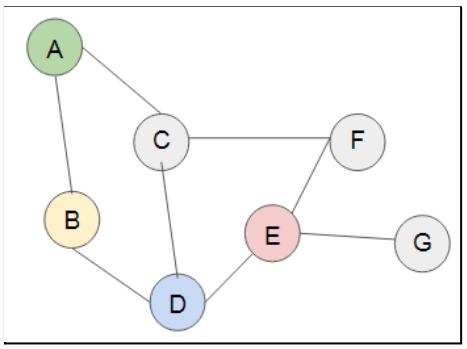
Podría seguir la B o la C. Vamos a seguir con la B.



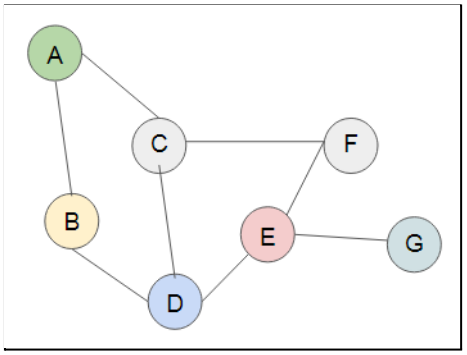
Debe seguir la D.



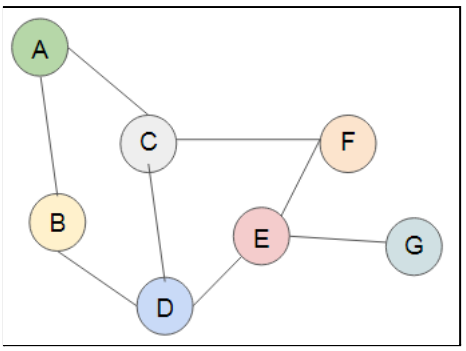
Podría seguir la C o la E. Vamos a seguir con la E.



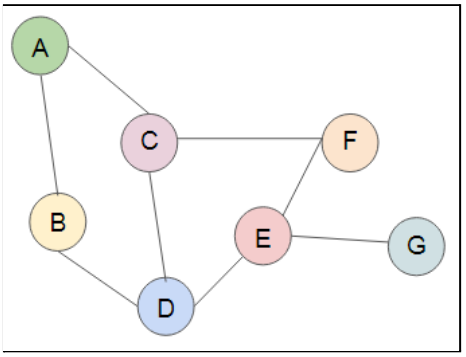
Podría seguir la G o la F. Vamos a seguir con la G.



La G no tiene a donde ir. Entonces, volvemos a la E y debe seguir por la F.



Desde la F, sólo queda visitar la C.



DFS en Java

Este código se puede representar con una pila. La forma más fácil de representar la pila es con una función recursiva, pero se puede hacer también de forma iterativa haciendo uso de la clase `Stack`. En cada paso de la función recursiva se recibe un nodo, se le aplica una función (En este caso imprimir el valor del nodo) y se llama la función recursivamente con todos los vecinos no marcados.

La complejidad de este algoritmo es **O(N+E)** siendo *N* la cantidad de nodos y *E* la cantidad de aristas. esto es porque cada nodo se recorre y marca una única vez y cada eje se recorre dos veces.

Dijkstra

El objetivo de ***Dijkstra*** es encontrar el camino más corto entre cualquier par de nodos cuyas aristas tengan pesos no negativos. Si se tiene aristas con pesos negativos, la alternativa es el algoritmo de Bellman-Ford, la desventaja de este último es que tiene un peor orden de complejidad.

Los pasos de algoritmo de Dijkstra son:

1. Empezamos marcando todos los nodos con infinito excepto el nodo inicial. Este número representa el menor costo conocido de llegar hasta ese nodo.

2. En cada paso, miramos al nodo con menor costo que aún no haya sido visitado y actualizamos el costo de sus vecinos.

Este algoritmo funciona porque cuando nos paramos en un nodo sabemos que no es posible encontrar un camino más corto hacia él, ya que es el nodo con menor costo asociado. Si hubiera otro nodo con costo menor nos hubiéramos parado en ese otro.

Dijkstra en Java

Para implementar este algoritmo, puede resultarnos útil hacer uso de una PriorityQueue. Además, a la implementación de grafo con pesos, debemos agregarle un campo `cost` a la clase `Node` que debe ser numérico y una clase auxiliar `PqNode` que usaremos para usar de nodo en la `PriorityQueue`.

El campo `cost` de la clase `Node` representa el menor costo posible para llegar hasta ese nodo con las aristas recorridas hasta el momento y el campo cost de `PqNode` representa el camino de un recorrido particular. Podrías haber más de un `PqNode` con el mismo nodo, pero se visitará primero el que tenga menor `cost`. Por eso es importante marcar un nodo al visitarlo y no volver a analizarlo si ya ha sido visitado.

La complejidad de este algoritmo es **O((N+E)*log(N))** siendo *N* la cantidad de nodos y *E* la cantidad de aristas. La *E* es porque cada arista se recorre 2 veces (1 por cada nodo que conecta) y *N* surge de que cada nodo se recorre una única vez. Los *log(N)* sumados a ambas letras salen de que cada vez que se analiza un nodo hay que hacer un `remove` de la `PriorityQueue` que tiene costo *log(N)* y los `add` de la `PQ` se terminan haciendo *E* veces. Existen formas de modificar el algoritmo para que la complejidad en el peor de los casos termine siendo **O(N*log(N)+E)** lo cual es una mejora para grafos densos especialmente.

PriorityQueue

PriorityQueue es una cola que devuelve siempre el menor elemento (en lugar del primero que haya sido insertado).

Los métodos `add` y `remove` tienen complejidad $O(log(N))$.

PriorityQueue - Documentación Java

Hashing (Falta colocar implentacion)

Hashing es una estructura de datos que utiliza un arreglo (lookup table) para almacenar *pares key/value* de una forma especial. No mantiene ni contigüidad, ni orden de las componentes.

Prioriza la búsqueda, tratando de que el algoritmo tenga complejidad cercana a $O(1)$.

Sabemos que los arreglos tienen el problema de re alocación del espacio y hay que realocar cuando el espacio es insuficiente. En una primera aproximación vamos a suponer que hay espacio suficiente para almacenar los pares *key/value*.

Sea **Keys** el conjunto de claves a hashear, y sea **LookUp** el conjunto de ranuras (arreglo) para albergar los pares *key/value*. Asumimos $|Keys| \Leftarrow |LookUp|$, es decir, hay posibilidad de que a cada key se le asigne una ranura en LookUp.

