Universidad Europea del Atlántico

Asignatura:

Estructuras de datos y algoritmos II

Universidad Europea del Atlántico

Índice

Capítulo 1

ESTRUCTURAS DE DATOS EN JAVA

Con anterioridad hemos visto las estructuras de datos desde un punto de vista interno y de cómo funcionan y se implementan. La realidad es que no vamos a desarrollar nuestras propias estructuras de datos una vez tengamos que enfrentarnos a un problema de programación. Normalmente los programadores se basan en código que ya existe y ha sido fuertemente testeado en infinidad de aplicaciones. Estas son las denominadas librerías.

En java como en cualquier otro lenguaje de programación, existen un conjunto de librerías que proporcionan al programador código ya creado que hará la vida del programador más fácil.

Ahora vamos a ver las estructuras de datos Java más comunes clasificadas por las interfaces que implementan.

```
Interface java.lang.Iterable
 Interface Collection
    Interface List
       Class ArrayList
       Class LinkedList (implements List )
       Class Vector
          Class Stack (legacy class, use Deque, which is more pow-
          erful)
    Interface Set
       Class HashSet
```

```
            Class LinkedHashSet
         Interface SortedSet
            Interface NavigableSet
               Class TreeSet
         Class EnumSet
      Interface Queue
         Class PriorityQueue
         Interface Deque
            Class LinkedList (implements  List)
            Class ArrayDeque

   Interface Map
   Class HashMap
   Interface SortedMap
      Interface NavigableMap
         Class TreeMap
```

Vamos a ver las principales:

ArrayList

Es una clase que implementa una lista con un Array. Nos va a permitir almacenar datos en memoria. La gran ventaja es que en este caso a diferencias de los Arrays no tenemos que definir el tamaño por defecto. ArrayList nos va a permitir cambiar el tamaño de forma dinámica.

Otra cosa a tener en cuenta de los ArrayList son los iterator() que nos van a permitir iterar a través de los elementos del mismo. En cuanto a su performance:

Acceso posicional eficiente, extraccción e inserción costosa.

Ejemplo de uso de un ArrayList:

```
ArrayList<String> nombreArrayList = new ArrayList<String>();


nombreArrayList.add("1");
int n = 3;
nombreArrayList.add(n , "2");
nombreArrayList.size();
nombreArrayList.get(2);
```

```
nombreArrayList.contains("2");
nombreArrayList.indexOf("2");
nombreArrayList.lastIndexOf("2");
nombreArrayList.remove(n);
nombreArrayList.remove("1");
nombreArrayList.clear();
nombreArrayList.isEmpty();

ArrayList arrayListCopia = (ArrayList) nombreArrayList.clone();
Object[] array = nombreArrayList.toArray();
```

LinkedList

La clase LinkedList implementa una lista con el formato de lista enlazada. Su acceso posicional es costoso así como la inserción y extracción. Esta clase hereda de AbstractSequenceList e implemeta la interface List.

Ejemplo usando LinkedList.

```
LinkedList<String> nombreArrayList = new LinkedList<String>();

nombreArrayList.add("1");
int n = 3;
nombreArrayList.add(n , "2");
nombreArrayList.size();
nombreArrayList.get(2);
nombreArrayList.contains("2");
nombreArrayList.indexOf("2");
nombreArrayList.lastIndexOf("2");
nombreArrayList.remove(n);
nombreArrayList.remove("1");
nombreArrayList.clear();
nombreArrayList.isEmpty();

ArrayList arrayListCopia = (ArrayList) nombreArrayList.clone();
Object[] array = nombreArrayList.toArray();
```

Vector

A vector class implementa un array dinámico. Es muy parecido a un ArraList, pero existen dos diferencias principales:

- Vector está sincronizado.
- Vector contiene muchos métodos que no forman parte de las collecciones.

Por esas razones se puede decir que los vectores suelen ser utilizados cuando no se tiene ni idea del tamaño que va a tener el array y se necesita cambiar el tamaño constantemente.

```
// initial size is 3, increment is 2
Vector v = new Vector(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
v.addElement(new Integer(1));
v.addElement(new Integer(2));
v.addElement(new Integer(3));
v.addElement(new Integer(4));
System.out.println("Capacity after four additions: " +
        v.capacity());
      v.addElement(new Double(5.45));
System.out.println("Current capacity: " +
    v.capacity());
    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Current capacity: " +
    v.capacity());
    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Current capacity: " +
    v.capacity());
    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
    System.out.println("First element: " +
       (Integer)v.firstElement());
    System.out.println("Last element: " +
       (Integer)v.lastElement());
    if(v.contains(new Integer(3)))
```

```
        System.out.println("Vector contains 3.");
    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();
    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
        System.out.print(vEnum.nextElement() + " ");
    System.out.println();
```

LinkedHashSet

Esta clase hereda o extends de HashSet. LinkedHashSet matiene una lista de entidades enlazadas en el mismo órden en el que fueron insertadas. Lo cual significa que si recorremos la lista usando un iterator(), los elementos se devolverán en el órden en el que fueron insertados. El hash es entonces utilizado cómo un índice que es el hascode asociado al key. Esta transformación del key en hashcode es automático.

Ejemplo de uso:

```
// hash set
      LinkedHashSet hs = new LinkedHashSet();
      // Adir elementos
      hs.add("B");
      hs.add("A");
      hs.add("D");
      hs.add("E");
      hs.add("C");
      hs.add("F");
      System.out.println(hs);
```

TreeSet

Es una implementación de la interface Set que usa un árbol para almacenar los datos. Los objetos en esta estructura se almacenan ordenados y de forma ascendente.

Los accesos y recuperación de los datos es muy rápido. Por ello es una buena elección cuando queremos almacenar grandes cantidades de datos y tener acceso rápido y eficiente.

```
//tree set
	TreeSet ts = new TreeSet();
	// Añadir elementos
	ts.add("C");
	ts.add("A");
	ts.add("B");
	ts.add("E");
	ts.add("F");
	ts.add("D");
	System.out.println(ts);
```

EnumSet

Es una clase collection. En este caso es un tipo un poco especial que se utiliza para enumerar elementos del mismo tipo. No podemos mezclar tipos dentro de los EnumSet. Por lo tanto es seguro en cuanto a tipado.

LinkedList

Esta clase hereda de AbstractSequenceList e implemeta la interface List.

```
// create a linked list
	LinkedList ll = new LinkedList();
	// add elements to the linked list
	ll.add("F");
	ll.add("B");
	ll.add("D");
	ll.add("E");
	ll.add("C");
	ll.addLast("Z");
```

```
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);

// remove elements from the linked list
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "
 + ll);

// remove first and last elements
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "
 + ll);
// get and set a value
Object val = ll.get(2);
ll.set(2, (String) val + " Changed");
System.out.println("ll after change: " + ll);
```

HashMap

Esta clase usa una Hashtable que a su vez implementa la interface Map.

```
// Create a hash map
      HashMap hm = new HashMap();
      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();
      // Get an iterator
      Iterator i = set.iterator();
      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();
      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
```

```
hm.put("Zara", new Double(balance + 1000));
System.out.println("Zara's new balance: " +
hm.get("Zara"));
```

TreeMap

Implementa la interface Map pero usando un Árbol-Tree. Al usar una estructura de árbol es muy eficiente porque los datos se almacenan de forma ordenada y ascendente.

```
// Create a hash map
   TreeMap tm = new TreeMap();
   // Put elements to the map
   tm.put("Zara", new Double(3434.34));
   tm.put("Mahnaz", new Double(123.22));
   tm.put("Ayan", new Double(1378.00));
   tm.put("Daisy", new Double(99.22));
   tm.put("Qadir", new Double(-19.08));

    // Get a set of the entries
   Set set = tm.entrySet();
   // Get an iterator
   Iterator i = set.iterator();
   // Display elements
   while(i.hasNext()) {
      Map.Entry me = (Map.Entry)i.next();
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
   }
   System.out.println();
```

```
// Deposit 1000 into Zara's account
double balance = ((Double)tm.get("Zara")).doubleValue();
tm.put("Zara", new Double(balance + 1000));
System.out.println("Zara's new balance: " +
tm.get("Zara"));
```

1.1. EJERCICIOS EVALUACIÓN CONTINUA

1. Desarrollar un programa que cree cinco objetos del tipo Persona. Cada persona tiene un identificador y nombre. Se quieren almacenar 5 objetos de manera que queden ordenados a través del identificador. Su ordenación será de menor a mayor.
2. Después se pide cambiar la estructura de datos para poder almacenar de mayor a menor.
3. Finalmente se pide la creación de una clase mediante el patrón Factory para decidir el método por medio de un string en su constructor "asc" ascendete y "desc" descendente.

```
package treeset.dataestructures.atlantico.es;

public class Person
{
        private int id;
        private String name;

        public Person(int idParam,String nameParam){
                this.id =  idParam;
                this.name = nameParam;
        }

        public int getId() {
                return id;
```

```
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

```
package treeset.dataestructures.atlantico.es;

public enum SortingOrder {

      ASC,
      DESC

}
```

```
package treeset.dataestructures.atlantico.es;

import java.util.Comparator;

public class PersonOrderAsc implements Comparator<Person>
{

      @Override
      public int compare(Person arg0, Person arg1) {

            if (arg0.getId() == arg1.getId())
                  return 0;
```

```
            else if (arg0.getId() < arg1.getId())
                  return -1;
            else
                  return 1;
      }

}
```

```
package treeset.dataestructures.atlantico.es;

import java.util.Comparator;

public class PersonOrderDesc implements Comparator<Person>
{

      @Override
      public int compare(Person arg0, Person arg1) {

            if (arg0.getId() == arg1.getId())
                  return 0;
            else if (arg0.getId() > arg1.getId())
                  return -1;
            else
                  return 1;
      }

}
```

```
package treeset.dataestructures.atlantico.es;

import java.util.Comparator;

public class PersonSortFactory {

      public Comparator<Person> GetSortableType(SortingOrder sort){
            if (sort == SortingOrder.DESC)
                  return new PersonOrderDesc();
            else
                  return new PersonOrderAsc();
      }
```

```
}


package treeset.dataestructures.atlantico.es;

import java.util.TreeSet;

import org.junit.Test;

import org.junit.Assert;


public class TreeSetTest {


	@Test
	public void TestSorteredList(){

		TreeSet<Person> treeList = new TreeSet<Person>(new PersonOr-
derAsc());

		Person p1 = new Person(10,"Juan");
		Person p2 = new Person(50,"Patricia");
		Person p3 = new Person(1,"Peter");
		Person p4 = new Person(6,"John");
		Person p5 = new Person(3,"Daniel");

		treeList.add(p1);
		treeList.add(p2);
		treeList.add(p3);
		treeList.add(p4);
		treeList.add(p5);

		Person[] array = treeList.toArray(new Per-
son[treeList.size()]);

		Assert.assertEquals(p3.getId(), array[0].getId());
		Assert.assertEquals(p5.getId(), array[1].getId());
		Assert.assertEquals(p4.getId(), array[2].getId());
		Assert.assertEquals(p1.getId(), array[3].getId());
		Assert.assertEquals(p2.getId(), array[4].getId());
```

```
    }


    @Test
    public void TestSorteredListDesc(){

        TreeSet<Person> treeList = new TreeSet<Person>(new PersonOr-
derDesc());

        Person p1 = new Person(10,"Juan");
        Person p2 = new Person(50,"Patricia");
        Person p3 = new Person(1,"Peter");
        Person p4 = new Person(6,"John");
        Person p5 = new Person(3,"Daniel");

        treeList.add(p1);
        treeList.add(p2);
        treeList.add(p3);
        treeList.add(p4);
        treeList.add(p5);

        Person[] array = treeList.toArray(new Per-
son[treeList.size()]);

        Assert.assertEquals(p2.getId(), array[0].getId());
        Assert.assertEquals(p1.getId(), array[1].getId());
        Assert.assertEquals(p4.getId(), array[2].getId());
        Assert.assertEquals(p5.getId(), array[3].getId());
        Assert.assertEquals(p3.getId(), array[4].getId());

    }

    @Test
    public void TestSorteredListFactory(){

        PersonSortFactory factory = new PersonSortFactory();
        TreeSet<Person> treeList = new TreeSet<Person>(factory.Get-
SortableType(SortingOrder.DESC));

        Person p1 = new Person(10,"Juan");
```

```
            Person p2 = new Person(50,"Patricia");
            Person p3 = new Person(1,"Peter");
            Person p4 = new Person(6,"John");
            Person p5 = new Person(3,"Daniel");

            treeList.add(p1);
            treeList.add(p2);
            treeList.add(p3);
            treeList.add(p4);
            treeList.add(p5);

            Person[] array = treeList.toArray(new Per-
son[treeList.size()]);

