72.34 - Estructura de Datos y Algoritmos

Carpeta teórica

Sandrini, Santiago

1C - 2021

1.Algoritmos para texto

Alfabeto $\sum$: conjunto de símbolos o caracteres.

Dado un alfabeto $\sum$ y $k \geq 0$, un string S es un elemento $k \in \sum^k$

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

Dado un alfabeto, $\sum^*$ es el conjunto de todos los strings posibles sobre el alfabeto $\sum$

Prefijos, sufijos y bordes

Dados un alfabeto y los strings x, w, z. 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

1.1) BÚSQUEDA EXACTA

Al momento de implementar un buscador, suelen usarse diferentes estrategias para encontrar un resultado en el caso de que la búsqueda lanzada no sea reconocida en el corpus que poseen sobre búsquedas. Estas estrategias pueden ser:

- Tener en cuenta el idioma del sistema
- Errores de tipeo del estilo “height - heigth”
- Etc

1.1.1 Algoritmo SOUNDEX (1918)

Soundex es un algoritmo fonético (el más conocido), un algoritmo para indexar un nombre por su sonido, al ser pronunciados en Inglés. El objetivo básico de este algoritmo es codificar de la misma forma los nombres con la misma pronunciación.

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

El código Soundex para un nombre consiste en una letra seguida de tres números: la letra es la primera letra del nombre, y el número codifica el resto de consonantes. Veamos el procedimiento con un ejemplo. Tomemos S = PHONE.

En el primer paso del algoritmo, se toma la primera letra del string para empezar el código. Luego, se compara el código del siguiente caracter con el anterior, en el caso de que sean iguales o el código del caracter actual sea cero, no se escribe. En este caso quedaría PHONE = P500. (se completa con ceros hasta formar los 3 dígitos).

Finalmente, para comparar dos strings se compara caracter a caracter. Notemos que PHONE y FOUN tienen un 0,75 de coincidencia.

Implementación en JAVA:

```java
public static String encode (String input) {

    input.toUpperCase();
    char [] ans = {'0','0','0','0'};
    ans[0] = input.charAt(0);

    char lastCode = data[ input.charAt(0) - 65 ];
    int index = 1;
    char current;

    for(int i = 1; i < input.length() && index < 4; i++, lastCode = current) {
        current = data [ input.charAt(i) - 65 ];
        if( current != '0' && current != lastCode ) {
            ans[index++] = current;
        }
    }

    return new String(ans);
}
```

(data es un arreglo de 26 posiciones con los códigos de cada letra respectivamente)

RESULTADOS

Strings	Encode
Threshold	T624
Hold	H430
Zresjoulding	Z624
Phone	P500
Foun	F500

1.1.2 Algoritmo Metaphone (1990)

Metaphone fue desarrollado por Lawrence Philips como respuesta a las deficiencias del algoritmo Soundex. Mejora fundamentalmente el algoritmo Soundex mediante el uso de información sobre variaciones e inconsistencias en la ortografía y pronunciación en inglés para producir una codificación más precisa, que hace un mejor trabajo al hacer coincidir palabras y nombres que suenan similares. Al igual que con Soundex, las palabras con un sonido similar deben compartir las mismas claves. Es más exacto que Soundex porque "entiende" las reglas básicas de pronunciación en inglés.

Los códigos originales de Metaphone usan los 16 símbolos consonantes 0BFHJKLMNPRSTWXY. El '0' representa "TH" , 'X' representa "SH" o "CH", y los demás representan sus pronunciaciones habituales en inglés. También se utilizan las vocales AEIOU, pero solo al principio del código. Ejemplos de las reglas (son 19) :

1. Elimine las letras adyacentes duplicadas, excepto C.
2. Si la palabra comienza con 'KN', 'GN', 'PN', 'AE', 'WR', elimine la primera letra.
3. Elimine 'B' si ocurre después de una 'M' al final de la palabra. (“lamb” -> “lam”)
4. 'CK’ se transforma a 'K'.
5. 'PH' se transforma a 'F'.
6. 'V' se transforma a 'F'.
7. Etc

1.1.3 Distancia de Levenshtein (1965)

La distancia de Levenshtein, distancia de edición o distancia entre palabras es el número mínimo de operaciones requeridas para transformar una cadena de caracteres en otra. Se entiende por operación, bien una inserción, eliminación o la sustitución de un carácter. Esta distancia recibe ese nombre en honor al científico ruso Vladimir Levenshtein, quien se ocupó de esta distancia en 1965. Es útil en programas que determinan cuán similares son dos cadenas de caracteres, como es el caso de los correctores ortográficos.