Función de Hash

Para minimizar la búsqueda de elementos, el LookUp será usado de la siguiente manera:

- El lugar que ocupa el elemento sea calculado por alguna fórmula:

  $hash: Keys \rightarrow LookUp$

  donde `LookUp[hash(key)]` contiene el valor.
- Siempre debe garantizarse que el `hash(key)` devuelve una ranura válida. Por lo tanto, `hash(key)` se define como:

  $hash(key) = prehash(key)\ \%\ |LookUp|$
- Así. permitimos que el usuario pueda proporcionar la función `prehash()`, sin conocer el tamaño alocado para el LookUp.

Se dice que dos claves $key1 \neq key2$ **colisionan** si $hash(key1) = hash(key2)$, es decir, se les asigna la misma ranura.

Se dice que una **función de Hash es perfecta** si no produce colisiones. Es decir, $key1 \neq key2 \Rightarrow hash(key1) \neq hash(key2)$. Sería una función inyectiva.

Una fn perfecta no es fácil de encontrar. Además se tiene un universo de claves posibles (aunque no se las precise hashear a todas) y por lo tanto, se debe garantizar que nunca van a colisionar.

Una **fn de hash buena** (aunque no perfecta) será aquella que minimiza la cantidad de colisiones.

Opciones para función de Hash

Cuando las claves son numéricas existen varias funciones posibles:

- **División** o **Módulo**: $hash(x) = x\ mod\ m$. Donde m debe ser número primo.

- **Mid-Square**: Se calcula el cuadrado de un número y se toman los bits centrales como lugar donde hashear. Por ejemplo: Si $x = 14$, $x \times x = 23420$, y hay que tomar los bits centrales para poder hashear. Si la tabla tuviera 11 elementos, usaría el número $34\%11$, ó $42\%11$.
- **Folding** o **Plegado**: Se divide el número en zonas de la misma longitud. Se las suma y se toman los bytes necesarios. Por ejemplo: Si el número es 20112241203123 y la tabla tienen 101 elementos, se arman grupos de a 3 dígitos, se obtiene: $020 + 112 + 241 + 203 = 699$ y se lo hashea a $699\%101$.
- **Análisis del dígito**: Implica un conocimiento previo de las características de la población. Se analizan los patrones de las claves, en busca de la información de la clave que menos repite. Por ejemplo: Si las claves para hashear los alumnos de una facultad en cierto año fuera su DNI, no convendría elegir sus primero dígitos porque todos los alumnos de un mismo año comienzan con las mismas cifras de DNI.

A efecto de mejorar la dispersión conviene que el tamaño de la tabla sea un **número primo**.

Factor de Carga

En algún momento el LookUp table puede quedarse sin lugar y hay que incrementar el tamaño del LookUp y rehashear.

El **Factor de Carga** (**Current Load Factor**) se define como: $\frac{|Keys\ usadas|}{|LookUp|}$. Cuando el factor de carga supera un umbral predefinido (**Load Factor Threshold**) hay que duplicar espacio y rehashear.

Hash en Java - Sin repetidos y no resuelve colisiones

```
public class Hash<K, V> {
    private final int initialLookupSize = 19;
    private Node<K, V>[] LookUp = new Node[initialLookupSize];
    private final Function<? super K, Integer> prehash;
       private final double threshold = 0.75;

    public Hash(Function<? super K, Integer> mappingFn) {
        prehash = mappingFn;
    }

    private int hash(K key) {
        if (key == null)
            throw new RuntimeException("No key provided");
        return prehash.apply(key) % LookUp.length;
    }

    public V getValue(K key) {
        Node<K, V> entry = get(key);
```

```
        if (entry == null)
            return null;

        return entry.value;
    }

    private Node<K, V> get(K key) {
        return LookUp[hash(key)];
    }

    // insert = update
    public void insert(K key, V value) {
        LookUp[hash(key)] = new Node<K, V>(key, value);

        if(getCurrentLoadFactor() > threshold) {
             rehash();
             }
    }


    public void delete(K key) {
        LookUp[hash(key)] = null;
    }

    public void dump() {
        for (int rec = 0; rec < LookUp.length; rec++)
            if (LookUp[rec] == null)
                System.out.printf("slot %d is empty\n", rec);
            else
                System.out.printf("slot %d contains %s\n", rec, LookUp[rec]);
    }

    private void rehash() {
            Node<K, V>[] newLookUp = new Node[LookUp.length * 2];
            for(int i = 0; i < LookUp.length; i++) {
                if(LookUp[i] != null) {
                    newLookUp[hash(LookUp[i].key)] = LookUp[i];
                }
            }
            LookUp = newLookUp;
       }

    private double getCurrentLoadFactor() {
       int usedKeys = 0;
       for(int i = 0; i < LookUp.length; i++) {
             if(LookUp[i] != null) {
                   usedKeys++;
                   }
             }
       return ((double)usedKeys)/LookUp.length;
```

```
    }

    static class Node<K, V> {
        final K key;
        V value;

        Node(K theKey, V theValue) {
            key = theKey;
            value = theValue;
        }

        public String toString() {
            return String.format("key = %s, value = %s", key, value);
        }
    }

}
```

Colisiones

Existe dos formas de resolver las colisiones:

- **Open Addressing** or **Closed Hashing**: Dentro de la misma tabla de hashing se guardan los elementos que colisionaron.
- **Open Hashing** or **Chaining**: Fuera del hashing se almacenan los elementos que colisionaron.

Open Addressing o Closed Hashing

Cada ranura puede tener null (está vacío o **baja física**). Aunque la ranura no esté vacía puede ser que el elemento no esté, ya que hay que manejar el concepto de **baja lógica** (además de las físicas). Es decir, una ranura representa 3 estados: tiene un elemento o no tiene un elemento (dado por baja lógica o bien por baja física).

Formas típicas de resolver una colisión:

- **Rehasheo Lineal** (Linear probing): Si hay colisión en la ranura i, entonces intentar con la ranura $i + 1$, y así siguiendo hasta encontrar que el elemento (se hace update) o encontrar un lugar vacío (baja física) y se inserta allí. Con esta técnica si hay lugar lo encuentra seguro. Se suele tratar al arreglo como una lista circular.
- **Rehasheo Cuadrático** (quadratic probing): El intervalo entre ranuras a usar, si hubiera colisión, será cuadrático.
- Otra combinación predeterminada (determinística) de fn: Siempre conviene que la última sea rehasheo lineal.

Borrado

No se puede reemplazar el lugar de borrado por una ranura vacía porque la búsqueda de alguna clave puede necesitar "pasar sobre ella" si hubo colisión. Se debe manejar dos tipos de borrado: **físico** (realmente se elimina el elemento) y **lógico** (se lo marca como que no está, y si más tarde hay que insertar en esa ranura se la puede aprovechar).