            Assert.assertEquals(p2.getId(), array[0].getId());
            Assert.assertEquals(p1.getId(), array[1].getId());
            Assert.assertEquals(p4.getId(), array[2].getId());
            Assert.assertEquals(p5.getId(), array[3].getId());
            Assert.assertEquals(p3.getId(), array[4].getId());

      }

}
```

Capítulo 2

INTRODUCCIÓN A LOS ALGORITMOS

En este capítulo vamos a intentar entender que son los algoritmos y cuál es el objetivo de los mismos dentro del campo de la computación.

La definición formal de un algoritmo podrá ser la siguiente:

> Un algoritmo es un procedimiento computacional que toma como entrada un valor o valores y produce como salida otro valor o valores como resultado de ese conjunto de pasos internos entre la entrada y la salida. Otra forma muy común de verlo es la solución a un determinado problema usando un método computacional.

Un caso muy común en el que usamos la algoritmia, es cuando queremos ordenar algo. Por regla general tenemos un conjunto de elementos de entrada que están desordenados y queremos que aparezcan ordenados a la salida. La ordenación puede ser diversa. Podemos ordenar de mayor a menor o por el contrario de menor a mayor.

Vamos a intentar desgranar este problema que vamos a llamar el problema de ordenar algo. Como ya hemos descrito anteriormente definimos una entrada, salida y pasos intermedios. En este caso vamos a ordenar un conjunto de número enteros.

Entrada: 5, 1, 6, 9, 4, 7

Pasos intermedios: $x1 < x2 < x3 <... < Xn$

Salida: 1, 4, 5, 6, 7, 9

La ordenación es un problema que se tiene que solucionar muy frecuentemente en programación y por ello existen muy diversos algoritmos que dan solución a este problema. La selección del mejor algoritmo depende de muchos factores. Por ejemplo el número de elementos a ser ordenados, que porcentaje de los mismos ya se encuentran ordenados, cómo se almacenan los datos (memoria, disco,...), el tipo de cpu del que disponemos etc.

En esta asignatura nos vamos a centrar en la algoritmia descrita a través de la programación, pero esto no tiene porqué ser siempre as. Es posible desarrollar algoritmos a más bajo nivel dentro de la arquitectura de la computadora o a nivel de lógica de circuitos.

2.1. APLICACIONES PRÁCTICAS

El Genoma Humano

El denominado proyecto del genoma humano es uno de los problemas más complejos que se han intentado resolver desde el principio de los tiempos. Su principal objetivo es identificar los 100,000 genes pertenecientes a la cadena de ADN. Almacenar esta información y hacerla accesible para su análisis. Todos estos pasos requieren de sofisticados algoritmos para su puesta en funcionamiento. Aunque es obvio que en esta asignatura no vamos a intentar desgranar este proceso, es cierto que muchos de los algoritmos que vamos a ver son aplicados diariamente en el Genoma Humano.

Rutas comerciales

Diariamente infinidad de negocios se enfrentan al problema de la logística. Uno de los mundos más apasionantes y complejos dentro del mundo de la empresa. Es fundamental entregar y proveer de productos a empresas y clientes que se encuentran en sitios dispares del mundo. Para ello es muy importante la eficiencia para intentar que los productos se muevan de la manera más rápida, segura y barata posible. Para resolver este problema se utilizan infinidad de algoritmos. Un caso claro es el cálculo de la ruta más corta entre un origen y un destino.

Sugerencias basadas en preferencias

Uno de los casos de mayor éxito dentro del mundo empresarial para este tipo de algoritmos es Amazon. El usuario recibe sugerencias de productos en los que puede estar interesado gracias a un complejo algoritmo. En este caso el algoritmo recopila información acerca de las preferencias del usuario, productos comparados

anteriormente, localización, segmento de edad y genera productos en los que el usuario pudiera estar interesado. Para ello utiliza machine learning, ontologías, taxonomías etc.

2.2. ANALIZANDO UN ALGORITMO

Este proceso es básicamente intentar predecir los recursos que va a necesitar para su correcto funcionamiento. Recursos son la memoria, ancho de banda, cpu, energía. De todos estos elementos el que más se controla es el tiempo de computación.

El proceso más utilizado para evaluar la eficiencia de un algoritmo, son los bancos de pruebas o benchmarking. Este proceso trata de evaluar métricas, para poder comparar en base a esas métricas la eficiencia de diversos algoritmos al ser ejecutados bajo las mismas condiciones.

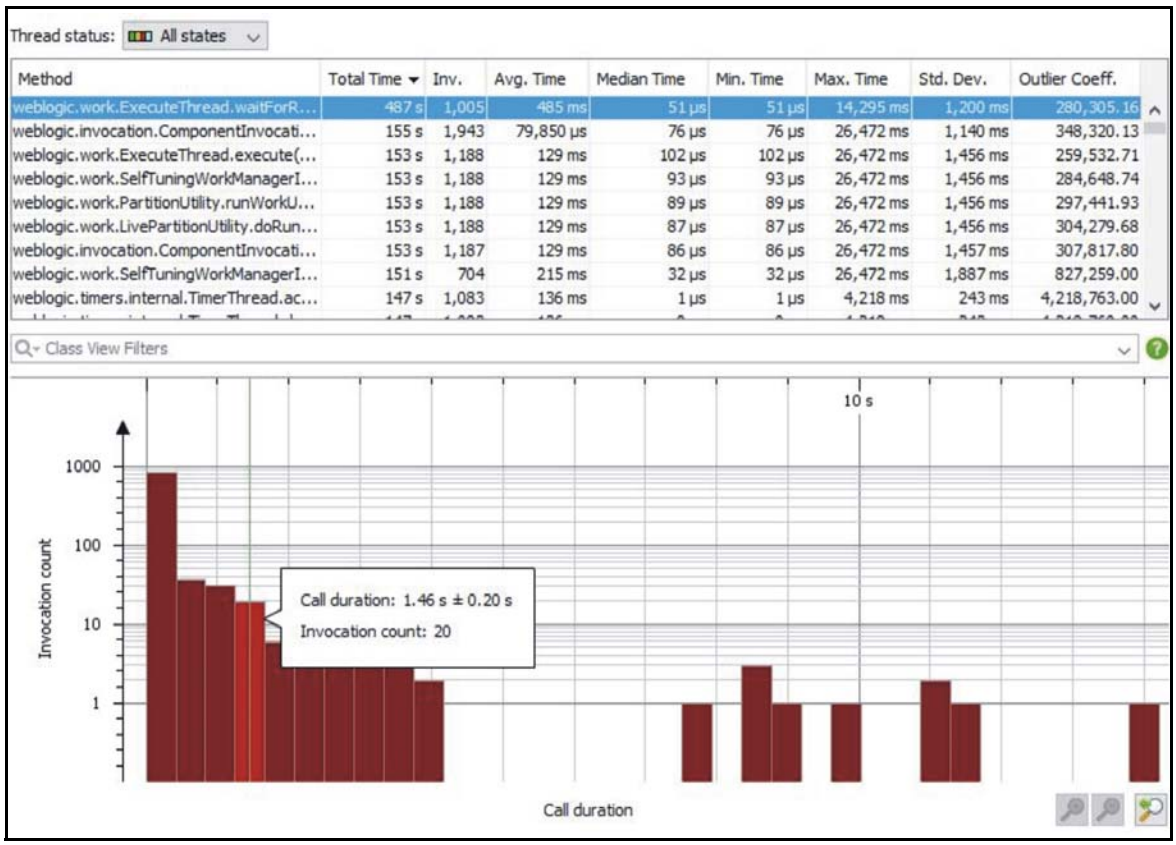


Figura 2.1. Ejemplo de benchmarking – banco de pruebas.

2.3. COMPLEJIDAD DE UN ALGORITMO

La complejidad de un algoritmo es una métrica teórica que nos permite analizar como de complejo es un algoritmo basado en unas métricas matemáticas. Como bien sabemos la resolución de un problema, se puede realizar en ciertos casos de muchas maneras posibles.

Es ideal poder analizar para cada solución cuál es su complejidad.

Existen diferentes maneras de analizar esta complejidad. La **complejidad temporal y la complejidad espacial**. La complejidad de un algoritmo es un valor y en el caso que más nos interesa que es la complejidad temporal este valor ser el tiempo de ejecución.

Por ejemplo, piensa en un típico algoritmo para ordenar los elementos de un vector. Seguro que conoces alguno. El algoritmo consta de una serie de instrucciones que se repiten una y otra vez (bucles), y probablemente, de una serie de selecciones (comparaciones) que hacen que se ejecute uno u otro camino dentro del algoritmo.

Se hace necesaria una pregunta: Tardar lo mismo un algoritmo de ordenación en ordenar un vector con 100 valores que uno con 100.000 valores?.... Obviamente no. Pues aquí es donde tenemos que empezar a hablar del tamaño del problema.

La complejidad se calcula en función de una talla genérica, y no concreta. Por ejemplo, la complejidad de un algoritmo de ordenación se calcula pensando en un array de longitud *n*, y no 1, 100 o 100.000 elementos.

Otra cosas a tener en cuenta es que la complejidad no es un tiempo sino una función. Es claro entender que la ejecución de un algoritmo en diferentes CPU nos dar diferentes tiempos de ejecución dependiendo de la capacidad de procesamiento de la CPU.

2.4. NOTACIÓN BIG O

https://www.youtube.com/watch?v=v4cd1O4zkGw

https://www.youtube.com/watch?v=V6mKVRU1evU

Vamos ahora a analizar el siguiente ejemplo llamado Juego de adivinanzas.

Un juego de adivinanzas

Supongan el siguiente juego:

> *"Un jugador piensa un número que está entre 1 y 1.000, el otro debe tratar de adivinarlo haciéndole preguntas al primero, a las que sólo se puede responder con un S o un No".*
>
> *Una manera de hacerlo es preguntando por cada número secuencialmente: "¿es el 1?", respuesta: No. "¿Es el 2?", "No". "Es el 3", "No". etc.*
>
> *Si la persona pensó en el 999 o en el 1.000 esta no es la forma más eficiente de adivinar el número. Podemos preguntar en forma "saltada" o aleatoria: "¿Es el 523?", No. "¿Es el 256?", "No"...*
>
> *Esa forma requiere que anotemos en algo cuaderno o libreta cuales números hemos preguntado. El problema de hacer eso es que al generar los números de forma aleatoria tendríamos que ir revisando la lista que llevamos escrita para evitar repetirlos.*
>
> *Una mejora sera tener una hoja de papel con los números del 1 al 1000 donde vamos tachando los ya preguntados, esto permite ver de forma directa lo preguntado sin tener que revisar nuestra libreta de notas.*
>
> *Tanto si hacemos las preguntas de forma secuencial, como si las hacemos de forma aleatoria, la cantidad de preguntas puede ir entre 1 a 999. Haremos sólo 1 pregunta si adivinamos el número a la primera, si tenemos mala suerte tendremos que hacer 999 preguntas. A menos que estemos muy desconcentrados nunca haremos más de 1.000 preguntas. Si el juego consiste en adivinar un número entre 1 y N, la cantidad de preguntas que haremos también ser un número entre 1 y N, y nunca haremos más de N preguntas.*
>
> *Pero hay otra manera de resolver este problema. Podemos partir por el medio y hacer la siguiente pregunta: "¿El número que piensas es menor que 500?", el otro jugador podrá responder "Sí", lo que significa que el número tiene que estar entre 1 y 499, con esta simple pregunta hemos reducido el universo de valores posibles a adivinar a la mitad. Podemos volver a preguntar: "¿El número es menor que 250?", un "Sí" como respuesta reduce el universo al rango 1 a 249. Preguntamos por tercera vez: "¿es el número menor que 125?", y si la respuesta es un "No", sabemos que el número tiene que estar entre 125 y 249 y podemos seguir as, siempre dividiendo el rango por la mitad.*
>
> *Lo interesante es que de esta forma nunca haremos más de 10 preguntas para adivinar el número. Si tenemos que adivinar un numero entre 1 y un millón*

tendremos que hacer a lo más unas 20 preguntas. En general, para cualquier número N haremos a lo más una cantidad de preguntas menor o igual al logaritmo en base 2 de N. Esto, porque siempre dividimos el rango de búsqueda en dos. Además con esto hemos descubierto un algoritmo eficiente para adivinar el número.

En el ejemplo anterior, del juego de adivinanza de un número entre 1 y N, cuando usamos el primer algoritmo, de ir preguntando en forma secuencial, o el segundo algoritmo, en que preguntamos al azar, tenemos que la cantidad de preguntas a realizar es a lo más N, en ese caso decimos que el algoritmo es de orden N, y lo denotamos: O(N).

En el segundo caso, en que acotamos el rango de búsqueda dividiéndolo en dos partes, decimos que el algoritmo es O(lg N), donde lg denota el logaritmo en base 2.

La notación O(f(N)) denota, en general, que la cantidad de operaciones básicas del algoritmo es una función de N. Por ejemplo, un algoritmo de ordenamiento conocido como el método de la burbuja requiere de N al cuadrado operaciones para ordenar una lista de N números, decimos entonces que ese algoritmo es O(N2), donde N2 es una notación para designar N elevado a 2. Hay otras formas de ordenar más eficientes, como QuickSort, que divide el problema sistemáticamente en dos grupos, en este caso el algoritmo es O(N lg N), es decir, N multiplicado por el logaritmo base 2 de N.

2.5. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Desarrollar un algoritmo que solucione el problema de ordenación de una secuencia de números por entrada. La salida tiene que ser una lista de números que descarte elementos repetidos y que los ordene de menor a mayor. Realizar pruebas unitarias para comprobar su correcto funcionamiento.
2. Analizar la ejecución del algoritmo utilizando un banco de pruebas. Para ello vamos a analizar diferentes elementos como la memoria, cpu etc. Analizar diferentes productos en el mercado y seleccionar el que más se ajuste a las necesidades que tenemos para analizar nuestro algoritmo.

Capítulo 3

RECURSIVIDAD

Muchas veces al desarrollar un algoritmo nos damos cuenta que la solución natural es llamarse a sí mismo tantas veces como sea necesario, eso es denominado en programación recursividad. Es importante destacar que para evitar la ejecución infinita es necesario definir una regla de fin.

Una de los mayores riesgos cuando trabajamos con recursividad es que la ejecución sea infinita. Es decir que si lo ejecutamos nunca termine entrando en algo llamado bucle infinito.

Vamos a ver un ejemplo muy claro que utilizando recursividad queda solucionado. Este es el caso del cálculo factorial de un número.

La ejecución de un algoritmo recursivo hace un uso intensivo de la pila. Cada vez que se hace una llamada a una función, se almacena en la pila el resultado de esa llamada. Por defecto en la ejecución de los programas se reserva un espacio considerable en la pila, pero debido a que el número de llamadas puede ser muy grande hay que tener cuidado y vigilarlo.

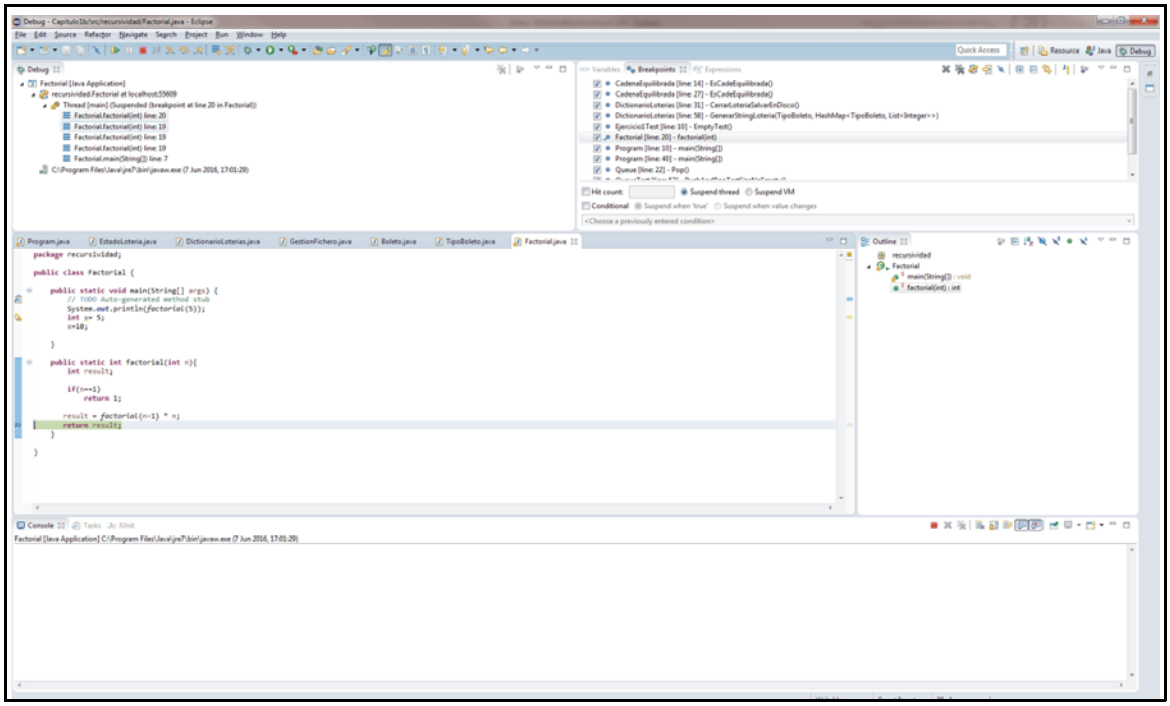
Fórmula matemática f(n) = n * (n-1)! Ejemplo f(5) = 5*4*3*2*1 = 120

Código fuente ejemplo del cálculo factorial:

