Algoritmos y Estructuras de Datos:

Motivación

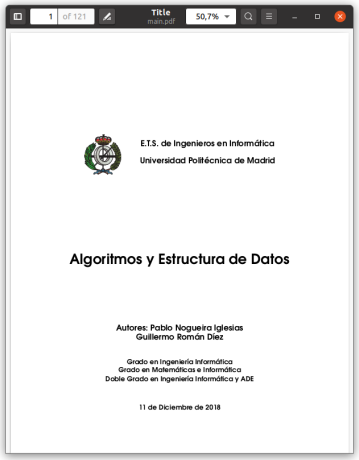
Lars-Åke Fredlund

Universidad Politécnica de Madrid

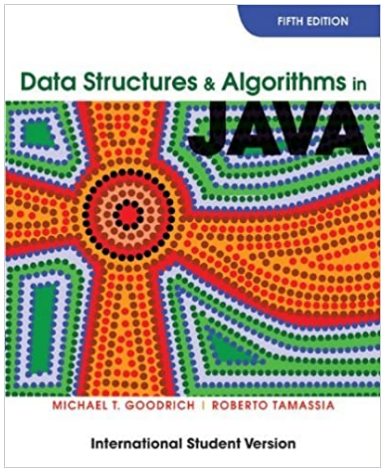
Curso 2021-2022

Material

- Transparencias, google, . . .
- Guion:



- Opcional: "Data Structures and Algorithms in Java" de Michael T. Goodrich, Roberto Tamassi:



Contenidos

- Énfasis en **estructuras de datos**, y algo sobre **algoritmos**
- Para los que tienen mucho interés en algoritmos – "The Art of Programming" (asignatura opcional)

Ejemplo Motivador: registro de coches

- Queremos implementar un registro de coches, donde se guarda información sobre los coches registrado en España.
- Primero hay que decidir que información es necesario guardar para cada coche:

Ejemplo Motivador: registro de coches

- Queremos implementar un registro de coches, donde se guarda información sobre los coches registrado en España.
- Primero hay que decidir que información es necesario guardar para cada coche:
  - matricula
  - tipo de coche (marca, numero bastidor, color, ...)
  - dueño?
  - resultado itv
  - multas asociadas

Representación en Java

Podemos crear una clase `Coche` que guarda estos datos, y para cada coche, crear un objeto de dicha clase:

```
class Coche {
  String matricula;
  String numeroBastidor;
  String color;
  String marca;
  String modelo;
  String fechaUltimoITV;
  int kmUltimoITV;
  ...
}
```

¿Que funcionalidad?

Que funcionalidad debería tener nuestro registro de coches?

¿Que funcionalidad?

Que funcionalidad debería tener nuestro registro de coches?

- Añadir un coche – `add(Coche)`
- Borrar un coche dado una matricula – `remove(Matricula)`
- Buscar un coche dado su matricula – `find(Matricula)`
- Modificar los datos de un coche registrado – `update(Coche)`

Un conjunto de operaciones/métodos que forman un API (application interface) para acceder al registro de coches.

API registro de coche

```
class Registro {
  public void add(Coche coche) { ... }
  public void remove(String matricula) { ... }
  public Coche find(String matricula) { ... }
  public void update(Coche coche) { ... }
}
```

Primer paso

Antes de empezar a programar el registro deberíamos hacernos una pregunta que siempre es muy relevante:

Primer paso

Antes de empezar a programar el registro deberíamos hacernos una pregunta que siempre es muy relevante:

- ¿Podemos reusar una librería/programa ya existente en vez de programar todo desde cero?

Primer paso

Antes de empezar a programar el registro deberíamos hacernos una pregunta que siempre es muy relevante:

- ¿Podemos reusar una librería/programa ya existente en vez de programar todo desde cero?
- En Java existe la librería estándar `java.util` (`https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html`) que contiene muchos estructuras de datos ya programadas

Primer paso

Antes de empezar a programar el registro deberíamos hacernos una pregunta que siempre es muy relevante:

- ¿Podemos reusar una librería/programa ya existente en vez de programar todo desde cero?
- En Java existe la librería estándar `java.util` (`https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html`) que contiene muchos estructuras de datos ya programadas
- En esta asignatura usaremos principalmente **aedlib** (`http://costa.ls.fi.upm.es/entrega/aed/docs/aedlib/`) – una librería parecido a `java.util` pero mas "pequeño"

AEDLIB – elegir estructura de dato

¿Que estructuras de datos contiene aedlib que son útiles para implementar el API (los métodos add, find, . . .) del registro?