- El **Borrado Físico** se lo usa cuando la ranura que le sigue (la que se obtiene al aplicar $hash_{i+1}$) está también borrado físicamente.
- El **Borrado Lógico** se lo usa en caso contrario, o sea cuando la ranura que le sigue está ocupada o bien borrada lógicamente.

Búsqueda

Por lo dicho en el punto anterior se comienza buscando la clave en la ranura calculada. Si el lugar está con baja física seguro que no está en otro lado. Caso contrario si está marcado como ocupado y coincide con el valor esperado, se ha encontrado.

Pero si el lugar está ocupado y no es el elemento buscado o bien está como baja lógica, **no se sabe si va a aparecer más adelante** (en la aplicación de las sucesivas funciones de hashing). O sea que en ese caso hay que seguir buscando hasta encontrarlo (hallar una ranura ocupada que coincida con el elemento) o bien hallar una baja física.

Inserción

Si la ranura calculada está marcada como baja física, el elemento se inserta allí. Caso contrario (está ocupado y no es el elemento a insertar, o bien está marcado como baja lógica) hay que comenzar a navegar con las sucesivas celdas **hasta encontrar la primera baja física** (O el error porque el elemento ya existía). Atención que no se puede detenerse en la primera baja lógica y pretende insertarlo allí porque justamente puede estar más adelante. Una vez que se encuentra la primera baja física se lo puede insertar allí o en alguna de las bajas lógicas halladas en ese trayecto.

Tip importante: Para no rebotar de más en las colisiones, lo mejor en el algoritmo de inserción es **no insertar en el primer espacio** libre que se encuentra, sino en la **primera baja lógica** que se encontró en el camino hasta descubrir que se podía insertar (se halló baja física).

ClosedHashing en Java

```
public class ClosedHash<K, V> {
    private final int initialLookupSize = 5;
    private int lookupSize = initialLookupSize;
    // estática. No crece. Espacio suficiente...
    private Node<K, V>[] LookUp = new Node[initialLookupSize];
    private static final double threshold = 0.75;
```

```
    private int uses = 0;

    private final Function<? super K, Integer> prehash;

    public ClosedHash(Function<? super K, Integer> mappingFn) {
        prehash = mappingFn;
    }

    // ajuste al tamaño de la tabla
    private int hash(K key) {
        if (key == null)
            throw new RuntimeException("No key provided");
        return prehash.apply(key) % lookupSize; //LookUp.lenght
    }

    public V getValue(K key) {
        Node<K, V> entry = get(key);
        if (entry == null)
            return null;

        return entry.value;
    }

    private Node<K, V> get(K key) {
        int pos = hash(key);
        while (LookUp[pos] != null) {
            if (LookUp[pos].key == key && LookUp[pos].occupied)
                return LookUp[pos];
            pos = (pos + 1) % lookupSize;
        }
        return null;
    }

    // insert = update
    public void insert(K key, V value) {
        int pos = hash(key);
        int firstEmpty = 0;
        boolean wasEmpty = false;
        while (LookUp[pos] != null) {
            if (LookUp[pos].key == key && LookUp[pos].occupied) {
                LookUp[pos].value = value;
                return;
            }
            if (!LookUp[pos].occupied && !wasEmpty) {
                wasEmpty = true;
                firstEmpty = pos;
            }
            pos = (pos + 1) % lookupSize;
        }
        if (wasEmpty) {
```

```
            LookUp[firstEmpty] = new Node<>(key, value);
        } else {
            LookUp[pos] = new Node<>(key, value);
        }
        uses++;
        if (Double.compare((double) uses / lookupSize, threshold) >= 0) {
            rehash();
        }
    }

    private void rehash() {
        lookupSize = lookupSize * 2;
        Node<K, V>[] aux = LookUp;

        LookUp = new Node[lookupSize];
        for (Node<K, V> kvNode : aux) {
            if (kvNode != null && kvNode.occupied) {
                uses = 0;
                insert(kvNode.key, kvNode.value);
            }
        }
    }

    public void delete(K key) {
        int pos = hash(key);
        while (LookUp[pos] != null) {
            if (LookUp[pos].key == key && LookUp[pos].occupied) {
                if (LookUp[(pos + 1) % lookupSize] != null) {
                    LookUp[pos].occupied = false;
                } else {
                    LookUp[pos] = null;
                    uses--;
                }
                return;
            }
            pos = (pos + 1) % lookupSize;
        }
    }

    public void dump() {
        for (int rec = 0; rec < LookUp.length; rec++)
            if (LookUp[rec] == null)
                System.out.printf("slot %d is empty%n", rec);
            else if (!LookUp[rec].occupied)
                System.out.printf("slot %d is free but had %s previously%n",
rec, LookUp[rec]);
            else
                System.out.printf("slot %d contains %s%n", rec, LookUp[rec]);
    }
```

```
    static class Node<K, V> {
        final K key;
        V value;
        boolean occupied;

        Node(K theKey, V theValue) {
            key = theKey;
            value = theValue;
            occupied = true;
        }


        public String toString() {
            return String.format("key=%s, value=%s", key, value);
        }
    }
}
```

Ventajas y Desventajas

Linear Hashing es muy eficiente para implementar la resolución de colisiones (aprovecha la localidad de la componente ⇒ elementos cercanos). Si hay lugar lo encuentra seguro.

La desventaja se presenta cuando el factor de carga es alto.

Open Hashing o Closed Addressing o Chaining overflow

Las colisiones se resuelven en una **estructura auxiliar** (lista lineal, etc.)

Cada ranura puede tener null, o bien una estructura auxiliar con las componentes que colisionaron en dicha ranura. Si la lista tiene una única componente, entonces no hubo colisión aún en esa ranura.

Borrado

Si la ranura está en null, el elemento no existía. Si hay zona de overflow se lo elimina de la misma. Si la zona de overflow queda vacía, la ranura vuelve a valer null.

Búsqueda

Si la ranura está en null, no está. Si hay zona de overflow, se lo navega allí para ver si se encuentra.

Inserción

Si se le asigna una ranura vacía, se habilita la zona de overflow. Si ya había zona de overflow se lo intenta insertar allí.

En un hashing no hay orden de los elementos. No es razonable pedir que el key sea comparable para que pueda ordenarse. No vamos a usar una lista ordenada simplemente encadenada. sino una lista. Se puede usar la que viene con Java: LinkedList.

Es importante redefinir `equals()`. Caso contrario, la lista no permitirá updates porque siempre considerará que los elementos son diferentes.

Manejar una lista en zona de overflow puede generar una complejidad mayor que 1 en inserción, búsqueda, update y delete.