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

	~	B	I	G		D	A	T	A
~	0	1	2	3	4	5	6	7	8
B	1	0	1	2	3	4	5	6	7
I	2	1	0	1	2	3	4	5	6
G	3	2	1	0	1	2	3	4	5
D	4	3	2	1	1	1	2	3	4
A	5	4	3	2	2	2	1	2	3
T	6	5	4	3	3	3	2	1	2
A	7	6	5	4	4	4	3	2	**1**

Implementación en JAVA:

```java
public static int distanceBetween(String s1, String s2) {
    int l1 = s1.length();
    int l2 = s2.length();
    int m[][] = new int[l1][l2];
    m[0][0] = 0;

    for(int i = 0; i < s1.length(); i++) {
        for(int j = 0; j < s2.length(); j++) {
            if(min(i,j) == 0) {
                m[i][j] = max(i, j);
            }
            else {
                int aux = m[i-1][j-1];
                if(s1.charAt(i) != s2.charAt(j)) {
                    aux++;
                }
                m[i][j] = min( a: m[i - 1][j] + 1,  b: m[i][j - 1] + 1, aux);
            }
        }
    }
    return m[l1-1][l2-1];
}
```

Notar que la complejidad temporal con esta implementación es O(N.M) y lo mismo con la complejidad espacial, aunque ésta última podría reducirse (sólo se necesitan las filas y columnas inmediatamente anteriores a la actual). Pero esta implementación permite además saber cuáles son las operaciones (borrado, inserción o sustitución) que se deben realizar para pasar de una string a la otra.

Distancia normalizada: Dist = 1 - (Levensthein(str1, str2) / max(str1.len , str2.len))

1.1.4 Damerau - Levenshtein (y otras variantes)

Es una variante del algoritmo de Levenshtein que agrega la operación de transposición de letras (por ejemplo cambiar de TH a HT con una única operación) dado que es un error frecuente al escribir. Observación: consistiría en tener en cuenta el valor de la diagonal a dos de distancia + 1.

Existen además otras variantes que ponderan los pesos de las diferentes operaciones debido a que algunas son más frecuentes que otras al equivocarse escribiendo una string.

1.1.5 Q-gramas

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 bigramas, si Q es 3 se generan tri-gramas.

Por ejemplo, para el string “JOHN” si se quiere generar hasta trigramas (Q <= 3), puede completarse al comienzo y al final con Q-1 símbolos especiales (que no pertenezcan al alfabeto) y deslizando la ventana imaginaria de tamaño Q, se va generando los Q-gramas. Sea ‘##JOHN##’:

QGr (John) = {‘J’, ‘O’, ‘H’, ‘N’, ‘#J’, ‘JO’, ‘OH’, ‘HN’, ‘N#’, ‘##J’, ‘#JO’, ‘JOH’, ‘OHN’, ‘HN#’, ‘N##’}

Observación: El caso en el que mejor funciona es por ejemplo al comparar apellido-nombre con nombre-apellido, y los algoritmos fonéticos fallarían.

IMPLEMENTACIÓN EN JAVA:

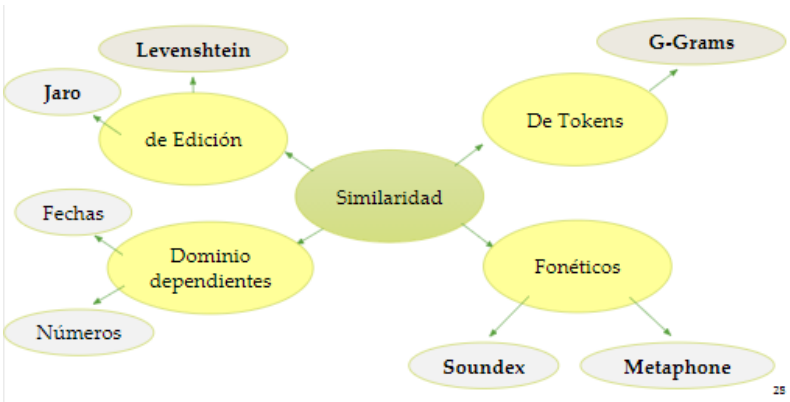
```
private void mapCreation(String str, Map<String, Integer> map2) {
    for (int j = 0; j<str.length() && (j+n)<=str.length() ;j++){
        String aux2 = str.substring(j,j+n);
        if(map2.containsKey(aux2))
            map2.replace(aux2,map2.get(aux2)+1);
        else
            map2.put(aux2,1);
    }
}

public  int size(Map<String,Integer> map){
    int cont=0;
    for (String aux: map.keySet()) {
        cont = cont + map.get(aux);
    }
    return cont;
}
```

```java
public  double similarity(String word1, String word2){
    String palabra1 = Text(word1);
    String palabra2 = Text(word2);
    Map<String,Integer> map1 = new HashMap<>();
    Map<String,Integer> map2 = new HashMap<>();
    mapCreation(palabra1, map1);
    mapCreation(palabra2, map2);
    int min;
    for (String aux: map1.keySet() ) {
        for (String aux2: map2.keySet()) {
            if(aux.equals(aux2) ){
                min = Math.min(map2.get(aux2),map1.get(aux));
                map1.replace(aux,map1.get(aux)-min);
                map2.replace(aux2,map2.get(aux2)-min);
            }
        }
    }
    int cont = 0;
    for (String key1: map1.keySet()) {
        cont = cont + map1.get(key1);
    }
    for (String key2: map2.keySet()) {
        cont = cont + map2.get(key2);
    }
    int i = palabra1.length() - 1 + palabra2.length() -1 ;
    double aux = (i)-cont;
    double aux2 = i;
    return aux/aux2;
}
```

La **distancia normalizada** se calcula con la sig. fórmula:

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

Resumen



1.2) BÚSQUEDA EXACTA

1.2.1 Fuerza bruta

En informática, la búsqueda por fuerza bruta, es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo. La búsqueda por fuerza bruta es sencilla de implementar y, siempre que exista, encuentra una solución. Sin embargo, su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente proporcional al tamaño del problema. Por el contrario, la búsqueda por fuerza bruta se usa habitualmente cuando el número de soluciones candidatas no es elevado, o bien cuando este puede reducirse previamente usando algún otro método heurístico.