- las listas indexada de Programación II (IndexedList)
- mapas (mejor, pero los introducimos mas tarde en la asignatura)

Decidimos usar IndexedLists.

Implementación de add y find usando indexedlists

Implementación de add y find usando indexedlists

- Asumimos que hay existe un atributo `ilist` de tipo indexedlist que contiene los objetos de tipo coche guardado en el registro:

```
IndexedList<Coche> ilist = new ArrayIndexedList<>();
```

Implementación de add y find usando indexedlists

- Asumimos que hay existe un atributo `ilist` de tipo indexedlist que contiene los objetos de tipo coche guardado en el registro:

```
IndexedList<Coche> ilist = new ArrayIndexedList<>();
```

- Implementación de add:

```
// Anade un coche nuevo
void add(Coche coche) {
  ilist.add(ilist.size(), coche);
}
```

Implementación de add y find usando indexedlists

- Implementación de `find`:

```
// Busca un objeto coche dado su matricula
Coche find(String matricula) {
  boolean encontrado = false;
  int i = 0;
  Coche coche = null;

  while (i < ilist.size() && !encontrado) {
    coche = ilist.get(i);
    if (coche.matricula.equals(matricula))
      encontrado = true;
    i++;
  }

  if (encontrado) return coche;
  else return null;
}
```

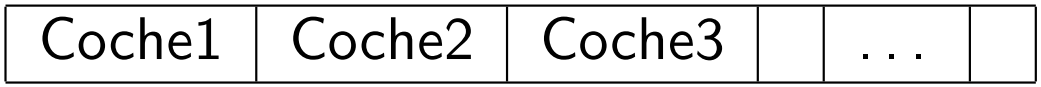
Algoritmos

- Bien – esperamos que el codigo sea correcto – ¿que hace?
- Pero no sabemos si el codigo es suficiente **eficiente**.
- La eficiencia es importante en la informática - si guardamos un millón de coches, y si muchas personas buscan coches a la vez, la operación de buscar (`find`) no debería ser demasiado ineficiente.

Eficiencia de `add(Coche)`

Como esta implementado `add` para `IndexedList`?

- Internamente una lista indexada usa un array donde se guarda los coches:

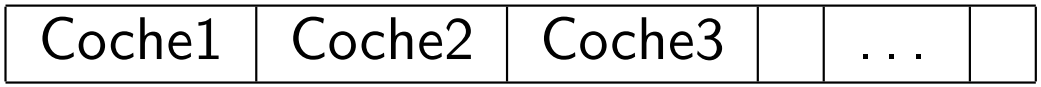


- El array no esta ordenado
- Entonces como se añade un coche nuevo? – `add(Coche4)`

Eficiencia de `add(Coche)`

Como esta implementado `add` para `IndexedList`?

- Internamente una lista indexada usa un array donde se guarda los coches:

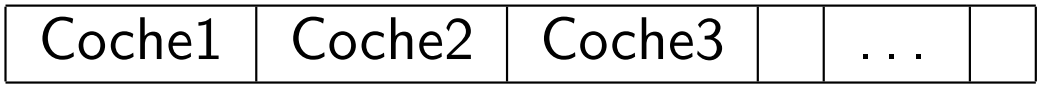


- El array no esta ordenado
- Entonces como se añade un coche nuevo? – `add(Coche4)`
- Simplemente se añade el coche nuevo al final del array:

Eficiencia de `add(Coche)`

Como esta implementado `add` para `IndexedList`?

- Internamente una lista indexada usa un array donde se guarda los coches:



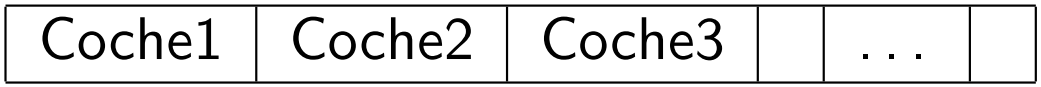
- El array no esta ordenado
- Entonces como se añade un coche nuevo? – `add(Coche4)`
- Simplemente se añade el coche nuevo al final del array:

- ¿Que eficiente es eso? No depende del numero de coches ya guardadas en el array – el método ejecute **muy rápido** (complejidad constante - $O(1)$).