OpenHashing en Java

```
public class OpenHash<K, V> {

    private final int initialLookupSize = 10;
    private LinkedList<Node<K, V>>[] LookUp = new LinkedList[initialLookupSize];
    private final Function<? super K, Integer> prehash;

    private int usedKeys = 0;
    private final double threshold = 0.75;


    public OpenHash(Function<? super K, Integer> mappingFn) {
        prehash = mappingFn;
    }

    // ajuste al tamaño de la tabla
    private int hash(K key) {
        if (key == null)
            throw new RuntimeException("No key provided");
        //System.out.println("Hash: " + (prehash.apply(key) % LookUp.length));
        return prehash.apply(key) % LookUp.length;
    }

    public V getValue(K key) {
        Node<K, V> entry = get(key);
        if (entry == null)
            return null;

        return entry.value;
    }

    private Node<K, V> get(K key) {
        LinkedList<Node<K, V>> list = LookUp[hash(key)];
        if(list != null) {
            for(Node<K, V> node : list) {
                if(node.key.equals(key)) {
                    return node;
                }
            }
```

```
        }
        return null;
    }

    // insert = update
    public void insert(K key, V value) {
        int hashKey = hash(key);
        if(LookUp[hashKey] == null) {
            LookUp[hashKey] = new LinkedList<Node<K, V>>();
        }

        Node<K, V> aux = get(key);
        if(aux != null) {
            aux.value = value;
        } else {
            LookUp[hashKey].add(new Node<K ,V>(key, value));
            usedKeys++;
        }

        if(getCurrentLoadFactor() > threshold) {
            rehash();
        }
    }


    public void delete(K key) {
        int hashKey = hash(key);
        Node<K, V> node = get(key);
        if(node != null) {
            LookUp[hashKey].remove(node);
            usedKeys--;
        }
        if(LookUp[hashKey].isEmpty()) {
            LookUp[hashKey] = null;
        }
    }

    public void dump() {
        for (int rec = 0; rec < LookUp.length; rec++)
            if (LookUp[rec] == null)
                System.out.printf("slot %d is empty\n", rec);
            else
                System.out.printf("slot %d contains %s\n", rec, LookUp[rec]);
    }

    private void rehash() {
        LinkedList<Node<K, V>>[] prevLookUp = LookUp;
        LookUp = new LinkedList[LookUp.length * 2];
        int prevUsedKey = usedKeys;
```

```
        for(List<Node<K, V>> list : prevLookUp) {
            if(list != null) {
                usedKeys = 0; //Evita el rehash
                for(Node<K,V> node : list) {
                    insert(node.key, node.value);
                }
            }
        }

        usedKeys = prevUsedKey;
    }

    private double getCurrentLoadFactor() {
        return ((double)usedKeys)/LookUp.length;
    }

    static class Node<K, V> {
        final K key;
        V value;

        Node(K theKey, V theValue) {
            key = theKey;
            value = theValue;
        }

        @Override
        public String toString() {
            return String.format("key = %s, value = %s", key, value);
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            Node<?, ?> node = (Node<?, ?>) o;
            return this.key.equals(node.key) &&
                    this.value.equals(node.value);
        }

        @Override
        public int hashCode() {
            return Objects.hash(key, value);
        }
    }
}
```

Bag - Implementación usando Hashing

Un Bag es una colección que permite elementos repetidos en ella (no los ignora ni los rechaza). Está diseñada para devolver rápidamente cuántas ocurrencias tiene cierto elemento en la misma.

Los elementos de un Bag no son *key/value* sino *elementos*.

Java no tiene implementada la clase Bag, pero Apache Commons Collections sí.

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-collections4
-->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-collections4</artifactId>
    <version>4.0</version>
</dependency>
```

Bag en Java

Los métodos de Bag serán los siguientes:

- `int getCount(T value)`: Si no existe devuelve 0, caso contrario, la cantidad de ocurrencias.
- `void add(T value)`: Agrega una ocurrencia.
- `void remove(T value)`: Elimina una ocurrencia.

Hashing de Java (O sea la que viene con Java, viste)

Hashing es una colección que viene implementada en Java.

Hashing - Java - Documentación

Al igual que lo hicimos más arriba, la versión de Java:

- Usa un arreglo de ranuras con zona de overflow:

```
transient Node<K,V>[] table;
```

- Manejo del espacio inicial y factor de carga:

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

- Si se acaba el espacio, lo duplica:

```
++modCount;
if (++size > threshold)
    resize();
```

- Tiene manejo de overflow. Si no hay mucha ocupación en una ranura, entonces usa una lista:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
```

Pero si hay demasiados elementos, los transforma en un [Red Black Tree]:

```
/**
 * Replaces all linked nodes in bin at index for given hash unless
 * table is too small, in which case resizes instead.
 */
final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}
```

Árboles (Falta colocar implementaciones)

Árbol Binario (Binary Tree o BT)

Un **Árbol Binario** es una estructura de datos formada por nodos, donde cada nodo o está vacío o tiene 3 componentes: datos, subárbol izquierdo y subárbol derecho. Existe un nodo distinguido llamado **raíz**.

Expresiones

Los compiladores interpretan expresiones formadas por constantes, variables y sub expresiones.

Una forma de evaluar una expresión consiste en representarla como una estructura jerárquica, esto es, un **árbol de expresiones**.

Árbol Binario de Expresiones

Un **Árbol Binario de Expresiones** se usa para representar expresiones algebraicas. Los nodos internos representan operadores binarios y unarios. Las hojas representan los operandos, es decir, constantes y variables.

Según como se recorra el árbol (transversal) in-order, pre-order o post-order, se obtiene una expresión infija, prefija o postfija respectivamente y se obtiene la correspondiente expresión.

Características

Permite representar expresiones en notación infija (obviamente no ordenado).

Así como usábamos una pila y una tabla de precedencia de operadores para pasar de una expresión en notación infija a postfija (para eliminar ambigüedad) y luego con una pila evaluamos la expresión, ahora también a partir de una **expresión**, por ejemplo infija, **construiremos el árbol de expresiones asociado** y lo evaluaremos para devolver el valor de la expresión.

Aclaración: Para evitar la discusión sobre la precedencia de operadores, vamos a aceptar solo expresiones infijas que tengan paréntesis. Los operadores son todos binarios y es obligatorio usar paréntesis para toda la expresión.

Para el input vamos a pedir que finalice en `'\n'`. Los espacios serán los separadores de tokens.