Es un método utilizado también cuando es más importante una implementación sencilla que una mayor rapidez. Este puede ser el caso en aplicaciones críticas donde cualquier error en el algoritmo puede acarrear serias consecuencias; también es útil como método "base" cuando se desea comparar el desempeño de otros algoritmos metaheurísticos. La búsqueda de fuerza bruta puede ser vista como el método metaheurístico más simple.

IMPLEMENTACIÓN EN JAVA:

```java
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++;
        }
        else  {
            idxTarget = (idxTarget - idxQuery) + 1;
            idxQuery = 0;
        }
    }

    if (idxQuery == query.length) // found!
        return idxTarget-idxQuery;
    return -1;
}
```

1.2.2 Knuth Morris Pratt

El algoritmo KMP es un algoritmo de búsqueda de subcadenas (patrón/query) dentro de otra cadena. La búsqueda se lleva a cabo utilizando información basada en los fallos previos. Información obtenida del propio patrón creando previamente una tabla de valores sobre su propio contenido. El objetivo de crear dicha tabla es poder determinar donde podría darse la siguiente existencia, sin necesidad de analizar más de 1 vez los caracteres de la cadena donde se busca. Este algoritmo resuelve la búsqueda en un tiempo equivalente a M+N en el peor caso.

Para armar la tabla del patrón se calculan los bordes y se coloca el tamaño del mayor borde distinto de los triviales. Por ejemplo:

query	B	C	B	C
Next	0	0	1	2

Notar que si empezamos con la palabra completa, el único border no trivial es BC (Longitud 2). Si tomamos BCB, el borde no trivial es B (Longitud 1), y luego como no hay borde no trivial se colocan ceros.

Luego para poder usar la tabla, mientras hagamos matching con el target se sigue avanzando, pero en caso de fallar si pquery > 0 entonces pquery cambiaría por next[pquery-1].

IMPLEMENTACIÓN EN JAVA:

```java
private static int[] nextComputation1(char[] query) {
    int[] next = new int[query.length];
    int border=0;  // Length of the current border
    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;
}
```

```java
public static int indexOf( char[] query, char[] target) {
    int pQuery = 0, pTarget = 0, ans = -1;
    int next[] = nextComputation1(query);
    while(pQuery < query.length && pTarget < target.length) {
        if(target[pTarget] == query[pQuery]) {
            pQuery++; pTarget++;
        }
        else {
            if(pQuery > 0) {
                pQuery = next[pQuery - 1];
            }
            else
                pTarget++;
        }
    }
    if(pQuery == query.length)
        ans = pTarget - query.length;
    return ans;
}
```

Ejercicio FindAll: se desea devolver todas las posiciones en que query aparece dentro de target.

```java
private static int[] next_computation(String pattern) {
    int m = pattern.length();
    char[] pat = pattern.toCharArray();
    int[] pi = new int[m];
    int k=0;
    System.out.println(Arrays.toString(pi));
    for (int i=1; i<m; i++) {
        System.out.println(Arrays.toString(pi));
        while ( k>0 && pat[k] != pat[i] )
            k = pi[k];
        if (pat[k] == pat[i]) k++;
        pi[i] = k;
    }
    return pi;
}
```

```java
public static List<Integer> matcher(String mainStr, String patternStr) {
    int ml = mainStr.length();
    int pl = patternStr.length();
    char[] main = mainStr.toCharArray();
    char[] pat = patternStr.toCharArray();
    int[] pi = next_computation(patternStr);
    int q = 0;
    List<Integer> idxs = new ArrayList<>();
    for (int i=0; i<ml; i++) {
        while ( q>0 && pat[q] != main[i] )
            q = pi[q-1];
        if ( pat[q] == main[i] ) q++;
        if (q==pl) {
            idxs.add(i - pl + 1);
            q = pi[q-1];
        }
    }
    return idxs;
}
```