Eficiencia de `add(Coche)`

Como esta implementado `add` para `IndexedList`?

- Internamente una lista indexada usa un array donde se guarda los coches:



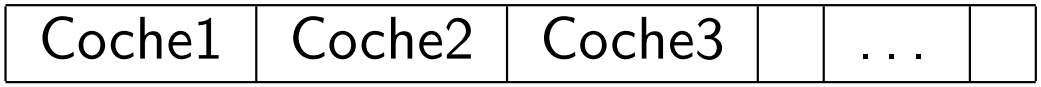
- El array no esta ordenado
- Entonces como se añade un coche nuevo? – `add(Coche4)`
- Simplemente se añade el coche nuevo al final del array:

- ¿Que eficiente es eso? No depende del numero de coches ya guardadas en el array – el método ejecute **muy rápido** (complejidad constante - $O(1)$).
- Excepto en el caso limite cuando no hay mas espacio en el array. . .

Eficiencia de `find(Coche)`

El tiempo de ejecutar una llamada a `find` es dominado por la busqueda de un coche dado su matricula en una lista indexada. ¿Que eficiente es?

- Una lista indexada usa un array donde se guarda los coches:

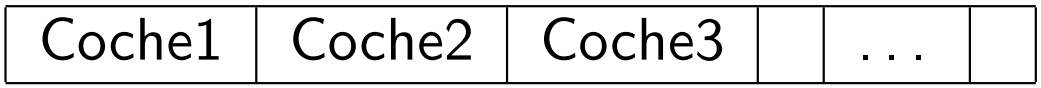


- **El array no es ordenado**
- Para encontrar un coche hay que acceder, en el peor caso, a **todos** los coches dentro el array (si el coche buscado esta en el ultimo lugar del array)
- Si hay un millón de coches, en el peor caso se accede a un millón coches.
- En el caso mediano, se accede a 1000000/2 coches.
- **Es lento**. En el peor caso, el numero de comprobaciones de matriculas necesarios para una búsqueda se puede caracterizar como una función lineal del numero de coches guardados en el array (complejidad lineal – $O(n)$).

¿Como se puede mejorar la eficiencia del registro de coches?

¿Como se puede mejorar la eficiencia del registro de coches?

- Idea: ordenar los coches dentro el indexedlist según su matricula:



donde la matricula de $Coche_i <$ matricula de $Coche_{i+1}$, etc.

¿Como se puede mejorar la eficiencia del registro de coches?

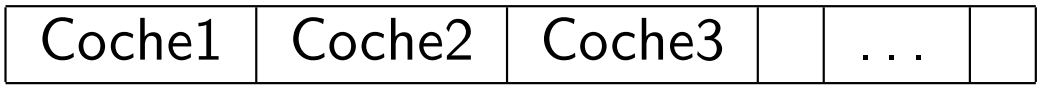
- Idea: ordenar los coches dentro el indexedlist según su matricula:

Coche1	Coche2	Coche3		...	

  donde la matricula de $Coche_i <$ matricula de $Coche_{i+1}$, etc.
- ¿Como se cambiara la implementación de add?
  - En vez de añadir un coche nuevo al final del array tenemos que *buscar* el lugar correcto (según el orden) para *insertarlo*.
  - ¿Que algoritmo podemos usar para buscar un elemento en una array ordenada? (¿y como eficiente es?)

¿Como se puede mejorar la eficiencia del registro de coches?

- Idea: ordenar los coches dentro el indexedlist según su matricula:



  donde la matricula de $Coche_i <$ matricula de $Coche_{i+1}$, etc.
- ¿Como se cambiara la implementación de add?
  - En vez de añadir un coche nuevo al final del array tenemos que *buscar* el lugar correcto (según el orden) para *insertarlo*.
  - ¿Que algoritmo podemos usar para buscar un elemento en una array ordenada? (¿y como eficiente es?)
  - ¡Búsqueda binaria! El coste de buscar un coche es *logarítmico* en vez de lineal – $O(logn)$.

¿Como se puede mejorar la eficiencia del registro de coches?