Reglas de Derivación de Expresiones

Formalmente, una expresión aritmética E está dada por las siguientes reglas de derivación:

E -> (E + E)
E -> (E – E)
E -> (E * E)
E -> (E / E)
E -> (E ^ E)
E -> cte

E -> (E op E)
E -> cte

Con op: + - * / ^

ExpTree en Java

```
public class ExpTree {

    private Node root;
    private final Map<String, Double> variables = new HashMap<>();

    public ExpTree(String infija, Map<String, Double> variables) {
        // token analyzer
        Scanner inputScanner = new Scanner(infija).useDelimiter("\\n");
        String line = inputScanner.nextLine();
        inputScanner.close();

        buildTree(line);
    }


    public ExpTree() {
        System.out.print("Introduzca la expresión en notación infija con todos
los parentesis y blancos: ");

        // token analyzer
        Scanner inputScanner = new Scanner(System.in).useDelimiter("\\n");
        String line = inputScanner.nextLine();
        inputScanner.close();

        buildTree(line);
    }

    private void buildTree(String line) {
        // space separator among tokens
        Scanner lineScanner = new Scanner(line).useDelimiter("\\s+");
        root = new Node(lineScanner);
        lineScanner.close();
```

```
    }

    public String preOrder() {
        return root.preOrder();
    }

    public String postOrder() {
        return root.postOrder();
    }

    public String inOrder() {
        return root.inOrder();
    }

    public double evaluate() {
       return evaluateRec(root);
       }

    private double evaluateRec(Node node) {
        if(node.isLeaf()) {
            if(node.data.matches("[A-Za-z][A-Za-z0-9]+")) {
                return variables.getOrDefault(node.data, 0.0);
            }
            return Double.parseDouble(node.data);
        }

        switch(node.data) {
            case "+": return evaluateRec(node.left) + evaluateRec(node.right);
            case "-": return evaluateRec(node.left) - evaluateRec(node.right);
            case "*": return evaluateRec(node.left) * evaluateRec(node.right);
            case "/": return evaluateRec(node.left) / evaluateRec(node.right);
            default: return 0;
        }
    }

    //add = update
    public void addVariable(String name, double value) {
        variables.put(name, value);
    }

    /**
     * Node
     */
    static final class Node {
        private final String data;
        private final Node left, right;

        private Scanner lineScanner;

        public Node(Scanner theLineScanner) {
```

```
            lineScanner = theLineScanner;

            Node aux = buildExpression();
            data = aux.data;
            left = aux.left;
            right = aux.right;

            if (lineScanner.hasNext())
                throw new RuntimeException("Bad expression");
        }

        private Node(Node left, Node right, String data) {
            this.left = left;
            this.right = right;
            this.data = data;
        }

        private Node buildExpression() {
            if (!lineScanner.hasNext()) {
                throw new RuntimeException("Missing expression");
            }
            if (!lineScanner.hasNext("\\(")) {
                String cte = lineScanner.next();
//                  if (!isConstant(cte))
//                      throw new RuntimeException("Invalid term");
                return new Node(null, null, cte);
            }
            lineScanner.next();
            Node leftNode = buildExpression();
            String op = lineScanner.next();
            if (!isOperator(op))
                throw new RuntimeException("Missing operator");
            Node rightNode = buildExpression();
            if (!lineScanner.hasNext("\\)")) {
                throw new RuntimeException("Bad expression");
            }
            lineScanner.next();
            return new Node(leftNode, rightNode, op);
        }

        private boolean isOperator(String c) {
            return c.equals("+") || c.equals("-")
                    || c.equals("*") || c.equals("/")
                    || c.equals("^");
        }

        private boolean isConstant(String string) {
            boolean numeric = true;
            try {
                Double.parseDouble(string);
```

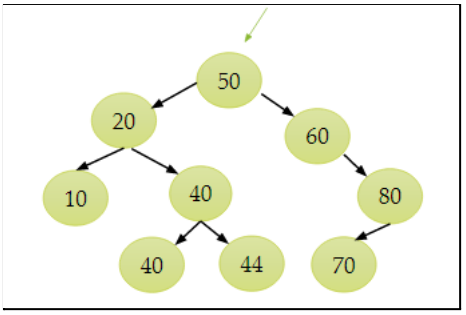
```
            } catch (NumberFormatException e) {
                numeric = false;
            }
            return numeric;
        }

        private boolean isLeaf() {
            return this.left == null && this.right == null;
        }

        private String preOrder() {
            return String.format("%s%s%s", data, left == null ? "" : (" " +
left.preOrder()), right == null ? "" : (" " + right.preOrder()));
        }

        private String postOrder() {
            return String.format("%s%s%s", left == null ? "" : (left.postOrder()
+ " "), right == null ? "" : (right.postOrder() + " "), data);
        }

        private String inOrder() {
           return String.format("%s%s%s", left == null ? "" : ("( " +
left.inOrder()), (data + " "), right == null ? "" : (right.inOrder()) + ") ");
        }

    }  // end Node class

}  // end ExpTree
```

Árbol Binario de Búsqueda (BST)

Los usos de árboles son múltiples. Además de los árboles de expresiones, usar una estructura de árbol ordenada para buscar elementos suena interesante:

- De la lista toma lo mejor: Encadenar los elementos con punteros y no tener que alocar zona contigua.
- De los arreglos ordenados toma lo mejor: La posibilidad de aplicar búsqueda binaria.

Un **Árbol Binario de Búsqueda** (**Binary Search Tree** o **BST**), es un árbol binario donde cada nodo no vacío cumple la siguiente condición: todos los datos de su **subárbol izquierdo** son menores o iguales que su dato, y todos los datos de su **subárbol derecho** son mayores que su dato.



La **altura** de un BST está dada por la longitud del camino más largo desde la raíz hacia las hojas. Un nodo formado por una raíz tiene altura 0.

Operaciones sobre un BST

Insertar: Un BST crece desde las hojas.

Borrar: No basta con eliminar un elemento, se debe mantener la forma del original (No deformarse). El algoritmo es el siguiente:

- Si el nodo a eliminar es *hoja*, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que ya no lo apunte más a él y pase a apuntar a null.
- Si el nodo a eliminar tiene *un solo hijo*, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que en vez de apuntarlo a él lo haga al hijo del que se borra.
- Si el nodo a eliminar tiene *dos hijos* se procede en dos pasos: Primero se lo reemplaza por un nodo ***lexicográficamente adyacente*** (su **predecesor inorder**, o sea el más grande de los nodos de su subárbol izquierdo, o bien su **sucesor inorder**, o sea el más chico de los nodos de su subárbol derecho), y finalmente se borra al nodo que lo reemplazó (seguro que dicho nodo tiene a lo sumo un solo hijo, sino no sería lexicográficamente adyacente, y por lo tanto es fácil de borrar).