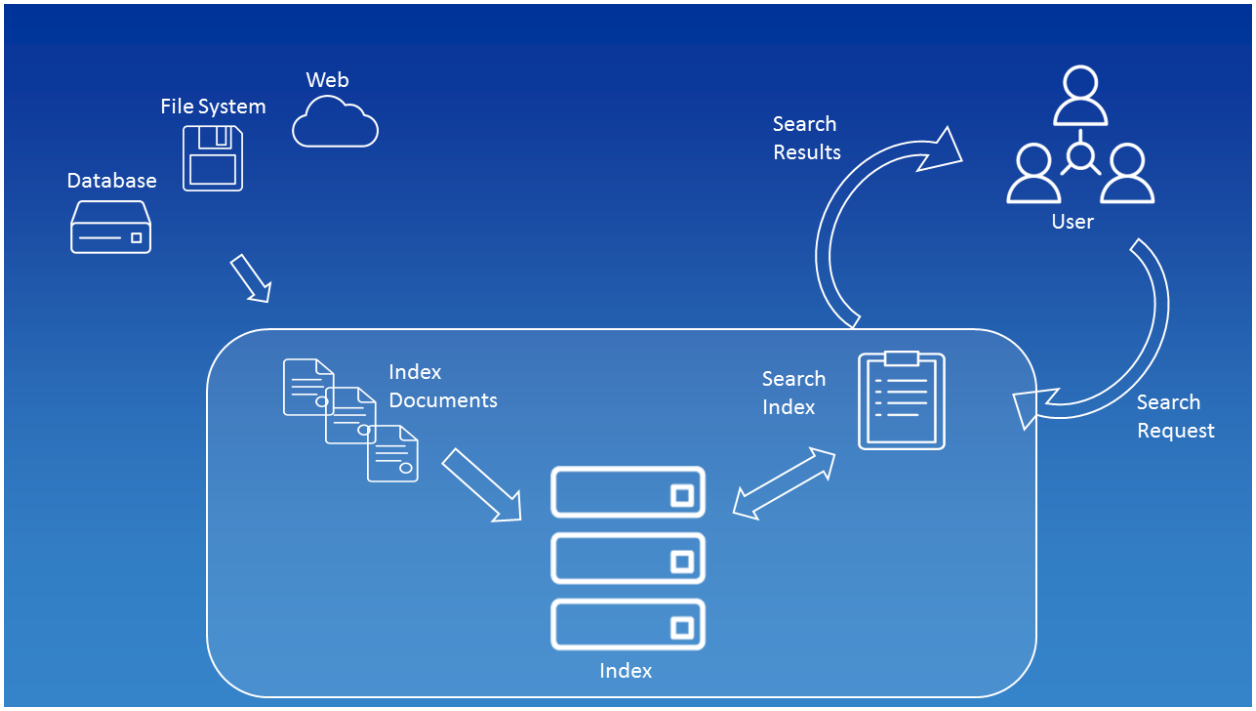
1.2.3 Lucene - Full text retrieval

Lucene es una biblioteca de programas gratuita y de código abierto que cualquier persona puede utilizar y modificar y que está publicada por la Apache Software Foundation. Permite la búsqueda de texto completo, es decir, se trata de un programa que busca en un conjunto de documentos de texto uno o más términos definidos por el usuario. De este hecho se deduce que Lucene no solo se utiliza en el contexto de la World Wide Web, si bien en este las funciones de búsqueda constituyen un elemento omnipresente. También se puede recurrir a Lucene en la búsqueda de archivos, bibliotecas o incluso en el ordenador de sobremesa y, además de aplicarse en documentos HTML, también trabaja con correo electrónico o incluso con archivos PDF.

El índice es un elemento decisivo en el proceso de búsqueda con Lucene y se considera el corazón del programa: en él se almacenan todos los términos de todos los documentos. Este tipo de índice invertido consiste principalmente en una tabla donde se guarda la posición de cada término. No obstante, para poder crear un índice, primero es necesaria la extracción de todos los términos de todos los documentos, proceso que cada usuario puede configurar individualmente. En la configuración, los desarrolladores establecen qué campos quieren incluir en el índice. Pero ¿a qué se refieren estos campos?

Los objetos con los que Lucene trabaja son documentos de cualquier tipo. Estos, desde el punto de vista del propio programa, contienen una serie de campos y estos incluyen,

por ejemplo, el nombre del autor, el título del documento o el nombre del archivo. Cada campo tiene una designación y un valor únicos. Por ejemplo, el campo llamado title puede tener el valor "Manual de usuario de Apache Lucene".



En la indexación de los documentos, tiene lugar también lo que se conoce como tokenization. Para una máquina, un documento es ante todo un conjunto de información. Incluso cuando se pasa del nivel de los bits al de contenido legible para los humanos, un documento sigue estando compuesto por una serie de signos: letras, puntuación, espacios, etc.

A partir de estos datos, se crean con los segmentos de tokenización los términos (en su mayoría palabras) que finalmente van a poder buscarse. La forma más sencilla de ejecutar este tipo de tokenización es mediante el método de espacio en blanco: un término termina cuando se encuentra un espacio en blanco. Sin embargo, este método pierde su utilidad cuando un término está compuesto por varias palabras separadas, como puede ser el caso de "sin embargo". Para solucionarlo, se recurre a diccionarios adicionales que se pueden implementar también en el código Lucene.

Al analizar los datos de los que forma parte el proceso de tokenization, Lucene también ejecuta un proceso de normalización. Esto significa que los términos se convierten en una forma estandarizada, de modo que, por ejemplo, todas las mayúsculas se escriben en minúsculas. Además, Apache Lucene introduce una clasificación mediante una serie de algoritmos, por ejemplo, a través de la medida tf-idf. Como usuario, es probable que primero desees obtener los resultados más relevantes o recientes, lo que es posible gracias a los algoritmos del motor de búsqueda.