- Idea: ordenar los coches dentro el indexedlist según su matricula:

Coche1	Coche2	Coche3		...	

  donde la matricula de $Coche_i$ < matricula de $Coche_{i+1}$, etc.
- ¿Como se cambiara la implementación de add?
  - En vez de añadir un coche nuevo al final del array tenemos que *buscar* el lugar correcto (según el orden) para *insertarlo*.
  - ¿Que algoritmo podemos usar para buscar un elemento en una array ordenada? (¿y como eficiente es?)
  - ¡Búsqueda binaria! El coste de buscar un coche es *logarítmico* en vez de lineal – $O(logn)$.
  - ¿Que costoso es insertar un coche nuevo? En el peor caso tenemos que insertar el coche en la primera posición en el array, y mover los demás coches "a la derecha" (un coste lineal en el numero de los coches).

Coche0	Coche1	Coche2	Coche3		...

  - ¿Cual es la complejidad final para implementar add(Coche)? El coste de mover los elementos domina el coste de add: coste lineal – $O(n)$.

¿Como se puede mejorar la eficiencia del registro de coches?

- ¿Como se cambiara la implementación de `find`?
- Solo hace falta una búsqueda binaria. Coste logarítmico (en el numero de coches en el registro) – $O(logn)$.

Análisis: ¿que implementación deberíamos usar?

Comparamos los costes de tener listas no ordenadas y listas ordenadas:

	Lista no ordenada	Lista ordenada
Complejidad `add(Coche)`	$O(1)$	$O(n)$
Complejidad `find(Matricula)`	$O(n)$	$O(logn)$

- Cuanto es log2(1 000 000)? Es decir ¿cuantos comparaciones de matriculas tenemos que hacer en el caso de una búsqueda binaria?

Análisis: ¿que implementación deberíamos usar?

Comparamos los costes de tener listas no ordenadas y listas ordenadas:

	Lista no ordenada	Lista ordenada
Complejidad `add(Coche)`	$O(1)$	$O(n)$
Complejidad `find(Matricula)`	$O(n)$	$O(logn)$

- Cuanto es log2(1 000 000)? Es decir ¿cuantos comparaciones de matriculas tenemos que hacer en el caso de una búsqueda binaria?
- ¡20! (bastante mejor que complejidad lineal)

Análisis: ¿que implementación deberíamos usar?

Comparamos los costes de tener listas no ordenadas y listas ordenadas:

	Lista no ordenada	Lista ordenada
Complejidad `add(Coche)`	$O(1)$	$O(n)$
Complejidad `find(Matricula)`	$O(n)$	$O(logn)$

- Cuanto es log2(1 000 000)? Es decir ¿cuantos comparaciones de matriculas tenemos que hacer en el caso de una búsqueda binaria?
- ¡20! (bastante mejor que complejidad lineal)
- Análisis: `add` es mas rápido en el caso de tener listas no ordenadas, y `find` es mas rápido en el caso de tener listas ordenadas.
- Entonces, ¿deberíamos usar listas ordenadas o no?

Análisis: ¿que implementación deberíamos usar?

Comparamos los costes de tener listas no ordenadas y listas ordenadas:

	Lista no ordenada	Lista ordenada
Complejidad `add(Coche)`	$O(1)$	$O(n)$
Complejidad `find(Matricula)`	$O(n)$	$O(logn)$

- Cuanto es log2(1 000 000)? Es decir ¿cuantos comparaciones de matriculas tenemos que hacer en el caso de una búsqueda binaria?
- ¡20! (bastante mejor que complejidad lineal)
- Análisis: `add` es mas rápido en el caso de tener listas no ordenadas, y `find` es mas rápido en el caso de tener listas ordenadas.
- Entonces, ¿deberíamos usar listas ordenadas o no?
- Para resolver esta duda tenemos que saber algo sobre como el registro va a ser usado en la practica:
  - Si en la practica se hace mucho mas búsquedas de coches que operaciones para añadir coches deberíamos usar una lista ordenada.
  - Si en la practica se añade coches nuevos mucho mas frecuentemente que búsquedas deberíamos usar una lista no ordenada.

"Moral" de la historia – o AED en 4 puntos

- **Reusamos** – usamos librerias apropiadas
- **Elegimos estructuras de datos apropiados**, que nos ayuda a escribir codigo facil de entender, y compacto
- **Analisamos la eficiencia** de nuestro codigo; que complejidad tiene los metodos de la libreria que usamos
- **Eligimos una implementacion de la libreria apropiada** para nuestro caso de uso (por ejemplo, muchas busquedas, pocos cambios o pocas busquedas y pocos cambios).