BST en Java

Árbol Binario Perfectamente Balanceado

Un **Árbol Binario Perfectamente Balanceado** es un árbol binario donde el único nivel que puede no estar completo es el último y ese último nivel debe estar completo de izquierda a derecha.

Árbol Binario Balanceado por Altura (AVL)

En 1962, G. M. **A**delson-**V**elskii y E. M. **L**andis propusieron la primera versión de BST que se balancea dinámicamente.

Un **AVL** es un BST donde en cada nodo la diferencia de alturas entre sus 2 subárboles es a los sumo 1.

Un AVL es un árbol con buena forma. Las inserciones y borrados en un AVL están definidas de tal manera que garantizan que se puede realizar en $O(logn)$ y garantizan ser invariantes ante su propiedad.

Factor de Balance o Equilibrio

El **Factor de Balance** es la diferencia entre las alturas del árbol derecho y el izquierdo:

$$Fb = altura\ subárbol\ derecho - altura\ subárbol\ izquierdo$$

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

Si el factor de equilibrio de un nodo es:

- 0: El nodo está equilibrado y sus subárboles tienen exactamente la misma altura.
- 1: El nodo está equilibrado y su subárbol derecho es un nivel más alto.
- -1: El nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

Si el factor de balance $|Fb| \geq 2$ es necesario reequilibrar.

Operaciones sobre un AVL

Inserción:

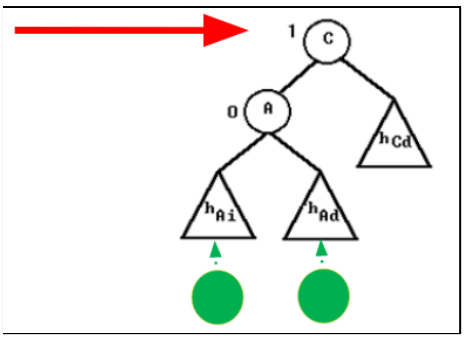
- Se inserta en el BST.
- Si está desbalanceado se aplican rotaciones para que siga siendo AVL. Se rota el árbol con pivote más joven desbalanceado (más cercano al nodo insertado). Más precisamente:
    - Caso 1: Si se inserta a la izquierda de un nodo con factor de balance 1, ese nodo va a tener factor de balance 2 y deja de ser AVL. Hay que rotar.
    - Caso 2: Si se inserta a la derecha de un nodo con factor de balance -1, ese nodo va a tener factor de balance -2 y deja de ser un AVL. Hay que rotar.

Rotación: Transforma un árbol binario en otro, de tal manera que preserva el orden de sus elementos al navegarlo en inorder. Existen rotaciones simples y dobles.

Derecha: Horario.

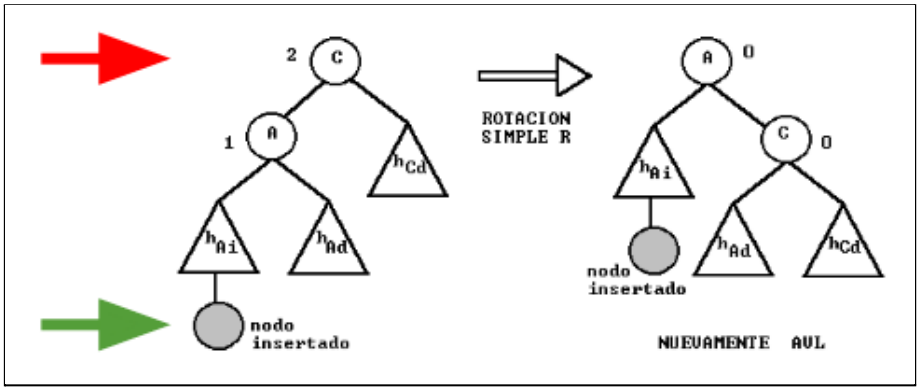
Izquierda: Antihorario.

- **Caso 1**: Insertar a izquierda de un nodo con fb 1. Este nodo tendría fb 2 y dejaría de ser AVL. Tiene dos subcasos.



- Rotación **simple a derecha** (R): Cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol izquierdo.

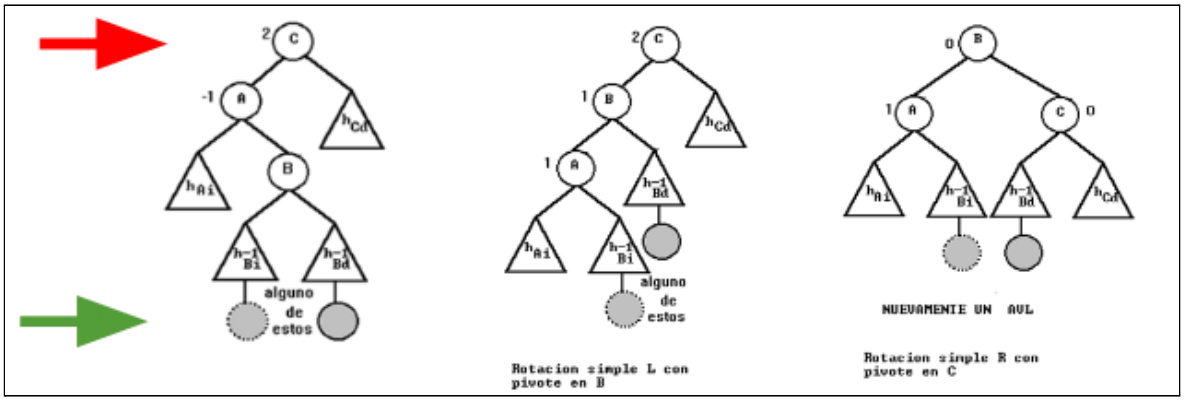
  Rotación hacia la derecha con pivote en el nodo más joven inicialmente desbalanceado con 2, o sea “C”.



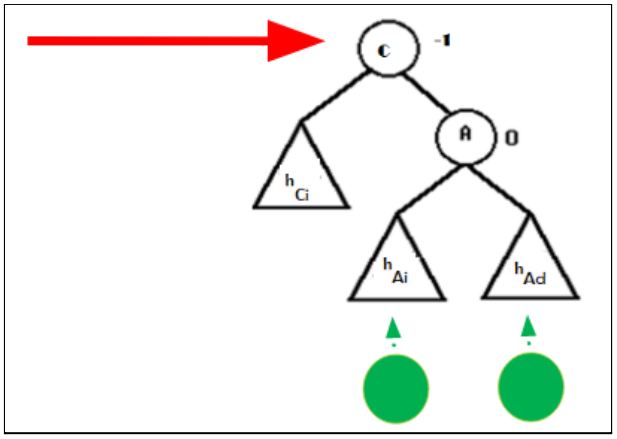
- Rotación **doble izquierda a derecha** (LR): Cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol derecho.