EJEMPLO DE ÍNDICE INVERTIDO:

Doc0.txt	Doc1.txt	Doc2.txt	Doc3.txt
store,, game	video	Game video, review game.	game

Value term	Freq en docs	[docid]
game	2	
review	1	
store	1	
video	2	

TIPOS DE BÚSQUEDA:

1. TermQuery: busca un solo término
2. PrefixQuery: busca por prefijo
3. TermRangeQuery: busca por rangos
4. PhraseQuery: busca secuencia
5. WildcardQuery
6. FuzzyQuery: Damerau-Levenshtein con MaxEdit 2
7. BooleanQuery

Para armar estas queries es posible usar el lenguaje de consulta de Lucene (QueryBuilder)

API	QueryBuilder
1.1TermQuery	fieldName:termino
1.2 PrefixQuery	fieldName:term*
1.3 TermRangeQuery	fieldName:{[start TO end]}
1.4 PhraseQuery	fieldName:"term1 ... termN"
1.5 WildcardQuery	fieldName:*subterm?
1.6 FuzzyQuery	fieldName:termino~2
1.7 BooleanQuery	AND OR NOT (OR default)

También es posible usar distintos analizadores para tokenizar nuestros términos:

- SimpleAnalyzer()
- StandardAnalyzer()
- WhitespaceAnalyzer()
- StopAnalyzer() =>
    - CharArraySet sw = StopFilter.makeStopSet("de", "y");
    - new StopAnalyzer(stw);
- EnglishAnalyzer() // opcional stop words
- SpanishAnalyzer() // opcional stop words.
- CustomAnalyzer()

Stemming es un método para reducir una palabra a su raíz o (en inglés) a un stem. Hay algunos algoritmos de stemming que ayudan en sistemas de recuperación de información. Stemming aumenta el recall que es una medida sobre el número de documentos que se pueden encontrar con una consulta. Por ejemplo una consulta sobre "bibliotecas" también encuentra documentos en los que solo aparezca "bibliotecario" porque el stem de las dos palabras es el mismo ("bibliotec").

El algoritmo más común para stemming es el algoritmo de Porter (usado por Lucene). Existen además métodos basados en análisis lexicográfico y otros algoritmos similares (KSTEM, stemming con cuerpo, métodos lingüísticos...).

1.2.4 PORTER

El algoritmo de Porter es un algoritmo que asegura que la forma de las palabras no penalice la frecuencia de éstas. Es decir, una palabra puede estar conjugada en cualquier género, número, persona, y solo se considerará como un solo término.

Ejemplo: «Aquel es un caballo de la caballería militar, los otros caballos no». La frecuencia del término “caball” (que hace referencia a caballo) es 3.

2.Estructuras lineales

2.1) Divide and Conquer

En las ciencias de la computación, el término divide y vencerás (DYV) hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

Esta técnica es la base de los algoritmos eficientes para casi cualquier tipo de problema como, por ejemplo, algoritmos de ordenamiento (quicksort, mergesort, entre muchos otros), multiplicar números grandes (Karatsuba), análisis sintácticos (análisis sintáctico top-down) y la transformada discreta de Fourier. Por otra parte, analizar y diseñar algoritmos de DyV son tareas que lleva tiempo dominar. Al igual que en la inducción, a veces es necesario sustituir el problema original por uno más complejo para conseguir realizar la recursión, y no hay un método sistemático de generalización.

El nombre divide y vencerás también se aplica a veces a algoritmos que reducen cada problema a un único subproblema, como la búsqueda binaria para encontrar un elemento en una lista ordenada (o su equivalente en computación numérica, el algoritmo de bisección para búsqueda de raíces). La corrección de un algoritmo de “divide y vencerás”, está habitualmente probada por una inducción matemática, y su coste computacional se determina resolviendo relaciones de recurrencia.

2.1.1) TEOREMA MAESTRO

En el análisis de algoritmos, el ***teorema maestro*** proporciona una solución sencilla en términos asintóticos (usando una Cota superior asintótica) para ecuaciones de recurrencia que ocurren en muchos algoritmos recursivos tales como en el Algoritmo divide y vencerás. Fue popularizado por el libro Introducción a los algoritmos por Cormen, Leiserson, Rivest, y Stein, en el cual fue introducido y demostrado formalmente. No todas las ecuaciones de recurrencia pueden ser resueltas con el uso del teorema maestro.

Si una recurrencia puede expresarse así:

$$T(N) = a * T\left(\frac{N}{b}\right) + c * N^{d}$$

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

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

- n es el tamaño del problema a resolver.
- a es el número de subproblemas en la recursión.
- n/b el tamaño de cada subproblema. (Aquí se asume que todos los subproblemas tienen el mismo tamaño)
- $c\,n^{d}$ es la combinación de las soluciones parciales

2.2) Arreglos

Algoritmos de ordenamiento

2.1.1) Quicksort

Es un algoritmo de ordenación creado por el científico británico en computación *C. A. R. Hoare.* Opera in-place y es posible implementarlo iterativa o recursivamente.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del conjunto de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la

izquierda del pivote, y otra por los elementos a su derecha.

- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Por ejemplo, supongamos que tenemos la siguiente lista de números y elegimos al último como pivote:

```
5 - 3 - 7 - 6 - 2 - 1 - 4

i                   j   p
```

Podemos iterar los índices i y j, e intercambiarlos si encontramos que alguno del índice i es mayor que el índice j. Finalmente cuando los índices se crucen podemos situar al pivote en el lugar donde se cruzan y de esta manera nos quedarán los menores del lado izquierdo y los mayores del largo derecho.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es $(n \cdot \log n)$.
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. Cabe aclarar que la probabilidad de encontrar un caso así es 1/N.
- En el caso promedio, el orden es $O(n \cdot \log n)$.