```
package recursividad;

public class Factorial {
```

```
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		System.out.println(factorial(5));

	}

	public static int factorial(int n){
		int result;

		if(n==1)
			return 1;

	 result = factorial(n-1) * n;
	 return result;
	}

}
```

En la siguiente imagen se pueden apreciar las llamadas en la pila.



Es importante entender las diferencias entre la solución iterativa y recursiva. En los dos casos el código de la función se carga en memoria una vez. En lo que se diferencias es en el uso de la pila. En la solución iterativa se almacena sólo en la llamada inicial y por el contrario en la solución recursiva se almacenará tantas veces como llamadas. Hay que tener en cuenta que cada solución recursiva se puede siempre transformar en una solución iterativa.

Aunque la solución recursiva pueda parecer compleja, no deja de ser una forma simple,natural y más elegante.

3.1. TIPOS DE RECURSIVIDAD

- **Recursividad simple:** Es el tipo de recursividad más común. Es cuando una función se llama a si misma.
- **Recursividad mútua:** En este tipo de recursividad la función no se llama a sí misma, sino que lo hace a través de otra. Un caso muy común es la creación de un algoritmo que identifique si un número es par o impar.


```
package recursividad;


public class Paridad{

	public static boolean impar (int numero){

	    if (numero==0)

	     return false;

	    else

	     return par(numero-1);

	}


	public static boolean par (int numero){

	    if (numero==0)

	     return true;
```

```
            else
                return impar(numero-1);
        }
}
```

3.2. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Programar un algoritmo recursivo que calcule un número de la serie fibonacci.

```
public static int fibonaci(int n){
        if(n==1 || n==2) {
              return 1;
        }
        else{
              return fibonaci(n-1)+fibonaci(n-2);
        }
}
```

2. Programar un algoritmo recursivo que permita invertir un número.

```
public static int invertirNum(int n) {
       if (n < 10) {
             return n;
       }
       else {
             return (n % 10) + invertirNum(n / 10) * 10;
       }
}
```

3. Programar un algoritmo recursivo que permita sumar los elementos de un vector

```
public static int sumaVector(int v [], int n) {
        if (n == 0) {
              return v [n];
        }
        else {
```

```
            return sumaVector(v, n - 1) + v [n];
        }
    }
```

4. Programar un algoritmo recursivo que permita sumar los elementos de una matriz

```
public int suma(int fila, int col, int orden, int mat [] []){
      if (fila == 0 && col == 0)
            return mat [0] [0];
      else if (col < 0)
        return suma (fila - 1, orden, orden, mat);
      else
        return mat [fila] [col] + suma (fila, col - 1, orden, mat);
```

5. Desarrollar un algoritmo recursivo que ejecute peticiones a un url con un máximo de 5 intentos con un espacio de tiempo de 3 segundos entre cada intento.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;

public class UrlConnector {

   public String GetUrl(String urlParam,int retries){
         String contenido = "";
         String line = "";

         if (retries == 0)
               return "-1";

         try {
            URL url = new URL(urlParam);
            URLConnection con = url.openConnection();

            BufferedReader in = new BufferedReader(new InputStream-
Reader(
```

```
                    con.getInputStream()));

                while ((line = in.readLine()) != null) {
                   contenido += line;
                }
                return contenido;
            }
             catch (Exception e) {
                   System.out.println(e.getMessage());
                   return GetUrl(urlParam,retries-1);
             }
      }
   }
```

```
import org.junit.Assert;
import org.junit.Test;

public class UrlConnectorTest {

      @Test
      public void TestPuedoConectar()
      {
            UrlConnector conector = new UrlConnector();
            String value = conector.GetUrl("http://www.google.es", 5);

            Assert.assertTrue(value.concat("google"), true);
      }

      @Test
      public void TestNoPuedoConectar()
      {
            UrlConnector conector = new UrlConnector();
            String value = conector.GetUrl("www.erdfgfgf.es", 5);

            Assert.assertTrue(value.concat("google"), true);
      }

}
```

Capítulo 4

ALGORITMOS DE ORDENACIÓN

Los algoritmos de ordenación son aquellos que dada una lista de elementos a la entrada, realizan una ordenación de los mismos permutándolos. La salida por lo tanto estará ordenada basada en una relación de órden dada. Esto significa que por ejemplo la ordenación puede ser ascendente o descendente.

Los algoritmos de ordenación son una parte muy importante desde los inicios de la programación y computación. Es un campo que ha recibido muchos esfuerzos desde el punto de vista de la investigación. La eficiencia en la ordenación es fundamental para muchos problemas del mundo real.

La clasificación de los algoritmos de ordenación es la siguiente:

1. Lugar donde se realice la ordenación
   - **Algoritmos de ordenamiento interno:** en la memoria del ordenador. BubbleSort, BucketSort etc..
   - **Algoritmos de ordenamiento externo:** en un lugar externo como un disco duro. External merge sort. No tenemos suficiente tamaño en memoria para trabajar.
2. Por el tiempo que tardan en realizar la ordenación, dadas entradas ya ordenadas o inversamente ordenadas:
   - **Ordenación natural:** Tarda lo mínimo posible cuando la entrada está ordenada comparándolo con el tiempo de proceso de otras secuencias desordenadas. (1, 2, 3, 4, 5, 6).

- **Ordenación no natural:** Tarda lo mínimo posible cuando la entrada está inversamente ordenada. (6, 5, 4, 3, 2, 1).
- **Por estabilidad:** un ordenamiento estable mantiene el orden relativo que tenían originalmente los elementos con claves iguales. Por ejemplo, si una lista ordenada por fecha se reordena en orden alfabético con un algoritmo estable, todos los elementos cuya clave alfabética sea la misma quedarán en orden de fecha. Otro caso sería cuando no interesan las mayúsculas y minúsculas, pero se quiere que si una clave aBC estaba antes que AbC, en el resultado ambas claves aparezcan juntas y en el orden original: aBC, AbC. Cuando los elementos son indistinguibles (porque cada elemento se ordena por la clave completa) la estabilidad no interesa. Los algoritmos de ordenamiento que no son estables se pueden implementar para que sí lo sean. Una manera de hacer esto es modificar artificialmente la clave de ordenamiento de modo que la posición original en la lista participe del ordenamiento en caso de coincidencia.

Ejemplo 7 , 6, 8, 9, 1, 6 -> 1, 6, 6, 7, 8, 9 (Estable)

Características de los algoritmos de ordenación:

Complejidad computacional Big O

La notación Big O define como de bien un algoritmo escala cuando los datos que se procesan aumentan. Por ejemplo cómo difiere la ejecución si como entrada nuestro algoritmo tiene 100 o 10,000 elementos de entrada. Por lo tanto no tiene que ver con la velocidad de ejecución sino con lo bien que escala.

Ejemplo: 45^N3 + 20N^2 + 19

(N =1) –> 45 +20 +19 = 84

(N =2) –> 360 +80 +19 = 459

(N =10) –> 45.000 + 2.000 +19 = 47,019

O(1) –> Se ejecuta en el mismo tiempo independientemente del número de elementos que reciba como entrada.

O(N) –> El tiempo en ejecutarse va a crecer de forma proporcional al aumento de los datos de entrada. Un buen ejemplo es la búsqueda secuencial. Si tenemos más datos más tardará recorrerlos de forma lineal.

O(N^2) –> El tiempo en ejecutarse será proporcional al cuadrado de la cantidad de elementos que reciba como entrada. Esto suele ocurrir con algoritmos con bucles dentro de bucles. Por ejemplo BubbleSort

O(log N) –> El tiempo en ejecutarse no difiere tanto en función de el tamaño. Por ejemplo en BinarySort.

Log10(1)=0 , Log10(1000)=3, Log10(10000)=4

Uso de memoria y otros recursos computacionales

También se usa la notación O(n).