  Primera rotación hacia la izquierda con el pivote en el hijo del nodo desbalanceado (del lado de la inserción), en nuestro caso “A”.

  Segunda rotación hacia la derecha con el pivote en nodo más joven inicialmente desbalanceado con 2, o sea “C”.



- **Caso 2**: Insertar a derecha de un nodo con fb -1. Este nodo tendría fb -2 y dejaría de ser AVL. Tiene dos subcasos.



- Rotación **simple a izquierda** (L): Cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol derecho.

  Rotación hacia la izquierda con pivote en el nodo más joven inicialmente desbalanceado con -2.

- Rotación **doble derecha a izquierda** (RL): Cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol izquierdo.

  Primera rotación hacia la derecha con el pivote en el hijo del nodo desbalanceado (del lado de la inserción).

  Segunda rotación hacia la izquierda con el pivote en nodo más joven inicialmente desbalanceado con -2.

TestAVL en Java

```
@Override
public boolean testAVL() {
    if(root == null) {
        return false;
    }
    return testAVL(root);
}

private boolean testAVL(Node<T> node) {
    if(node == null) {
        return true;
    }
    return Math.abs(getHeightRec(node.left) - getHeightRec(node.right)) <= 1
        && testAVL(node.left)
        && testAVL(node.right);
}
```

Árbol de Fibonacci

El **Árbol de Fibonacci** es el AVL que presenta el peor desbalance posible. Se define de la siguiente manera:

1. Fibonacci de orden **0** es el árbol nulo.
2. Fibonacci de orden **1** es un nodo.
3. Fibonacci de orden **h** es un árbol que tiene:
    a. Como hijo izquierdo un Fibonacci de orden $h - 1$.
    b. Como hijo derecho un Fibonacci de orden $h - 2$.

Sea un AVL de altura h con la menor cantidad de nodos posibles en esa altura. Este AVL de altura h tendrá la siguiente cantidad de nodos:

$$1\,nodo \;+\; (cantNodos\ AVL\ altura\ h - 1) \;+\; (cantNodo\ AVL\ altura\ h - 2)$$

AVL altura h con menos cantidad de nodos	**Cant de nodos en relación a los números de Fibonacci**
$H = 0$ Cant de Nodos = 1	$Fibo(H + 3) - 1 = Fibo(3) - 1 = 1$
$H = 1$ Cant de Nodos = 2	$Fibo(H + 3) - 1 = Fibo(4) - 1 = 2$
$H = 2$ Cant de Nodos = $1 + 1 + 2 = 4$	$Fibo(H + 3) - 1 = Fibo(5) - 1 = 4$
$H = 3$ Cant de Nodos = $1 + 4 + 2 = 7$	$Fibo(H + 3) - 1 = Fibo(6) - 1 = 7$
$H = 4$ Cant de Nodos = $1 + 7 + 4 = 12$	$Fibo(H + 3) - 1 = Fibo(7) - 1 = 12$
$H = 5$ Cant de Nodos = $1 + 12 + 7 = 20$	$Fibo(H + 3) - 1 = Fibo(8) - 1 = 20$
…	
$H = h$ Cant de Nodos = ?	$Fibo(h + 3) - 1$

Generador de números de Fibonacci (numberempire.com)

Así, sea un AVL con altura h, su **cantidad de nodos** es $fibo(h + 3) - 1$. Ese es el AVL que menos nodos tendrá. Otros AVL tendrán un número n de nodos, donde $n \geq fibo(h + 3) - 1$.

Además, se sabe que $fibo(w) \geq \frac{a^w}{\sqrt{5}}$. Donde a es el "golden number", $a = \frac{1+\sqrt{5}}{2} \simeq 1.618$.

O sea, $n \geq fibo(h + 3) - 1$, entonces:

$$n \geq \frac{a^{h+3}}{\sqrt{5}} - 1$$

h es $O(log\ n)$, o sea, para un AVL con n nodos, la altura está acotada por $O(log\ n)$.

Hay diferentes algoritmos para garantizar que un árbol sea balanceado y esta propiedad sea invariante ante inserciones/borrados.

Las transformaciones que hay que hacer para mantenerlo apropiado llegan en general $O(log\ n)$.

Red Black Tree

Los **Red Black Tree** son árboles balanceados que siguen las siguientes reglas:

- Todo los nodos son **negros** o **rojos**.
- La raíz es **negra**, al igual que los nil o null.
- Si un nodo es **rojo**, no puede tener hijos **rojos**.
- Todo camino tienen que tener la misma cantidad de nodos **negros**.

Página útil para la visualización de un red black tree.

Algoritmo de Inserción

Hay cuatro casos posibles para la inserción:

- Si el nodo a insertar es la raíz → se pinta de **negro**.
- Si el tío del nodo a insertar es **rojo** → se re-colorea el padre, el abuelo y el tío.
- Si el tío del nodo a insertar es **negro** (forma un triangulo) → se rota el padre del nodo a insertar.
- Si el tío del nodo a insertar es **negro** (forma un linea) → se rota el abuelo del nodo a insertar y re-colorear padre y abuelo originales.

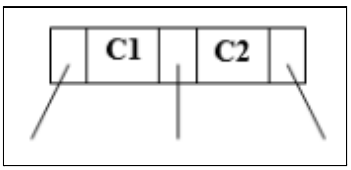
Complejidad temporal: *O(log(n))*.

Árboles Multicamino

Árbol Multicamino M-ario (orden M)

En un **Árbol Multicamino M-ario**, los nodos guardan hasta $M - 1$ claves de información, con un máximo de M hijos. Cada clave C_i de un cierto nodo será tal que las claves almacenadas en su subárbol izquierdo serán menores y las almacenadas en su subárbol derecho serán mayores que él.

Por ejemplo, un árbol multicamino con M=3, podría ser:



Este nodo tiene dos claves $C_1 < C_2$ y además todas las claves del subárbol de C_1 serán menores que él y las de su subárbol derecho (que es el mismo que el izquierdo de C_2) serán mayores que él. Análogamente ocurre para C_2.

Arboles Multicaminos Balanceados

El equilibrio perfecto resulta muy costoso de mantener y es poco práctico.