Técnicas de elección del pivote:

El algoritmo básico del método Quicksort consiste en tomar cualquier elemento de la lista al cual denominaremos como pivote, dependiendo de la partición en que se elija, el algoritmo será más o menos eficiente. Algunas opciones:

- Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección «a ciegas» siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en $O(n)$ y asegura que hasta en el peor de los casos, el algoritmo sea $O(n \cdot \log n)$. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

IMPLEMENTACIÓN EN JAVA:

```java
public class QSort {
    public static void quicksort(int A[], int izq, int der) {

        int pivote=A[izq]; // tomamos primer elemento como pivote
        int i=izq;         // i realiza la búsqueda de izquierda a derecha
        int j=der;         // j realiza la búsqueda de derecha a izquierda
        int aux;

        while(i < j){                          // mientras no se crucen las búsquedas
            while(A[i] <= pivote && i < j) i++; // busca elemento mayor que pivote
            while(A[j] > pivote) j--;           // busca elemento menor que pivote
            if (i < j) {                        // si no se han cruzado
                aux= A[i];                      // los intercambia
                A[i]=A[j];
                A[j]=aux;
            }
        }

        A[izq]=A[j];      // se coloca el pivote en su lugar de forma que tendremos
        A[j]=pivote;      // los menores a su izquierda y los mayores a su derecha

        if(izq < j-1)
            quicksort(A,izq, der: j-1);          // ordenamos subarray izquierdo
        if(j+1 < der)
            quicksort(A, izq: j+1,der);          // ordenamos subarray derecho
    }
}
```

2.1.2) InsertionSort

// no lo vimos

2.1.3) MergeSort

Fue desarrollado en 1945 por John Von Neumann.

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

- Si la longitud de la lista es 0 o 1, entonces ya está ordenada. En otro caso:
- Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
- Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
- Mezclar las dos sublistas en una sola lista ordenada.

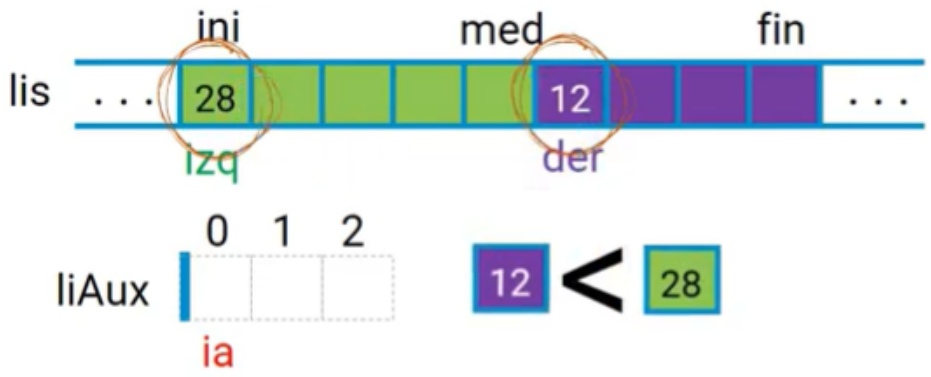
El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

- Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
- Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

El algoritmo en código Java quedaría escrito así:

```java
public static void sort(int arr[], int l, int r) {
    if (l < r) {
        int m =l+ (r-l)/2;
        sort(arr, l, m);
        sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Sólo falta diseñar la función merge, que consiste en mezclar ambas sublistas. Como ambas sublistas ya están ordenadas, en la primera posición de cada sublista está el menor elemento de esa sublista, entonces el menor de éstos será el menor elemento y lo guardaremos en la primera posición.



En este ejemplo, como 12 es menor que 28, 12 irá en la primer posición de liAux, y ahora pasaremos al siguiente elemento de la lista de la derecha y lo compararemos con 28. El menor de éstos, será el siguiente elemento de liAux y así sucesivamente.

VIDEO CON LA EXPLICACIÓN MUY CLARA:
https://www.youtube.com/watch?v=kOgzXagXpTg&ab_channel=ClandelChelela

IMPLEMENTACIÓN EN JAVA:

```java
private static void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[] = new int[n1];
    int R[] = new int[n2];
    for (int i = 0; i < n1; ++i)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; ++j)
        R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) { /* Copy remaining elements of L[] if any */
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) { /* Copy remaining elements of R[] if any */
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

2.1.4) HeapSort

// no lo vimos

2.1.5) Counting Sort

// no lo vimos

2.1.6) Radix Sort

// no lo vimos

2.1.7) Bucket Sort

// no lo vimos

2.3) Pilas y Colas

La **pila/stack/LIFO** 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 que llegó. 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.