Los algoritmos de ordenación que vamos a estudiar son:

Nombre	Tipo	Complejidad	Memoria	Método
BubbleSort	Estable	O(n^2)	O(1)	Intercambio
BucketSort	Estable	O(n)	O(n)	No comparativo
BinaryTreeSort	Estable	O(n log n)	O(n)	Inserción
QuickSort	Inestable	Promedio: O(n log n), peor caso: O(n^2)	O(log n)	Partición

Clase Big O Notaion

```
// Big O notation is a way to measure how well a
// computer algorithm scales as the amount of data
// involved increases. It is not always a measure
// of speed as you'll see below

// This is a rough overview of Big O and doesn't
// cover topics such as asymptotic analysis, which
// covers comparing algorithms as data approaches
// infinity

// What we are defining is the part of the algorithm
// that has the greatest effect. For example
// 45n^3 + 20n^2 + 19 = 84 (n=1)
```

```
// 45n^3 + 20n^2 + 19 = 459 (n=2) Does 19 matter?
// 45n^3 + 20n^2 + 19 = 47019 (n=10)
// Does the 20n^2 matter if 45n^3 = 45,000?
// Has an O(n^3) Order of n-cubed

public class BigONotation {

	private int[] theArray;

	private int arraySize;

	private int itemsInArray = 0;

	static long startTime;

	static long endTime;

	public static void main(String[] args) {

		/*
		 * 0(1) Test BigONotation testAlgo = new BigONotation(10);
		 *
		 * testAlgo.addItemToArray(10);
		 *
		 * System.out.println(Arrays.toString(testAlgo.theArray));
		 */

		BigONotation testAlgo2 = new BigONotation(100000);
		testAlgo2.generateRandomArray();

		BigONotation testAlgo3 = new BigONotation(200000);
		testAlgo3.generateRandomArray();

		BigONotation testAlgo4 = new BigONotation(30000);
		testAlgo4.generateRandomArray();
```

```
BigONotation testAlgo5 = new BigONotation(400000);
testAlgo5.generateRandomArray();

/*
 * O(N) Test
 *
 * testAlgo2.linearSearchForValue(20);
 *
 * testAlgo3.linearSearchForValue(20);
 *
 * testAlgo4.linearSearchForValue(20);
 *
 * testAlgo5.linearSearchForValue(20);
 */

// O(N^2) Test
/*
 * testAlgo2.bubbleSort();
 *
 * testAlgo3.bubbleSort();
 *
 * testAlgo4.bubbleSort();
 *
 * // 0 (log N) Test
 *
 * testAlgo2.binarySearchForValue(20);
 * testAlgo3.binarySearchForValue(20);
 */

// O (n log n) Test

startTime = System.currentTimeMillis();

testAlgo2.quickSort(0, testAlgo2.itemsInArray);

endTime = System.currentTimeMillis();
```

```
            System.out.println("Quick Sort Took " + (endTime - start-
Time));

      }

      // O(1) An algorithm that executes in the same
      // time regardless of the amount of data
      // This code executes in the same amount of
      // time no matter how big the array is

      public void addItemToArray(int newItem) {

            theArray[itemsInArray++] = newItem;

      }

      // 0(N) An algorithm thats time to complete will
      // grow in direct proportion to the amount of data
      // The linear search is an example of this

      // To find all values that match what we
      // are searching for we will have to look in
      // exactly each item in the array

      // If we just wanted to find one match the Big O
      // is the same because it describes the worst
      // case scenario in which the whole array must
      // be searched

      public void linearSearchForValue(int value) {

            boolean valueInArray = false;
            String indexsWithValue = "";

            startTime = System.currentTimeMillis();
```

```
            for (int i = 0; i < arraySize; i++) {

                  if (theArray[i] == value) {
                        valueInArray = true;
                        indexsWithValue += i + " ";
                  }

            }

            System.out.println("Value Found: " + valueInArray);

            endTime = System.currentTimeMillis();

            System.out.println("Linear Search Took " + (endTime - start-
Time));

      }

      // O(N^2) Time to complete will be proportional to
      // the square of the amount of data (Bubble Sort)
      // Algorithms with more nested iterations will result
      // in O(N^3), O(N^4), etc. performance

      // Each pass through the outer loop (0)N requires us
      // to go through the entire list again so N multiplies
      // against itself or it is squared

      public void bubbleSort() {

            startTime = System.currentTimeMillis();

            for (int i = arraySize - 1; i > 1; i--) {

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

```
                    if (theArray[j] > theArray[j + 1]) {

                        swapValues(j, j + 1);

                    }
                }
            }

            endTime = System.currentTimeMillis();

            System.out.println("Bubble Sort Took " + (endTime - start-
Time));
        }

        // O (log N) Occurs when the data being used is decreased
        // by roughly 50% each time through the algorithm. The
        // Binary search is a perfect example of this.

        // Pretty fast because the log N increases at a dramatically
        // slower rate as N increases

        // O (log N) algorithms are very efficient because increasing
        // the amount of data has little effect at some point early
        // on because the amount of data is halved on each run through

        public void binarySearchForValue(int value) {

            startTime = System.currentTimeMillis();

            int lowIndex = 0;
            int highIndex = arraySize - 1;

            int timesThrough = 0;

            while (lowIndex <= highIndex) {
```

```
                int middleIndex = (highIndex + lowIndex) / 2;

                if (theArray[middleIndex] < value)
                      lowIndex = middleIndex + 1;

                else if (theArray[middleIndex] > value)
                      highIndex = middleIndex - 1;

                else {

                      System.out.println("\nFound a Match for " + value
                                  + " at Index " + middleIndex);

                      lowIndex = highIndex + 1;

                }

                timesThrough++;

          }

          // This doesn't really show anything because
          // the algorithm is so fast

          endTime = System.currentTimeMillis();

          System.out.println("Binary Search Took " + (endTime - start-
Time));

          System.out.println("Times Through: " + timesThrough);

     }

     // O (n log n) Most sorts are at least O(N) because
     // every element must be looked at, at least once.
     // The bubble sort is O(N^2)
```

```
// To figure out the number of comparisons we need
// to make with the Quick Sort we first know that
// it is comparing and moving values very
// efficiently without shifting. That means values
// are compared only once. So each comparison will
// reduce the possible final sorted lists in half.
// Comparisons = log n! (Factorial of n)
// Comparisons = log n + log(n-1) + .... + log(1)
// This evaluates to n log n

public void quickSort(int left, int right) {

      if (right - left <= 0)
            return;

      else {

            int pivot = theArray[right];

            int pivotLocation = partitionArray(left, right, pivot);

            quickSort(left, pivotLocation - 1);
            quickSort(pivotLocation + 1, right);

      }

}

public int partitionArray(int left, int right, int pivot) {

      int leftPointer = left - 1;
      int rightPointer = right;

      while (true) {

            while (theArray[++leftPointer] < pivot)
```

```
                ;
            while (rightPointer > 0 && theArray[--rightPointer] >
pivot)
                ;
            if (leftPointer >= rightPointer) {
                break;
            } else {
                swapValues(leftPointer, rightPointer);
            }
        }
        swapValues(leftPointer, right);
        return leftPointer;
    }
    BigONotation(int size) {
        arraySize = size;
        theArray = new int[size];
    }
    public void generateRandomArray() {
        for (int i = 0; i < arraySize; i++) {
```

```
                theArray[i] = (int) (Math.random() * 1000) + 10;

            }

            itemsInArray = arraySize - 1;

        }

        public void swapValues(int indexOne, int indexTwo) {

            int temp = theArray[indexOne];
            theArray[indexOne] = theArray[indexTwo];
            theArray[indexTwo] = temp;

        }

}
```

4.1. BUBBLE SORT

Es probablemente de los algoritmos más sencillos. El algoritmo realiza tantas iteraciones como sean necesarias comparando los elementos adyacentes. En caso de dos elementos adyacentes no estén ordenados, entonces los permutará.

El siguiente ejemplo muestra la evolución de los elementos a ordenar:

{ 5 , 2 , 18 , 15 , 22 }

5	2	18	15	22
5	2	18	15	22
2	5	18	15	22
2	5	18	15	22
2	5	15	18	22

```
public class BubbleSort {

      public static void Sort(int[] num)
      {
           int j;
           boolean flag = true; // set flag to true to begin first pass
           int temp;

           while (flag)
           {
            flag= false;
            for( j=0;  j < num.length -1;  j++ )
            {
               if ( num[ j ] > num[j+1] )
               {
                   temp = num[j];
                   num[j] = num[j+1];
                   num[j+1] = temp;
                   flag = true;
               }
               System.out.println(Arrays.toString(num));
            }
           }
      }

    public static void main(String[] args) {
        int[] input = { 5, 2, 18, 15, 22  };
        Sort(input);

}
```

4.2. BUCKET SORT

El ordenamiento por casilleros (bucket sort o bin sort, en inglés) es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos. Después cada uno de esos casilleros se ordena

individualmente con otro algoritmo de ordenación (que podría ser distinto según el casillero), o se aplica recursivamente este algoritmo para obtener casilleros con menos elementos.

El algoritmo contiene los siguientes pasos:

1. Crear una colección de casilleros vacíos
2. Colocar cada elemento a ordenar en un único casillero
3. Ordenar individualmente cada casillero
4. Devolver los elementos de cada casillero concatenados por orden

```
import java.util.*;

public class BucketSort{

   public static void sort(int[] a, int maxVal) {
      int [] bucket=new int[maxVal+1];


      for (int i=0; i<a.length; i++) {
         bucket[a[i]]++;
      }

      int outPos=0;
      for (int i=0; i<bucket.length; i++) {
         for (int j=0; j<bucket[i]; j++) {
            a[outPos++]=i;
         }
      }
   }


   public static void main(String[] args) {
      int maxVal=22;
      int [] data = {5,2,18,15,22};
```

```
        System.out.println("Before:  " + Arrays.toString(data));
        sort(data,maxVal);
        System.out.println("After:   " + Arrays.toString(data));
    }

}
```

4.3. BINARY TREE SORT

Este algoritmo se estudió en la primera parte de la asignatura. Se basa en la creación de un árbol binario introduciendo los elementos ya ordenados. Después, se puede recorrer el árbol en inorden.

4.4. QUICK SORT

Este algoritmo está basado en la técnica de divide y vencerás. Permite ordenar n elementos en un tiempo proporcional a n log n.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento de la lista 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.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- 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 O(n·log n).
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.

```
import java.util.Arrays;

public class QuickSort {

	public static void main(String[] args) {
		int[] data = {5,2,18,15,22};

		System.out.println("Before: " + Arrays.toString(data));
		int left = 0;
		int right = data.length-1;

		quickSort(left, right,data);
		System.out.println("After: " + Arrays.toString(data));

	}


	// This method is used to sort the array using quicksort algorithm.
	// It takes the left and the right end of the array as the two cur-
sors.
	private static void quickSort(int left,int right,int a[]){

		// If both cursor scanned the complete array quicksort exits
		if(left >= right)
			return;

		// For the simplicity, we took the right most item of the
array as a pivot
```

```
            int pivot = a[right];
            int partition = partition(left, right, pivot,a);

            // Recursively, calls the quicksort with the different left
and right parameters of the sub-array
            quickSort(0, partition-1,a);
            quickSort(partition+1, right,a);
      }

      // This method is used to partition the given array and returns the
integer which points to the sorted pivot index
      private static int partition(int left,int right,int pivot,int a[]){
            int leftCursor = left-1;
            int rightCursor = right;
            while(leftCursor < rightCursor){
                while(a[++leftCursor] < pivot);
                while(rightCursor > 0 && a[--rightCursor] > pivot);
                 if(leftCursor >= rightCursor){
                       break;
                 }else{
                       swap(leftCursor, rightCursor,a);
                 }
            }
            swap(leftCursor, right,a);
            return leftCursor;
      }

      //swap the values between the two given index
      public static void swap(int left,int right,int a[]){
            int temp = a[left];
            a[left] = a[right];
            a[right] = temp;
      }

}
```

Otro ejemplo:

```
public class QuickSort {

    private int array[];
    private int length;

    public void sort(int[] inputArr) {

        if (inputArr == null || inputArr.length == 0) {
            return;
        }
        this.array = inputArr;
        length = inputArr.length;
        quickSort(0, length - 1);
    }

    private void quickSort(int lowerIndex, int higherIndex) {

        int i = lowerIndex;
        int j = higherIndex;
        // calculate pivot number, I am taking pivot as middle index num-
ber
        int pivot = array[lowerIndex+(higherIndex-lowerIndex)/2];
        // Divide into two arrays
        while (i <= j) {
            /**
            * In each iteration, we will identify a number from left side
which
             * is greater then the pivot value, and also we will identify
a number
             * from right side which is less then the pivot value. Once
the search
             * is done, then we exchange both numbers.
             */
            while (array[i] < pivot) {
                i++;
```

```
            }
            while (array[j] > pivot) {
                j--;
            }
            if (i <= j) {
                exchangeNumbers(i, j);
                //move index to next position on both sides
                i++;
                j--;
            }
        }
        // call quickSort() method recursively
        if (lowerIndex < j)
            quickSort(lowerIndex, j);
        if (i < higherIndex)
            quickSort(i, higherIndex);
    }

    private void exchangeNumbers(int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String a[]){