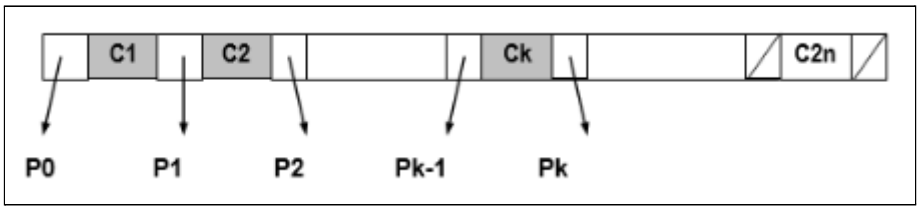
En 1970, R. Bayer y E. M. Mc Creight postularon un criterio razonable que permite implementar algoritmos relativamente sencillos para búsquedas, inserciones y eliminaciones. Para mantener estos árboles multicaminos balanceados se utiliza una estructura subyacente de la familia de los árboles B.

Árbol B de Orden N

Un **Árbol B de Orden N** es aquel que cumple los siguiente axiomas:

- Cada nodo contiene a lo sumo $2 * N$ claves.
- Cada nodo, excepto la raíz, contiene por lo menos *N* claves.
- Cada nodo o es hoja o tiene $M + 1$ descendientes donde M es el número de claves que posee realmente ese nodo (lugares ocupados).
- Todas las hojas están al mismo nivel.

Un **nodo** genérico de un árbol B de orden N será de la forma:



En este caso el nodo posee realmente k claves (presentes con información) y por lo tanto $k + 1$ punteros (los demás son nulos).

Página útil para la visualización de árboles B.

Algoritmo de Búsqueda

Buscamos la clave X en un nodo. Para ello lo recorremos secuencialmente desde C_1 hasta C_k, siendo k el número de claves que realmente posee dicho nodo, hasta que se den alguno de estos casos:

- Si $X < C_1$, como en el nodo las claves están ordenadas no tiene sentido seguir buscando en ese nodo, luego sigo buscando en el subárbol apuntado por P_0.
- Si $X = C_i$ para algún $i \leq k$ entonces lo encontré.
- Si $C_i < X < C_{i+1}$ para algún i prosigo en la búsqueda en el subárbol apuntado por P_i.
- Si $C_k < X$ siendo k la cantidad de claves que posee, entonces sigo la búsqueda en el subárbol apuntado por C_k.

Si en algún caso el puntero por donde hay que seguir la búsqueda fuera null, entonces el elemento buscado no está.

Algoritmo de Inserción

Si se quiere insertar una clave X en un árbol B de orden N, se procede de la siguiente manera:

- La inserción siempre se hace **en las hojas** (para poder detectar si el nodo a insertar ya está presente o no).
- Para insertar se coloca el elemento X en la hoja que corresponda (el nodo debe estar ordenado).
- Si el elemento nuevo hace que la cantidad nueva k sea mayor que el $2 * N$ permitido, el nodo se abre en dos, subiendo la clave del medio al nodo antecesor de dicho nodo. Este algoritmo es recursivo hasta la raíz, o sea si al ubicar la clave del medio en el nodo antecesor ocasiona que el nodo viole la condición de árbol B de orden N ($k > 2 * N$) el procedimiento se repite.

Operación de Borrado

- Si no se encuentra en un nodo hoja, se lo reemplaza por una clave lexicográficamente adyacente, por ejemplo el sucesor in order y se lo elimina de dicha hoja. Si fuera hoja se lo elimina directamente.
- Luego, para la hoja que colaboró en el borrado se analiza si cumple las condiciones de árbol B de orden N. Si ha quedado en rojo (tiene menos elementos que los permitidos), se une dicho nodo con su hermano y medio antecesor (el cual es eliminado del nodo al cual pertenece, porque acude en ayuda de su hijo) armando un sólo nodo. Se verifica si cumple las condiciones de árbol B, y si no se lo particiona

subiendo el elemento del medio, Después se analiza qué sucede con el nodo donde estaba su medio antecesor, y se sigue el proceso recurrentemente hasta llegar a la raíz.

Heurísticas

Muchos juegos se pueden resolver explorando un grafo implícito. Es decir, se explora un espacio de soluciones generadas implícitamente por un grafo.

Ese grafo representa en sus nodos una solución parcial del problema. El nodo raíz representa la configuración inicial del problema. Hay un eje desde un nodo n_1 hacia n_2 ($n_1 \rightarrow n_2$) si desde n_1 se puede ir a n_2 aplicando una regla de juego (que lleve a un estado válido).

Técnica de Búsqueda Exhaustiva

Técnica para buscar todas las posibles soluciones explorando el espacio de soluciones en forma implícita. Sirve para problemas que tienen muchas soluciones.

El grafo es implícito (no lo genero). Típicamente se implementa con recursión (o sea Stack) aunque puede hacerse con una Queue. Ej: DFS, BFS.

Idea para esta técnica (Típicamente recursiva):

1. Si el nodo no puede expandirse más (no hay más opciones a partir de él), entonces retornar/imprimir el resultado.
2. Si no, por cada posibilidad para ese nodo de expandir un próximo nivel (ciclo for):
    a. El nodo puede resolver un caso pendiente.
    b. Explorar nuevos pendientes (soluciones quizás parciales).
    c. El nodo puede deshacer/quitar el caso pendiente generado.

SolveHelper en Java

Técnicas Backtracking

Ante la presencia de restricciones, pueden aprovecharlas para no explorar todo el grafo de soluciones y podar aquellos nodos que no conducen a la solución.

Ahí es donde las **Técnicas de Backtracking** entran en juego: no expande innecesariamente nodos que ya se saben (gracias a la restricción) no conducirán a la solución.

¿Cómo puedo evitar no expandir más un nodo? Un nodo intermedio ya lleva acumulados valores. En el mejor de los escenarios completará con números bajos, es decir "1" en lo que falta. Si esa suma de valores actuales junto con $faltantes * 1$ supera el umbral, imposible seguir.

Backtracking en Java

Backtracking con Programación Dinámica en Java

Agreguemos la suma como parámetro.

Divide and Conquer

Divide and Conquer es una técnica que descompone un problema de tamaño N en problemas más pequeños que tengan solución. Finalmente se debe proponer cómo construir la solución final a partir de las soluciones de los problemas menores. Ejemplo: Mergesort, Quicksort, búsqueda en un BST, búsqueda en un arreglo ordenado.

Algoritmos Ávidos (Greedy)

Greedy es una técnica que busca en cada etapa un óptimo local con el objetivo de llegar al óptimo global (aunque de esta forma no siempre se consigue el óptimo global). Ejemplo: algoritmo de Kruskal para encontrar el mínimo árbol generador de un grafo.

Fuerza Bruta/Búsqueda Exhaustiva (con Stack o Queue)

Calcula todas las posibles soluciones. Si se agregan restricciones, recién en el final se evalúa cual es la mejor. Es decir, en las hojas se evalúa si se satisface la restricción pedida.