Un caso de uso muy habitual es para evaluar expresiones postfijas, prefijas e infijas, planteemos una implementación:

Evaluador postfija:

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

Si extendemos el algoritmo:

- Si el operador current es un un “(“, el mismo debe postergarse hasta que aparezca “)”. Es decir, se lo pushea.
- Si el operador current es un “)” el mismo debe sacar todos los operadores de la pila y concatenarlos en el string de salida hasta encontrar el “(“ que aparea con él. Cuando en el tope aparezca el “(“ debe sacarlo del tope de la pila pero no concatenarlo (ya que los paréntesis no van a la expresión postfija).

IMPLEMENTACIÓN EN JAVA:

```java
public Double evaluate() {
    Stack<Double> aux = new Stack<Double>();
    while (scannerLine.hasNext()) {
        String s = scannerLine.next();
        if (isOperand(s)) {
            aux.push(Double.valueOf(s));
        } else {    // operator o error
            if (!isOperator(s))
                throw new RuntimeException("invalid operator " + s);
            Double operand2;
            if (aux.empty())
                throw new RuntimeException("operand missing");
            else
                operand2 = aux.pop();
            Double operand1;
            if (aux.empty())
                throw new RuntimeException("operand missing");
            else
                operand1 = aux.pop();
            aux.push(eval(s, operand1, operand2));
        }
    }
    double rta = aux.pop();
    if (aux.empty())
        return rta;
    throw new RuntimeException("operator missing");
}
```

```java
private boolean isOperand(String s) {
    try {
        Double.valueOf(s);
    } catch (NumberFormatException e) {
        return false;
    }
    return true;
}

private boolean isOperator(String s) { return s.matches( regex: "\\+|-|\\*|/"); }

private double eval(String op, double val1, double val2) {
    switch (op) {
        case "+":
            return val1 + val2;
        case "-":
            return val1 - val2;
        case "*":
            return val1 * val2;
        case "/":
            return val1 / val2;
    }
    ;
    throw new RuntimeException("invalid operator" + op);
}
```

(Ver diapositivas de 23 a 26 de la Unidad 3/ Pila - Cola / A para un ejemplo hecho a mano)

Veamos ahora el caso de conversión de una expresión de infija a postfija, para luego poder evaluarla:

- Cada vez que aparezcan varios operadores se consultará un 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.

	+	-	*	/
+	true	true	False	false
-	true	true	False	false
*	true	true	True	True
/	true	true	True	true

A esta tabla posteriormente le agregaremos la potencia y los paréntesis.

(Ver diapositivas de 11 a 14 de la Unidad 3/ Pila - Cola / B para un ejemplo hecho a mano)

IMPLEMENTACIÓN EN JAVA:

```java
private static final Map<String, Integer> mapping = new HashMap<String, Integer>() {
    { put("+", 0); put("-", 1); put("*", 2); put("/", 3); put("^", 4); }
};

private static final boolean[][] PRECEDENCES =
        {  { true, true, false, false, false },
           { true, true, false, false, false },
           { true, true, true, true, false },
           { true, true, true, true, false},
           { true, true, true, true, false}
        };

private boolean getPrecedence(String tope, String current) {
    Integer topeIndex;
    Integer currentIndex;
    if ((topeIndex= mapping.get(tope))== null)
        throw new RuntimeException(String.format("tope operator %s not found", tope));
    if ((currentIndex= mapping.get(current)) == null)
        throw new RuntimeException(String.format("current operator %s not found", current));
    return PRECEDENCES[topeIndex][currentIndex];
}
```

```java
public double evaluate(String expression) {
    Scanner scanner = new Scanner(expression).useDelimiter("\\s+");
    StringBuilder ans = new StringBuilder();
    Stack<String> stack = new Stack<String>();
    while(scanner.hasNext()) {
        String currentChar = scanner.next();
        if(isOperand(currentChar)) {
            ans.append(currentChar);
            ans.append(" ");
        }
        else if (isOperator(currentChar)){
            if (!stack.isEmpty()) {
                if (getPrecedence(stack.peek(), currentChar)) {
                    while (!stack.isEmpty() && getPrecedence(stack.peek(), currentChar)) {
                        ans.append(stack.pop());
                        ans.append(" ");
                    }
                }
            }
            stack.push(currentChar);
        }
        else
            throw new IllegalArgumentException();
    }
    while(!stack.isEmpty()) {
        ans.append(stack.pop());
        ans.append(" ");
    }
```

Si queremos extender el algoritmo al uso de paréntesis (que es lo más importante de la notación infija) entonces:

- Si el operador current es un un “(“, el mismo debe postergarse hasta que aparezca “)”. Es decir, se lo pushea.
- Si el operador current es un “)” el mismo debe sacar todos los operadores de la pila y concatenarlos en el string de salida hasta encontrar el “(“ que aparea con él. Cuando en el tope aparezca el “(“ debe sacarlo del tope de la pila pero no concatenarlo (ya que los paréntesis no van a la expresión postfija).