        QuickSort sorter = new QuickSort();
        int[] input = {24,2,45,20,56,75,2,56,99,53,12};
        sorter.sort(input);
        for(int i:input){
            System.out.print(i);
            System.out.print(" ");
        }
    }

}
```

4.5. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Realizar un programa que ordene una lista con nombres de personas utilizando el algoritmo Bubble Sort. {“Pedro”, “Maria del Mar”, “Mr John Smith Kalgary”}.
2. Realizar el ejercicio anterior usando el algoritmo Bucket Sort.
3. Realizar el ejercicio anterior usando el algoritmo Quick Sort.
4. Comparar la ejecución de los programas anteriores para ver las diferencias en tiempo de ejecución.

Capítulo 5

HASHING

Hashsing es el proceso de transformación de una estructura de datos a un string que va a representar el acceso a esa información. Hashing se utiliza como mecanismo para indexar y recuperar información de una manera mucho más rápida y eficiente.

La técnica de hashing se aplica a multitud de soluciones software, como es el caso de base de datos relacionales, ficheros, motores de búsqueda etc.

El proceso se desarrolla de la siguiente forma. Imaginémonos que tenemos una lista de nombres almacenadas en una base de datos. Por ejemplo “Mariano García”, “Peter Seller”, “John Morgan”. Si quisiéramos buscar un nombre concreto, tendríamos que buscar carácter por carácter en nuestra lista de nombre el nombre de entrada. Cómo se puede ver, este proceso sería lento e ineficiente. Imaginémonos ahora que los nombres fueran convertidos a través de una función hash a una clave única de 4 caracteres.

Sería mucho más rápido buscar esa única clave de 4 caracteres que hacerlo en sus strings originales.

Ejemplo paso a paso:

Los strings se transforman usando la función hash.

Nombre	Hash
Mariano García	8965
Peter Seller	7654
John Morgan	1698

- El valor a buscar se transforma usando la misma función hash. Ejemlo “Peter Seller” –> 7654
- La búsqueda se ejecutará. comparando 4 dígitos que es algo mucho más rápido que strings aleatorios.

5.1. FUNCIONES HASH

Existe un gran número de funciones hash en la actualidad. Prácticamente cualquier programador puede construir una función hash basándose en sus necesidades concretas. Hay dos reglas básicas que se deben de seguir para construir una buena función hash.

- *La función debe ser sencilla y su cálculo lo más rápido posible. Si el cálculo no es rápido entonces esto afectará el rendimiento global del proceso.*
- *La generación del resultado de la función hash debe estar distribuido. Esto va a evitar colisiones.*

A continuación vamos a ver las funciones hash más comunes:

Truncamiento

Este caso es bastante sencillo, lo que se hace es ignorar parte de la estructura inicial. Habría que decidir que rango va a tener la clave. En este caso en rango es de 0 a 9999, ya que vamos a utilizar una clave-key de 4 dígitos. En este caso elegimos las posiciones 1,4,6,8.

Clave: 76258910 –> **7590**

Este algoritmo tiene una gran desventaja ya que puede generar colisiones. Cuando hay colisiones se requiere de un proceso adicional que limita la eficiencia del mismo.

Plegamiento

Consiste en dividir el valor en diferentes partes. Luego esas partes se combinan mediante una operación para conformar el índice. Por ejemplo tomada la clave se puede dividir en grupos de dos dígitos y se suman despreciando el primer dígito para dar el valor del índice.

Clave: 83491231 -> 83 + 49 + 12 + 31 -> 175 –> **75**

Clave: 53461459-> 53 + 46 + 14 + 59 -> 172 –> **72**

Genera colisiones

Aritmética Modular

Este método es probablemente el más común. Consiste en convertir la clave a un entero, resultado de dividir pro el tamaño del rango del índice y tomar el resto de la división entera. La expresión es **H(Clave) = Clave mod vector**

Vector: 100 posiciones, Clave:12325 -> H(12345) = 12345 mod 100 =

5.2. COLISIONES

La colisión se produce cuando para más de una clave se tiene el mismo valor asociado al aplicar el hashing. Este se produce porque el número de claves posibles es mayor al número de índices del vector de almacenamiento.

Vamos a intentar entenderlo usando los DNI de personas. Imaginémonos que tenemos un vector de 100 posiciones. Usando **H(Clave) = Clave mod (vector +1)**.

DNI	H(Clave)
52122985	16
13307255	0
42789566	7
22795514	16

Por ello es importante dimensionar usando una capacidad que sea suficiente para que no se produzcan colisiones. Para solucionar el problema de las colisiones existen diferentes alternativas y normalmente son bastante costosas. Vamos a ver dos de ellos.

Arrays anidados

Cada elemento del array tiene asociado un nuevo array. En este nuevo array se irán almacenando los elementos colisionados. Es sencilla pero ineficiente. En este caso tendrá un coste de memoria para almacenar los nuevos arrays asociados a cada elemento.

Encadenamiento

En este caso cada elemento del array apunta a una lista que almacena los valores colisionados.

5.3. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Implementar una función hash por
   - Truncamiento.
   - Plegamiento.
   - Aritmética modular.
2. Realizar un método que facilite la posición que ocupará un número entero al aplicar h(clave) = (clave mod 101).
3. Implementar una clase que contiene DNI, Nombre y Apellidos como variables de tipo String. Crear un método getHashCode() que sobreescriba el por defecto de java. La lógica de getHashCode() será la de devolver un número que se genera a partir de los datos de tipo String. Usar para ello la aritmética modular n%101. Implementar otra clase que gestione el almacenamiento de estos objetos en una estructura de datos basándonos en sus claves hash. Tener en cuenta que hay que resolver colisiones.

```
public class Person {

        private String dni;
        private String name;
```

```
        private String surname;

        Person(String dniStr, String nameStr, String sur-
    nameStr){
            this.dni = dniStr;
            this.name = nameStr;
            this.surname =surnameStr;
        }

        @Override
        public int hashCode(){
            int value = this.dni.hashCode() + this.name.hash-
            Code() + this.surname.hashCode();
            return Math.abs(value % 101);
        }
    }
```



```
import java.util.ArrayList;

public class MyHashTable {

    ArrayList[] list = new ArrayList[101];

    public void Add(Person p){
        if (list[p.hashCode()] == null){
            list[p.hashCode()] = new ArrayList<>();
            list[p.hashCode()].add(p);
        }
        else
            list[p.hashCode()].add(p);

    }
}
```

```
import org.junit.Test;

public class MyHashTableTest {

	@Test
	public void TestOneObject(){
		Person p = new Person("1255X", "Juan", "Prieto");
		MyHashTable hash = new MyHashTable();
		hash.Add(p);
	}

	@Test
	public void TestCollisions(){
		Person p = new Person("1255X", "Juan", "Prieto");
		Person p2 = new Person("1255X", "Juan", "Prieto");
		MyHashTable hash = new MyHashTable();
		hash.Add(p);
		hash.Add(p2);
	}

}
```


Time complexity in big O notation	
Algorithm	**Average**
Space	O(n)
Search	O(1)
Insert	O(1)

4. Desarrollar un programa que emule un sistema gestor de bases de datos utilizando técnicas de hashing. El sistema almacenará registros de personas en ficheros. Cada persona constará de DNI, nombre, apellidos. Utilizaremos la aritmética modular como técnica de hashing para su almacenamiento.
5. Como segundo paso del ejercicio 1 se desarrollará una opción de búsqueda para un DNI dado por entrada.

Capítulo 6

APLICACIONES DE BÚSQUEDA

En el siguiente capítulo vamos a desgranar una aplicación más real de todo lo estudiado hasta el momento. Dentro del mundo de la programación hay un sector que cada vez está tomando mayor relevancia. Este es el campo de las aplicaciones de búsqueda. Una de las razones principales es la ingente cantidad de datos que se generar diariamente y las necesidades de poder tener accesibilidad a esa información de la mejor manera posible.

Ya no sólo es importante ser capaz de encontrar la información buscada. Ahora también es crucial devolver los resultados que el usuario necesita y que por lo tanto sean relevantes con respecto a sus expectativas.

Aunque parezca algo no demasiado complejo, en realidad si lo es y supone uno de los campos dentro de la programación que más ha evolucionado en los últimos años.

El uso de las estructuras de datos y algoritmos ha resultado crucial para poder resolver parte de los desafíos generados por este mundo de las aplicaciones de búsqueda.

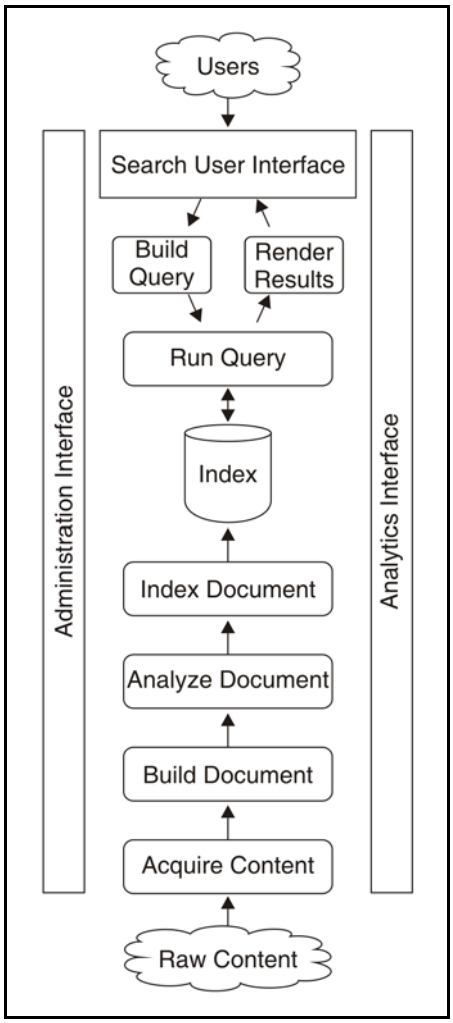
El principal objetivo de este capítulo es el entender cómo un problema real que se plantea desde el punto de vista del usuario, se traslada al mundo de la computación y como a su vez se resuelve usando estructuras de datos y algoritmia.

Vamos a ver primero los principales componentes que componen una aplicación de búsqueda:

1. Proceso de adquisición de la información
2. Proceso de indexado.
3. Proceso de query y recuperación.

- El **proceso de la adquisición** de la información, corresponde a la identificación de la información que se va a procesar para poder ser recuperada más adelante gracias a la búsqueda. Esta información se puede encontrar almacenada en diversos lugares y formatos. En lugares, puede ser en un DFS (Distributed file system), en una uri en internet, en un disco duro etc. Por otro lado el formato, que puede variar desde un Word, pdf, html. Todo esto es importante a la hora de identificar el denominado proceso de adquisición.
- El **proceso de indexado**. En esta parte los datos una vez recuperados se transforman a estructuras de datos que van a ser almacenadas en una forma totalmente dirigida a la recuperación y comparación de las mismas. En este caso normalmente la información se almacena en forma de índices. Más adelante veremos esto en más detalle.
- El **proceso de query o consulta y recuperación**. En este caso una vez que la información ha sido identificada, extraída e indexada toca la parte referente a cómo encontrarla y recuperarla. Para poder encontrarla es necesario que se busque en la misma forma en la que se indexa.

Ahora vamos a ver un diagrama de este proceso:



6.1. CREANDO UN ÍNDICE

Lo primero que vamos a intentar entender es que es un índice. Al igual que en los libros un índice es una lista de palabras o frases que permiten ubicar la información de un material interior de manera rápida. En el caso de los libros, las palabras están asociadas a la página donde se encuentran.

En el siguiente ejemplo queda claramente explicado:

Palabra	Página
El	5,15,25,65,78
Casa	8,15,25,69,85

El caso de los índices como mecanismos de búsqueda, funcionan conceptualmente de la misma manera. Existe una estructura de datos que contiene información referente a en que sitio se encuentra que datos en concreto.

Si en vez de utilizar índices usáramos una búsqueda secuencial a través de todo el contenido que queremos buscar, es claro entender que la eficiencia sería terrible.

Un término muy frecuentemente utilizado dentro de las aplicaciones de búsqueda son los denominadas índices invertidos o inverted index. En el caso de los índices invertidos, podemos verlo como un conjunto de términos donde cada término apunta al documento en el que aparece.

Vamos a ver a continuación un conjunto de documentos que se van a convertir a un índice invertido.

Documento	Contenido
1	La guerra del Golfo
2	El chico era un golfo
3	El bien y el mal

Índice invertido – Inverted Index

Term	Documento
La	1
Guerra	1
Del	1
Golfo	1,2
El	2,3
Chico	2
Era	2
Un	2
Bien	3
Y	3
Mal	3

Como se puede ver en la tabla, es una sencilla representación de las palabras encontradas en los documentos y en que documentos aparecen. Ahora vamos a implementar una estructura de datos que nos permita almacenar esa información.

Existen diferentes opciones para almacenar un índice invertido. Vamos a empezar por un ejemplo sencillo e ir haciéndolo más complejo a medida que se complique nuestra aplicación. Para ello vamos a empezar usando una estructura datos en java denominada HashMap. Es muy parecida a HasTable.

Empezamos por la definición de HashMap: Es una estructura de datos que implementa Map interface. Por lo tanto su implementación provee de todas las operaciones de Map. Esta estructura de datos se base en pares de key, value. Se representa como HashMap<K,V>.

Su completa descripción, así como sus métodos y propiedades las podemos encontrar en muchas distribuciones de java. Un ejemplo puede ser la de Oracle:

https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

GrepCode ayuda a entender y explorer la implementación.

http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/HashMap.java#HashMap

Vamos a pasar ahora a utilizar HashMap<K,V> para crear nuestro índice invertido. Primero vamos a ver los métodos que tendríamos que implementar en caso de codificar nosotros mismos la implementación.