CAMBIOS EN LA IMPLEMENTACIÓN:

```
private static final boolean[][] PRECEDENCES =
        {  { true, true, false, false, false, false, true },
           { true, true, false, false, false, false, true },
           { true, true, true, true, false, false, true },
           { true, true, true, true, false, false, true},
           { true, true, true, true, false, false, true},
           { false, false, false, false, false, false, false},
           { true, true, true, true, true, false, true}
        };
```

```
    else if (isOperator(currentChar)) {
        while (!stack.isEmpty() && getPrecedence(stack.peek(), currentChar)) {
            ans.append(stack.pop());
            ans.append(" ");
        }
        if(currentChar.equals(")")) {
            if(!stack.isEmpty() && stack.peek().equals("(") )
                stack.pop();
            else
                throw new RuntimeException("Missing (");
        }
        else
            stack.push(currentChar);
    }
    else
        throw new IllegalArgumentException();
}
while(!stack.isEmpty()) {
    if(stack.peek().equals("("))
        throw new RuntimeException("Missing )");
    else {
        ans.append(stack.pop());
        ans.append(" ");
    }
}
```

La Cola (Queue) es una colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de dos elementos distinguidos: FIRST indica cuál es el más antiguo de los elementos de la colección y tiene prioridad para salir, y LAST marca el elemento más reciente que ha llegado a la colección

Las operaciones que debe ofrecer son:

- queue(element): agrega un elemento a la colección convirtiéndolo en el más reciente o sea, se convierte en el nuevo LAST.
- deque(): quita el elemento más antiguo de la colección (FIRST) y cambia el FIRST. Es una operación destructiva y solo puede usarse si la colección no está vacía.
- peek(): devuelve el elemento más antiguo de la colección (FIRST) sin removerlo (sin cambiar el FIRST). No es destructiva. Solo puede usarse si la colección no está vacía.
- isEmpty(): devuelve true/false según la colección tenga o no elementos.
- size(): (opcional) devuelve la cantidad de elementos de la colección y es ideal para estimar cuánto hay que esperar para ser atendido.

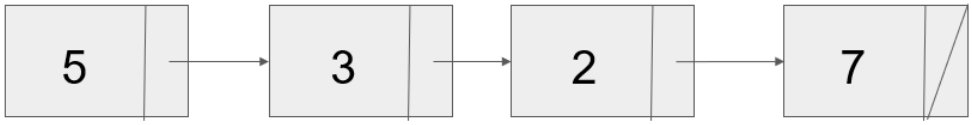
2.3) Listas

En ciencias de la computación, una lista enlazada es una de las estructuras de datos fundamentales, y puede ser usada para implementar otras estructuras de datos. Consiste en una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros al nodo anterior o posterior. El principal beneficio de las listas enlazadas respecto a los vectores convencionales es que el orden de los elementos enlazados puede ser diferente al orden de almacenamiento en la memoria o el disco, permitiendo que el orden de recorrido de la lista sea diferente al de almacenamiento. Una lista enlazada es un tipo de dato autorreferencial porque contienen un puntero o enlace a otro dato del mismo tipo. Las listas enlazadas permiten inserciones y eliminación de nodos en cualquier punto de la lista en tiempo constante (suponiendo que dicho punto está previamente identificado o localizado), pero no permiten un acceso aleatorio. Existen diferentes tipos de listas enlazadas: listas enlazadas simples, listas doblemente enlazadas, listas enlazadas circulares y listas enlazadas doblemente circulares.

Lista Simplemente Encadenada

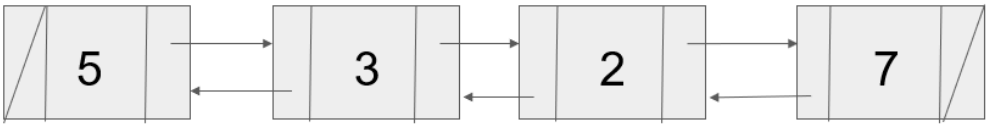
- Estructura de datos alternativa a arreglos para guardar información.

- Cada nodo tiene un valor y un puntero hacia el siguiente elemento.



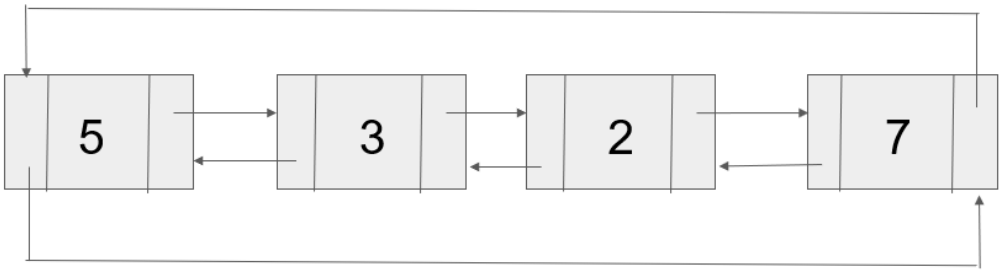
Lista Doblemente Encadenada

- Cada nodo además guarda una referencia al nodo anterior
- Permiten agregar o quitar nodos eficientemente.
- Se puede recorrer en sentido inverso eficientemente.



Lista Circular

- El último nodo además tiene una referencia al primero



Listas vs Arreglos:

	Listas	Arreglos
Inserción en principio o fin	O(1)	O(N)
Inserción por índice	O(N)	O(N)
Borrado en principio o fin	O(1)	O(N)
Borrado por índice	O(N)	O(N)
Búsqueda por índice	O(N)	O(1)

REFERENCIAS

1. Escribe aquí tu texto
2. Escribe aquí tu texto
3. Escribe aquí tu texto Escribe aquí tu texto