```
package index.atlantico.es;

import java.util.Collection;
import java.util.Map;
import java.util.Set;

public class InvertedIndex implements Map<String, int[]> {

	@Override
	public void clear() {
		// TODO Auto-generated method stub

	}
```

```
@Override
public boolean containsKey(Object key) {
	// TODO Auto-generated method stub
	return false;
}

@Override
public boolean containsValue(Object value) {
	// TODO Auto-generated method stub
	return false;
}

@Override
public Set<java.util.Map.Entry<String, int[]>> entrySet() {
	// TODO Auto-generated method stub
	return null;
}

@Override
public int[] get(Object key) {
	// TODO Auto-generated method stub
	return null;
}

@Override
public boolean isEmpty() {
	// TODO Auto-generated method stub
	return false;
}

@Override
public Set<String> keySet() {
	// TODO Auto-generated method stub
	return null;
}

@Override
public int[] put(String key, int[] value) {
	// TODO Auto-generated method stub
	return null;
}
```

```
	@Override
	public void putAll(Map<? extends String, ? extends int[]> m) {
		// TODO Auto-generated method stub

	}

	@Override
	public int[] remove(Object key) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public int size() {
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public Collection<int[]> values() {
		// TODO Auto-generated method stub
		return null;
	}
```



```
}
```

Ahora vamos a ver un ejemplo donde simplemente usamos HashMap.

```
package index.atlantico.es;
```

```
package index.atlantico.es;

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

public class InvertedIndex {
```

```
	public Map<String, int[]> index = null;

	InvertedIndex(){
		this.index = new HashMap<String,int[]>();
	}


}
```

Insertamos los datos del índice invertido en nuestra estructura de datos.

```
package index.atlantico.es;

import org.junit.Assert;
import org.junit.Test;


public class InvertedIndexTest {

	@Test //Prueba insertando Key,value pairs
	public void InsertarKeyValue() {
		InvertedIndex inverIndex = new InvertedIndex();
		inverIndex.index.put("La", new int[] {1});
		inverIndex.index.put("Guerra", new int[] {1});
		inverIndex.index.put("Del", new int[] {1});
		inverIndex.index.put("Golfo", new int[] {1,2});
		inverIndex.index.put("El", new int[] {2,3});
		inverIndex.index.put("Chico", new int[] {2});
		inverIndex.index.put("Era", new int[] {2});
		inverIndex.index.put("Un", new int[] {2});
		inverIndex.index.put("Bien", new int[] {3});
		inverIndex.index.put("Y", new int[] {3});
		inverIndex.index.put("Mal", new int[] {3});

		Assert.assertEquals(inverIndex.index.size(), 11);

	}

}
```

El siguiente paso es buscar la información que hemos almacenado. Para ello tendremos que encontrar en nuestra estructura de datos todos los documentos que hacen referencia a un determinado término.

Vamos a poner como ejemplo que queremos encontrar los documentos donde se hace referencia al término "Golfo.". En primer lugar tomaremos el término como entrada, en vez de buscar a través de todos los documentos de forma secuencial lo haremos usando el índice que hemos generado anteriormente. El siguiente ejemplo busca en un índice invertido.

```
package index.atlantico.es;

import org.junit.Assert;
import org.junit.Test;


public class InvertedIndexTest {

	@Test //Prueba insertando Key,value pairs
	public void InsertarKeyValue() {
		InvertedIndex inverIndex = new InvertedIndex();
		InsertarDatos(inverIndex);
		Assert.assertEquals(inverIndex.index.size(), 11);
	}

	@Test //Prueba insertando Key,value pairs
	public void BuscarKeyValue() {
		InvertedIndex inverIndex = new InvertedIndex();
		InsertarDatos(inverIndex);

		int[] value = inverIndex.index.get("Golfo");

		Assert.assertEquals(value.length, 2);
	}

	public void InsertarDatos(InvertedIndex inverIndex){
		inverIndex.index.put("La", new int[] {1});
		inverIndex.index.put("Guerra", new int[] {1});
```

```
            inverIndex.index.put("Del", new int[] {1});
            inverIndex.index.put("Golfo", new int[] {1,2});
            inverIndex.index.put("El", new int[] {2,3});
            inverIndex.index.put("Chico", new int[] {2});
            inverIndex.index.put("Era", new int[] {2});
            inverIndex.index.put("Un", new int[] {2});
            inverIndex.index.put("Bien", new int[] {3});
            inverIndex.index.put("Y", new int[] {3});
            inverIndex.index.put("Mal", new int[] {3});
      }

}
```

6.2. APLICAR RELEVANCIA USANDO TF-IDF

Hasta este momento hemos analizado como funciona una aplicación de búsqueda. Cuáles son las diferentes partes que componen una aplicación de búsqueda y dentro de la recuperación de la información ahora vamos a ver cómo definir como de relevante es la información recuperada.

Del inglés term frequency – inverse document frequency. Su traducción sería frecuencia del término – frecuencia inversa del documento. Este es un algoritmo que ha formado parte de las tecnologías de la recuperación por mucho tiempo. Ha sido objeto de investigación en innumerables ocasiones y como resultado se ha obtenido uno de los algoritmos más famosos que existen en lo que se refiere a la definición de relevancia de los resultados a buscar.

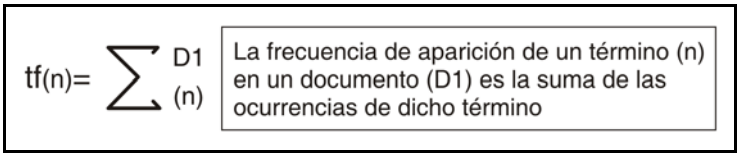
Tiene en cuenta el número de veces que una palabra aparece en un documento y se compensa con la frecuencia de esa palabra en el conjunto de documentos.

Si queremos verlo de forma práctica, imaginémonos que queremos determinar los documentos más relevantes para el texto a buscar “programador junior”. Lo primero es eliminar de los resultados los documentos que no contengas las palabras “programdor” y “junior”. Para mejorar los resultados a devolver, lo primero es contar el número de veces que cada palabra “programador” y “junior” aparece en el documento. A mayor número de ocurrencias se le dará mayor relevancia. Por otro lado se cuenta la aparición de esas palabras pero de forma global en el conjunto de todos los documentos. En este caso a mayor frecuencia se le dará una penalización.

A continuación vamos a ver la implementación matemática de las dos funciones.

TF – Term frequency

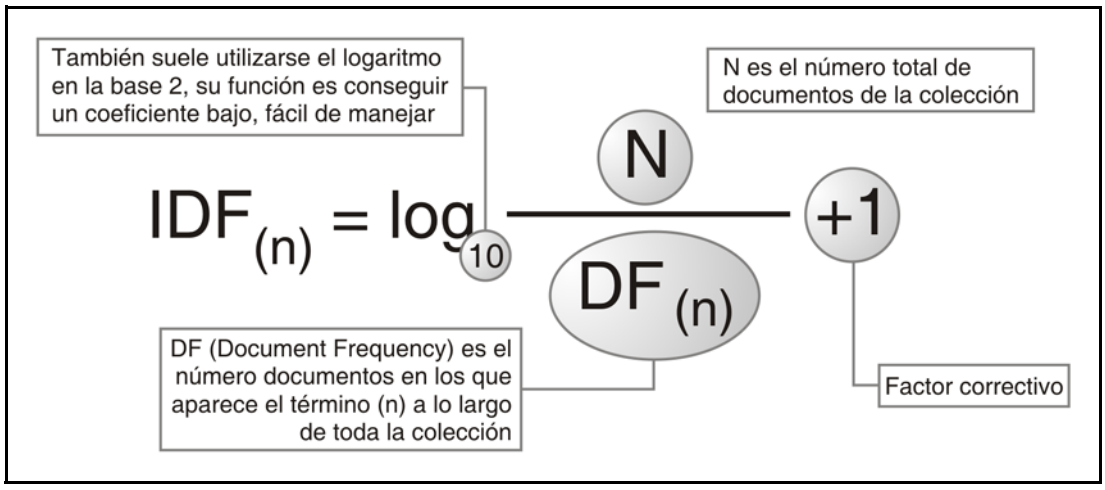
Esta es la frecuencia de aparición de un término en un documento. Esta frecuencia es relativa a un documento y no a toda la colección.



IDF – Inverse document frequency

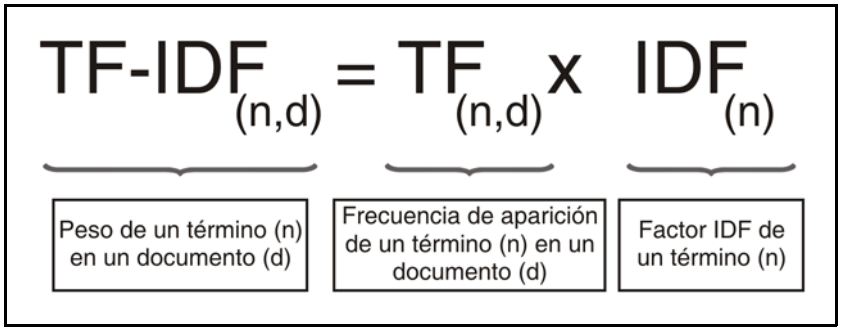
Esta es la frecuencia inversa del documento para el término. Lo que quiere decir esto es que este factor es inversamente proporcional al número de documentos en los que aparece dicho término. Por lo tanto cuanto menor sea el número de documentos en los que aparezca el término mayor será su IDF y al revés si el número de documentos que contienen el término, menor será su valor. A este coeficiente también se le llama capacidad discriminatoria de un término con respecto a la colección de documentos.

El factor IDF es único para cada término de la colección. Esto significa que su cálculo, (véase siguiente figura), el IDF de un término dado (n) se realiza aplicando el logaritmo en base 10 de N (Número total de documentos de la colección) dividido entre la "Frecuencia de documentos para un término (n) en la colección" (o lo que es lo mismo el número de documentos de la colección en los que aparece el término (n) dado). Al valor resultante se le suma 1 para corregir los valores para los términos con IDF muy bajos (Aunque esta variación depende del sistema de recuperación).



Ponderación TF-IDF

Por lo tanto el peso de un documento es el producto de su frecuencia de aparición y su frecuencia inversa de documento.



La siguiente tabla representa un ejemplo de estos valores:

Frecuencia de aparición de los términos TF			
Término	**Doc1**	**Doc2**	**Doc3**
biblioteca	27	4	24
archivo	3	33	0
documento	14	0	17
museo	0	33	29
Cálculo de Pesos TF-IDF			
biblioteca	$TF\text{-}IDF_{(biblioteca,Doc1)}$	$TF\text{-}IDF_{(biblioteca,Doc2)}$	$TF\text{-}IDF_{(biblioteca,Doc3)}$
	27 x 2,65 = 71,55	4 x 2,65 = 10,60	24 x 2,65 = 63,60
archivo	$TF\text{-}IDF_{(archivo,Doc1)}$	$TF\text{-}IDF_{(archivo,Doc2)}$	$TF\text{-}IDF_{(archivo,Doc3)}$
	3 x 3,08 = 9,24	33 x 3,08 = 101,64	0 x 3,08 = 0
documento	$TF\text{-}IDF_{(documento,Doc1)}$	$TF\text{-}IDF_{(documento,Doc2)}$	$TF\text{-}IDF_{(documento,Doc3)}$
	14 x 2,50 = 35	0 x 2,50 = 0	17 x 2,50 = 42,50
museo	$TF\text{-}IDF_{(museo,Doc1)}$	$TF\text{-}IDF_{(museo,Doc2)}$	$TF\text{-}IDF_{(museo,Doc3)}$
	0 x 2,62 = 0	33 x 2,62 = 86,46	29 x 2,62 = 75,98

6.3. EJERCICIOS DE EVALUACIÓN CONTINUA

1. Crear un programa que dado un conjunto de documentos los procese y los almacene en una estructura de datos de tipo índice invertido.
2. Implementar un método que calcule el TF de un término dado en un documento.

```
package tdidf.atlantico.es;

import java.util.List;

public class TermFrequency {

	public static double tf(List<String> doc, String term) {
	    double result = 0;
	    for (String word : doc) {
	       if (term.equalsIgnoreCase(word))
	              result++;
	       }
	    return result / doc.size();
	}

}
```

3. Implementar un método que calcule el IDF

```
package tdidf.atlantico.es;

import java.util.List;

public class InverseTermFrequency {

	public static double idf(List<List<String>> docs, String term) {
	    double n = 0;
	    for (List<String> doc : docs) {
	        for (String word : doc) {
	            if (term.equalsIgnoreCase(word)) {
	                n++;
	                break;
	            }
	        }
	    }
	    return Math.log(docs.size() / n);
	}

}
```

4. Implementar un método que calcule TF-IDF.

```
package tdidf.atlantico.es;

import java.util.List;

public class TfIdf {

   public static double tfIdf(List<String> doc, List<List<String>>
docs, String term) {
         return TermFrequency.tf(doc, term) * InverseTermFre-
         quency.idf(docs, term);
   }
}
```

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

   List<String> doc1 = Arrays.asList("Lorem", "ipsum", "dolor", "ipsum",
"sit", "ipsum");
   List<String> doc2 = Arrays.asList("Vituperata", "incorrupte", "at",
"ipsum", "pro", "quo");
   List<String> doc3 = Arrays.asList("Has", "persius", "disputationi",
"id", "simul");
   List<List<String>> documents = Arrays.asList(doc1, doc2, doc3);


   double tfidf = TfIdf.tfIdf(doc1, documents, "ipsum");
   System.out.println("TF-IDF (ipsum) = " + tfidf);


}
```

5. Investigar que es BM25 y presentarlo en clase para los demás alumnos.

Capítulo 7

APLICACIONES DISTRIBUIDAS

Las aplicaciones distribuidas no son más que procesos que ejecutan tareas en diferentes unidades de procesamiento. Esto es lo que se denomina CPU. Esta CPU puede estar en la misma máquina o en máquinas separadas entre sí. Lo que permite el desarrollo de aplicaciones distribuidas es que estas tareas se comuniquen entre sí.

En este capítulo vamos a ver las diferentes capas que forman una aplicación distribuida así que como la forma de implementarlas usando Java RMI (Java Remote Method Invocation). Esta tecnología se usa para invocar un método de forma remota. Es importante destacar que RMI se usa única y exclusivamente para comunicar aplicaciones implementadas en el lenguaje de programación Java.

En caso de que se quiera comunicar otro tipo de tecnologías, entonces tendríamos que usar CORBA (Common Object Request Broker Architectire) o SOAP (Simple Object Access Protocol). Estas arquitecturas permiten a módulos escritos en diferentes lenguajes comunicarse e interaccionar entre ellos.

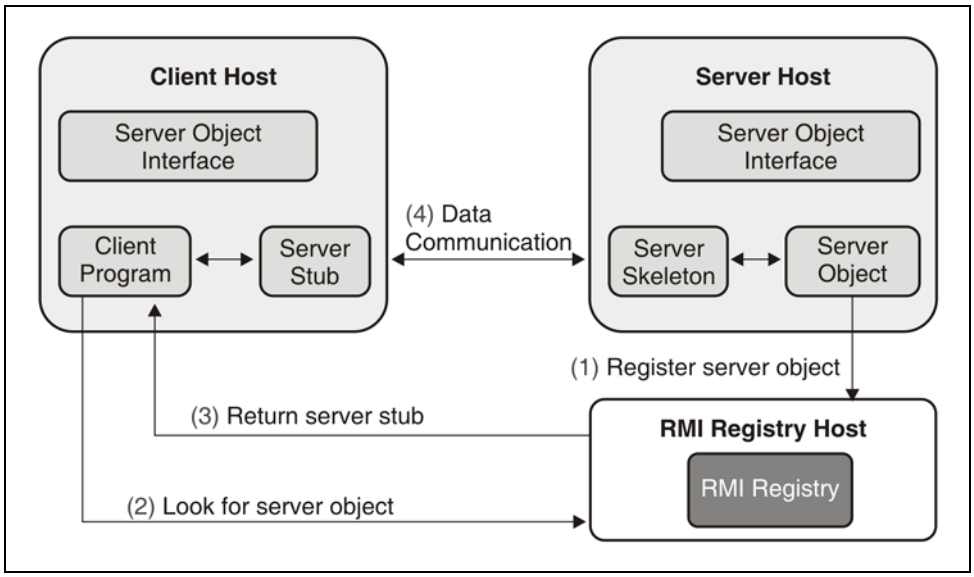
Pasos básicos de la arquitectura RMI

1. El servidor se registra con un nombre en el registro. Esto es el binding.
2. El cliente localiza los servicios en el registro por su nombre.
3. El cliente crea un stub que hace referencia al skeleton
4. El cliente invoca al stub local.
5. El stub manda un mensaje al skeleton remoto.

6. El skeleton remoto invoca al método local del servidor.
7. El skeleton tranfiere al stub los resultados de la llamada.
8. El stub finaliza la llamada.

Diferentes componentes de aplicaciones distribuidas RMI.

- **Cliente:** Se encargan de identifcar y llamar a los métodos de los objetos de servidor.
- **Servidor:** Son objetos que exponen interfaces remotas públicas que son llamadas por los clientes.
- **Registro:** Este es un servicio dónde se registran todos los objetos remotos y los clientes pueden localizarlos para llamarlos.



Ahora vamos a ver un ejemplo práctico “Hello world”.

Los pasos sería los siguientes:

1. Lo primero de todo es la definición de la interfaz que derive de la interfaz Remote y que defina los métodos necesarios. En este caso podemos crear un método print().

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface IServiceHelloWorld extends Remote{
```

```
        String print() throws RemoteException;

}
```

2. El siguiente paso es la implementación del servicio y sus métodos que implementan la interfaz creada anteriormente.

```
import java.rmi.RemoteException;

public class ServiceHelloWorld implements IServiceHelloWorld{

        @Override
        public String print() throws RemoteException {
                // TODO Auto-generated method stub
                return "Hello World";
        }


}
```

3. Ahora vamos a crear la clase que inicialice el servidor. Este servidor va a inicializar el servicio remoto y lo va a registrar en el registry para que sea visible por los clientes. Una vez ejecutada esta clase va a registrar el servicio RMI y va a estar escuchando a posibles clientes que ejecuten los métodos remotos.

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;

public class ServerHelloWorld {

        public static void main(String[] args) {
                if (args.length!=1) {
                        System.err.println("Uso: ServidorEco numPuertoRegis-
tro");
                        return;
                }
                if (System.getSecurityManager() == null) {
                    System.setSecurityManager(new RMISecurityManager());
                }
```

```
        try {
            ServiceHelloWorld srv = new ServiceHelloWorld();
            Naming.rebind("rmi://localhost:" + args[0] + "/HelloWorld",
srv);
        }
        catch (RemoteException e) {
            System.err.println("Connection error: " + e.toString());
            System.exit(1);
        }
        catch (Exception e) {
            System.err.println("Server excepcion:");
            e.printStackTrace();
            System.exit(1);
        }

}
```

4. Ahora se desarrolla la parte del cliente.

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class ClientHelloWorld {
      public static void main(String[] args) {
            if (args.length<2) {
           System.err.println("ClientHelloWorld parameters hostregistro numPu-
ertoRegistro ...");
           return;
        }

        if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());

        try {

            ServiceHelloWorld srv = (ServiceHelloWorld) Naming.lookup("//" +
args[0] + ":" + args[1] + "/HelloWorld");

            for (int i=2; i<args.length; i++)
                System.out.println(srv.print());
        }
```

```
        catch (RemoteException e) {
            System.err.println("Communication error: " + e.toString());
        }
        catch (Exception e) {
            System.err.println("HelloWorld Exception");
            e.printStackTrace();
        }

}
```

Capítulo 8

MAP REDUCE

8.1. HISTORIA

Map-Reduce es una nueva forma de programación creada por Google para dar solución al procesamiento de grandes colecciones de datos en lo que se denomina commodity hardware. El nombre de este framework está basado en las dos operaciones principales que son el Map y el Reduce que veremos más adelante.

Commodity hardware

Una vez google liberó los papeles donde se explicaba este modelo, Yahoo empezó a desarrollar un proyecto de software libre denominado Hadoop. Más adelante se empezó a distribuir bajo licencia Apache.

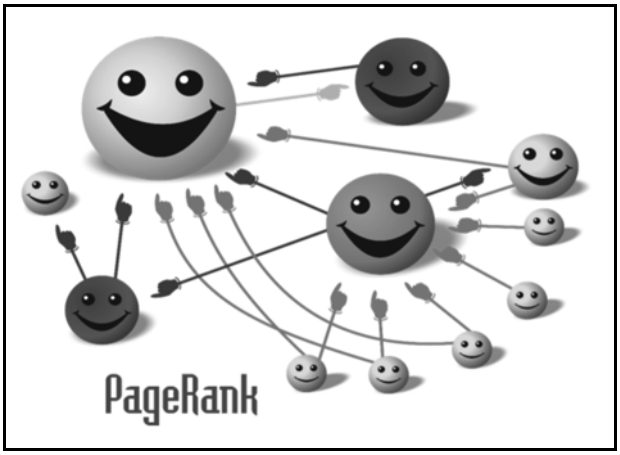
La idea principal es poder ejecutar de forma paralela tantos procesos como queramos para poder disminuir el tiempo de ejecución sobre la colección completa de datos. De todos modos este tipo de procesamiento no es aplicable a todo tipo de problemas, más bien lo contrario. Sólo es aplicable a aquellos problemas que por su naturaleza son imposibles de solventar con el procesamiento tradicional.

Hay que cambiar el concepto que se tiene de la programación clásica donde lo que hacemos es ejecutar un proceso en una máquina. Los datos se recuperan de dónde sea necesario, un fichero, base de datos etc. Los datos por lo tanto van al proceso y se ejecuta el mismo. En Map-Reduce es al contrario, los datos residen en un sistema de ficheros llamado HDFS y los procesos se mueven a esas máquinas que contienen los datos.

El principal caso de uso por el que Google desarrolló la tecnología Map-Reduce es por la necesidad de multiplicar matrices para calcular el PageRank. Google se dio cuenta que era imposible procesar tales cantidades de datos sin dividir el problema y atacarlo de una forma diferente.

Wikipedia

> *PageRank es una marca registrada y patentada1 por Google el 9 de enero de 1999 que ampara una familia de algoritmos utilizados para asignar de forma numérica la relevancia de los documentos (o páginas web) indexados por un motor de búsqueda. Sus propiedades son muy discutidas por los expertos en optimización de motores de búsqueda. El sistema PageRank es utilizado por el popular motor de búsqueda Google para ayudarle a determinar la importancia o relevancia de una página. Fue desarrollado por los fundadores de Google, Larry Page (apellido, del cual, recibe el nombre este algoritmo) y Sergey Brin, en la Universidad de Stanfordmientras estudiaban el posgrado en ciencias de la computación.*



8.2. MAP AND REDUCE

El paradigma principal en el que se basa Map-Reduce es el envío de los programas a los datos que residen en una computadora de potencia intermedia, denominado commodity hardware.

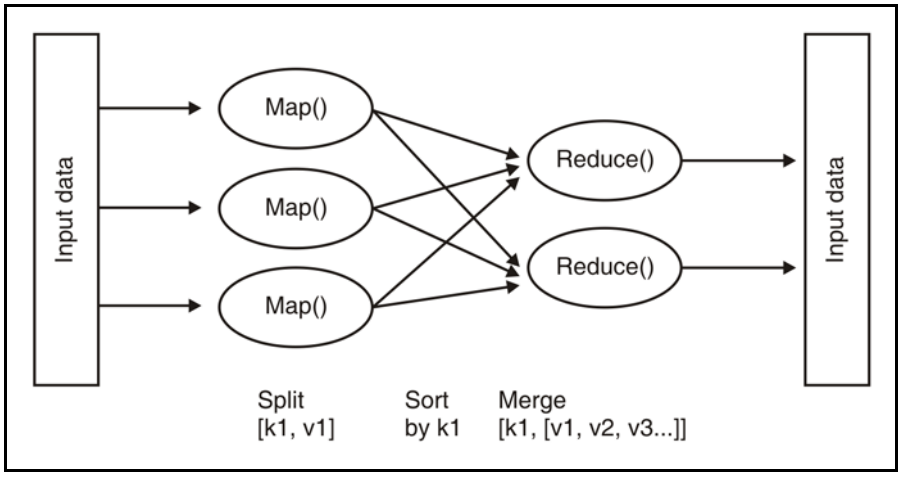
Las fases en las que se divide son la fase de mapeo, shuffle y reducción. Vamos a ver con más detalle cada una de estas etapas:

Map:

En la fase de mapeo el objetivo principal es procesar los datos de entrada. Estos datos se encuentran almacenados normalmente en disco. Existe una diferente capa de almacenamiento que se basa en el sistema de archivos HDFS que veremos más adelante. Los datos por lo tanto son procesados por la fase de mapeo y generan una salida.

Reduce:

La fase Reduce combina el shuffiling y el reduce. Se van a procesar los datos que llegan desde la función Map, estos se agrupan y se vuelven a procesar para generar otro resultado.



Vamos a ver más en detalle las operaciones Map y Reduce:

Map(k1,v1) -> list(k2,v2)

Esta función se encarga del Mapeo. Se ejecuta en paralelo y tomo como entrada una pareja e Key,value pairs. Se va a procesar uno a uno de la entrada este conjunto de k1,v1. Después el framework va a juntar todos los datos con la misma clave(key) y los va a agrupar. Creando por lo tanto una lista por cada clave.

Reduce(k2, list(v2)) -> list(v3)

Cada llamada a este método genera una lista de valores list(v3).

Inicialmente entender el framework Map-Reduce es un poco confuso ya que tenemos que cambiar el esquema mental en el que trabajamos como programadores en tareas cotidianas.

Vamos a ver un ejemplo que se basa en contar el número de veces que aparece cada palabra en un texto o conjunto de documentos.

MAP

```
map(String name, String document):
  // clave: nombre del documento
  // valor: contenido del documento
  for each word w in document:

    EmitIntermediate(w, 1);
```

REDUCE

```
reduce(String word, Iterator partialCounts):
  // word: una palabra
  // partialCounts: una [[Iterador (patrón de diseño)|lista parcial]]
para realizar cuentas agregadas
  int result = 0;
  for each v in partialCounts:
    result += ParseInt(v);

  Emit(result) ;
```

Ejemplo real en java

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;


public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
```

```
                        ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

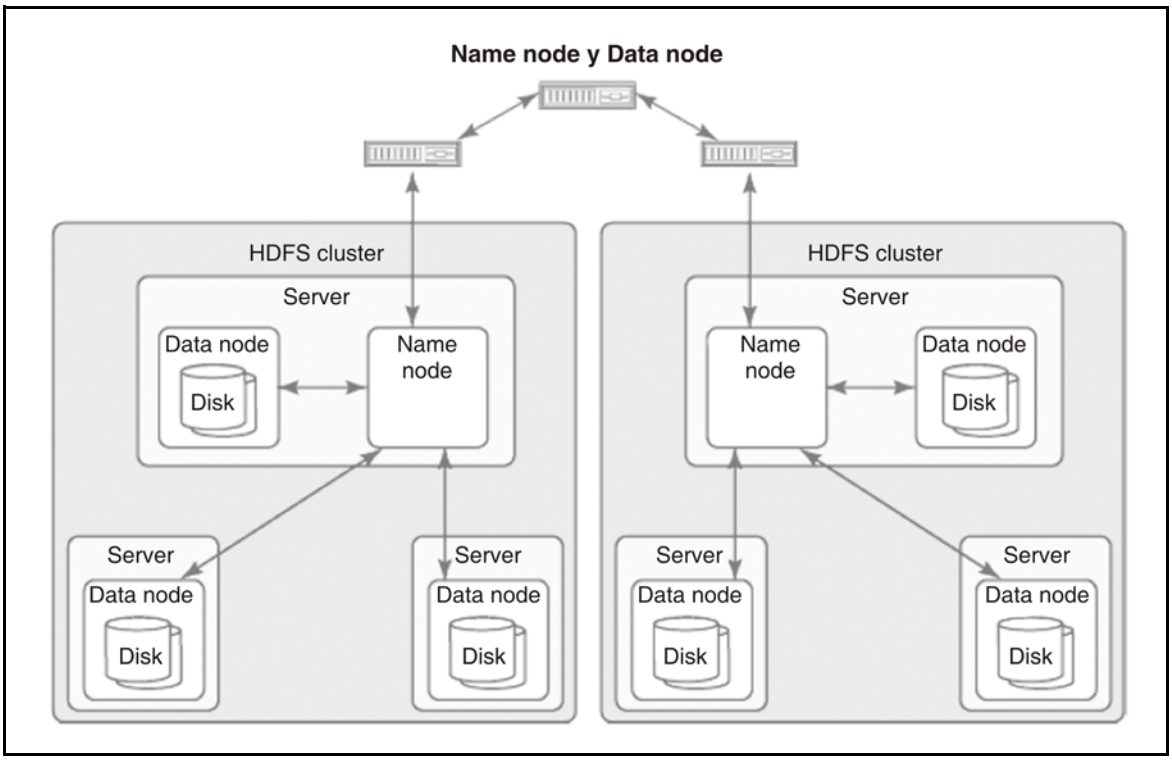
8.3. ARQUITECTURA

Vamos a ver a continuación cual es la arquitectura que permite ejecutar el framework de procesamiento distribuido Hadoop.

Vamos dividir inicialmente el problema en dos partes, los datos y el procesamiento. Para poder procesar los datos, estos deben de encontrarse inicialmente almacenados en algún lugar. Este lugar por regla general es **HDFS** (Haddop Distributed File System).

- **HDFS** (Haddop Distributed File System): Es un Sistema de almacenamiento que permite almacenar los datos teniendo en cuenta que se van a procesar de forma

distribuida. Por lo tanto es capaz de almacenarlos proporcionando tolerancia a fallos, alta disponibilidad etc. Para ello simplemente usa un sistema de replicación de los datos en diferentes máquinas. Los archivos suelen ser procesados por HDFS de manera que si son muy grandes se dividen en bloques y se distribuyen en diferentes nodos del cluster. El nombre que recibe cada una de las máquinas que contienen datos es Data Node.



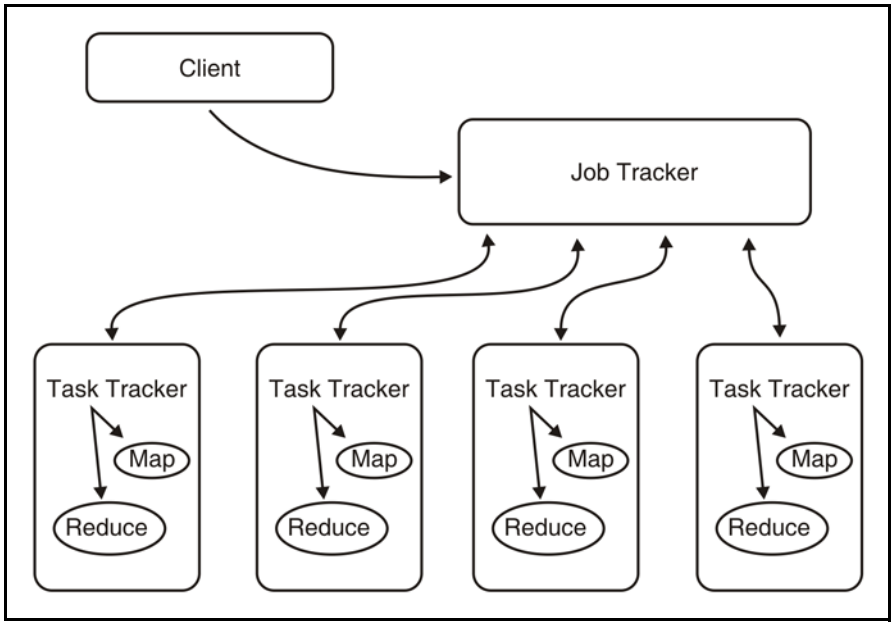
- **Namenode:** El NameNode es la parte referente al almacenamiento de los datos que gestiona todo lo relacionado con el almacenamiento distribuido. Decide cómo se van a almacenar los datos en los DataNode, cuantos bloques se van a crear y cómo y dónde se van a replicar. Si comparamos esta arquitectura de una forma tradicional veríamos esta parte cómo el master que gestiona todo. Esta máquina guarda un registro de dónde reside cada bloque de datos.
- **DataNode:** Por otro lado el DataNode sería la máquina que almacena los bloques, es decir los datos y sobre la que se realizan las operacions de I/O.

Ahora vamos a ver la parte referente al procesamiento.

- **JobTracker:** Esta es la parte que gestiona al procesamiento de los datos. Es la parte encargada de la gestión y planificación de la ejecución de las tareas. Una vez que se envía la ejecución de una tarea, el JobTracker decide el plan de

ejecución sobre que datos y en que máquinas. También se encargará de monitorizar y gestionar los recursos hasta la finalización de la ejecución.

- **TaskTracker:** Esta máquina se encarga de la ejecución de las tareas en sí. No gestiona nada simplemente usa la CPU para la ejecución.



8.4. EJEMPLO PRÁCTICO

Realizar una multiplicación de matrices usando el paradigma Map-reduce. Para que esto llegue a ser eficiente deberíamos de ejecutar una gran cantidad de operaciones en paralelo. El objetivo ahora es entender cómo funciona.

http://hadoopgeek.com/mapreduce-matrix-multiplication/

```
public class MatrixMapper extends Mapper<LongWritable, Text, Text, Text>
{
	@Override
	protected void map(LongWritable key, Text value, Context context)
		throws IOException, InterruptedException
	{
		// input format is ["a", 0, 0, 63]
		String[] csv = value.toString().split(",");
		String matrix = csv[0].trim();
		int row = Integer.parseInt(csv[1].trim());
```

```
                int col = Integer.parseInt(csv[2].trim());
                if(matrix.contains("a")){
                      for (int i=0; i < lMax; i++){
                            String akey = Integer.toString(row) + "," +
                            Integer.toString(i);
                            context.write(new Text(akey), value);
                      }
                }
                if(matrix.contains("b")){
                      for (int i=0; i < iMax; i++){
                            String akey = Integer.toString(i) + "," +
                            Integer.toString(col);
                            context.write(new Text(akey), value);
                      }

                                                  }

      }

}

public class MatrixReducer extends Reducer<Text, Text, Text, IntWritable>
{

@Override
protected void reduce(Text key, Iterable<Text> values, Context context)
                throws IOException, InterruptedException {

      int[] a = new int[5];
      int[] b = new int[5];
      // b, 2, 0, 30
      for (Text value : values) {
            System.out.println(value);
            String cell[] = value.toString().split(",");
            if (cell[0].contains("a")) // take rows here{
                  int col = Integer.parseInt(cell[2].trim());
                  a[col] = Integer.parseInt(cell[3].trim());
            }
```

```
            else if (cell[0].contains("b")) // take col here{
                  int row = Integer.parseInt(cell[1].trim());
                  b[row] = Integer.parseInt(cell[3].trim());
            }
      }
      int total = 0;
      for (int i = 0; i < 5; i++) {
            int val = a[i] * b[i];
            total += val;

}

context.write(key, new IntWritable(total));
      }

}
```

Capítulo 9

PROYECTO FINAL

El principal objetivo que persigue el proyecto final es la puesta en práctica de la mayor parte de los conocimientos adquiridos durante los capítulos anteriores.

A continuación se definen los requisitos y casos de uso.

- Desarrollar un motor de búsqueda que contenga dos módulos principales.
- Modulo Indexación: La indexación de un conjunto de documentos de texto en formato txt. Estos documentos se almacenarán en una estructura de datos en memoria en forma de índice invertido.
- Módulo de búsqueda: Este módulo desarrollará la funcionalidad referente a la búsqueda de resultados en los documentos indexados.
- Para ello se va a utilizar una arquitectura distribuida basada en java RMI.
- Crear dos servicios Remotos uno para la indexación y otro para la búsqueda.
- El servicio remoto de indexación. Tendrá un método que recibirá la ubicación de los ficheros a indexar. Ejecutará la indexación y creará el índice en memoria.
- El servicio remoto de búsqueda. Tendrá un método que recibirá el string a buscar y devolverá los ficheros que contengan ese string buscado.
- Implementar el cliente que sea capaz de registrar y llamar a los servicios remotos.

Bibliografía

[1] BERRY, M.W.; BROWNE, M. 2005. Understanding Search Engines: Mathematical modeling and text retrieval. Siam. 34-41pp.

[2] ROBERTSON, S. 2004. Understanding Inverse Document Frequency: On theoretical arguments for IDF. Journal of Documentation. Vol.60: (5), 503-520 pp.

[3] Chuck Lam, 2011. Hadoop in action, Manning publications Relevant Search, 2015, Manning.

Enlaces web:

[1] Enlace web: http://lycog.com/distributed-systems/java-rmi-overview/

[2] Enlace web: http://www.lnds.net/blog/2013/11/la-notacion-big-o.html