Anexo

Teoremas de Coste, para métodos recursivos

n, talla del problema; T(n), coste del método recursivo

Teorema 1: $T(n) = a \cdot T(n - c) + b$, con $b \geq 1$
- Si $a = 1$, $T(n) \in \Theta(n)$
- Si $a > 1$, $T(n) \in \Theta(a^{n/c})$

Teorema 2: $T(n) = a \cdot T(n - c) + b \cdot n + d$, con b y $d \geq 1$
- Si $a = 1$, $T(n) \in \Theta(n^2)$
- Si $a > 1$, $T(n) \in \Theta(a^{n/c})$

Teorema 3: $T(n) = a \cdot T(n / c) + b$, con $b \geq 1$

cantidad de llamadas recursivas
- Si $a = 1$, $T(n) \in \Theta(\log_c n)$
- Si $a > 1$, $T(n) \in \Theta(n^{\log_c a})$

Teorema 4: $T(n) = a \cdot T(n / c) + b \cdot n + d$, con b y $d \geq 1$
- Si $a < c$, $T(n) \in \Theta(n)$
- Si $a = c$, $T(n) \in \Theta(n \cdot \log_c n)$
- Si $a > c$, $T(n) \in \Theta(n^{\log_c a})$

Entre más uniforme las particiones mejor.

En una de las partes no voy a hacer nada ya que ya se en base al límite que puedo despreciar una mitad

Ecuación de Recurrencia para el caso general de **vencer**

$$T_{vencer}(n > n_{base}) = a * T_{vencer}(n / c) + \underbrace{T_{dividir}(n) + T_{combinar}(n)}$$

El coste estará en función de: Número de llamadas recursivas (a) · Reducción de la talla (geométrica) (c) · Sobrecarga en cada llamada

public static TipoResultado vencer(TipoDatos n) {
TipoResultado resMetodo, resLlamada_1,...,resLlamada_a;
if (n == nbase) { resMetodo = solucionCasoBase(n); } else {
int c = dividir(n); resLlamada_1 = vencer(n / c);
...
resLlamada_a = vencer(n / c);
resMetodo = combinar(n, resLlamada_1, ...,resLlamada_a); }
return resMetodo; }

2. Estrategia RL

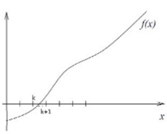
Ejercicios (I)

Ejercicio 1: sea *v* un array de *int* que se ajustan al perfil de una curva continua y monótona creciente, tal que *v[0]<0* y *v[v.length-1]>0*. Existe una única posición *k* de *v*, *0≤k<v.length-1*, tal que entre *v[k]* y *v[k+1]* la función vale 0, i.e. tal que *v[k]≤0* y *v[k+1]>0*.
Diseña el "mejor" método recursivo que calcule *k* y analiza su coste

Los siguientes son 2 ejemplos del contenido de *v* para la curva f(x) del dibujo:

0	1	2	3	4	5
-3	-2	-1	1	4	5

-3	-2	-1	0	1	4	5




```java
 */
public class EjerciciosDyV {
    public static int puntoCruce(int []v){ // es un vector
        return puntoCruce(v,0,v.length-1);
    }

    private static int puntoCruce(int []v, int ini, int fin){
        int m = (ini+fin)/2;
        if (v[m] <= 0 && v[m+1] >0)
            return m;
        // me interesa ver si me interesa la izquierda o la derecha.
        if (v[m]>0)
        return puntoCruce(v,ini,m-1); // yo ya miré la m así que le quito 1
        else
            return puntoCruce(v, m+1, fin);
    }
}
```

2. Estrategia RL

Ejercicios (II)

Ejercicio 3: Componente del array con valor igual a posición
Diseña un método recursivo que, con el menor coste posible, determine si un array `v` de tipo `int`, Ordenado Asc. y sin elementos repetidos, contiene alguna componente cuyo valor es igual a la posición que ocupa; si existe tal componente el método devuelve su posición y sino -1

```
boolean exists = False

public static int ExistsArrayWithEqualPosition (int []v){
    return ExistsArrayWithEqualPosition()
}

private static int ExistsArrayWithEqualPos
    int m = (ini + fin) / 2;
    // asumiendo que está ordenado
    if (v[m] > m

}
```

```java
public static int valYposIguales(int[] v) {
   return valYposIguales(v, 0, v.length-1);
}

public static int valYposIguales(int[] v, int i, int j) {
   if (i > j) return -1;
   int m = (i + j) / 2;
   if (v[m] == m) return m;
   if (v[m] < m)  return valYposIguales(v, m + 1, j);
   return valYposIguales(v, i, m - 1);
}
```

Ejercicio 4: Diseña un método recursivo que, con el menor coste posible, compruebe si dos *String* ***x*** e ***y*** (tal que *x* es menor estricto que *y*) ocupan posiciones consecutivas en un array de *String* ***v***, ordenado ascendentemente y sin elementos repetidos.

```java
private static boolean vecinas(String[] v, String x, String y, int i, int j){
    int m = (i+j)/2;
    if (v[m].compareTo(x) > 0){
        return vecinas(v,x,y,i,m-1);
    }
    if (v[m].compareTo(x)==0){
        return v[m+1].equals(y);
    }
    return vecinas(v, x, y, m+1, j);
}
```

quicksort:
merge

1	3	7	10

0	4	5	7

0 1

```java
public static <E extends Comparable<E>> E[] fusion(E[] a, E[] b) {
   E[] res = (E[]) new Comparable[a.length + b.length];
   int i = 0, j = 0, k = 0;
   while (i < a.length && j < b.length) {
      if (a[i].compareTo(b[j]) < 0) { res[k++] = a[i++]; }
      else { res[k++] = b[j++]; }
   }
   for (int r =  i; r < a.length; r++ ) { res[k++] = a[r]; }
   for (int r =  j; r < b.length; r++ ) { res[k++] = b[r]; }
   return res;
}
```

```java
private static <E extends Comparable <E>> void mergeSort(E[] v, int i, int j)
{
   if (i < j) {    mientras sea posible dividir
      int m = (i + j) / 2;          // DIVIDIR
      mergeSort(v, i, m);           // VENCER
      mergeSort(v, m + 1, j);       // VENCER
      fusionDyV(v, i, m + 1, j);  // COMBINAR
   }
}
public static <E extends Comparable<E>> void mergeSort(E[] v) {
    mergeSort(v, 0, v.length - 1);
}
```

Modificación de ***fusion*** para que, en lugar de dos arrays, reciba un único array como parámetro

Tal como se había previsto, $T_{mergeSort}(n) \in \Theta(n \cdot \log n)$ por **T4**:

$$T_{vencer}(n > n_{base}) = \underset{a = 2}{a} * T_{vencer}(n/\underset{c = 2}{c}) + \underbrace{T_{dividir}(n)}_{\text{constante}} + \underbrace{T_{combinar}(n)}_{\text{lineal}}$$

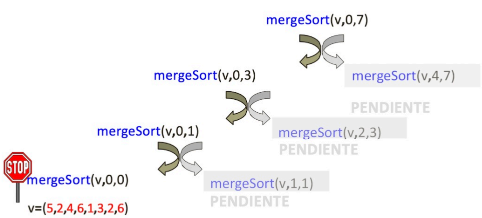
constante. lineal

$\Theta(n)$

¿por qué dividir es constante?

3.1. Merge Sort: ¿cómo ordena realmente?

- **Traza:** generación del Árbol de Llamadas para v = {5, 2, 4, 6, 1, 3, 2, 6}



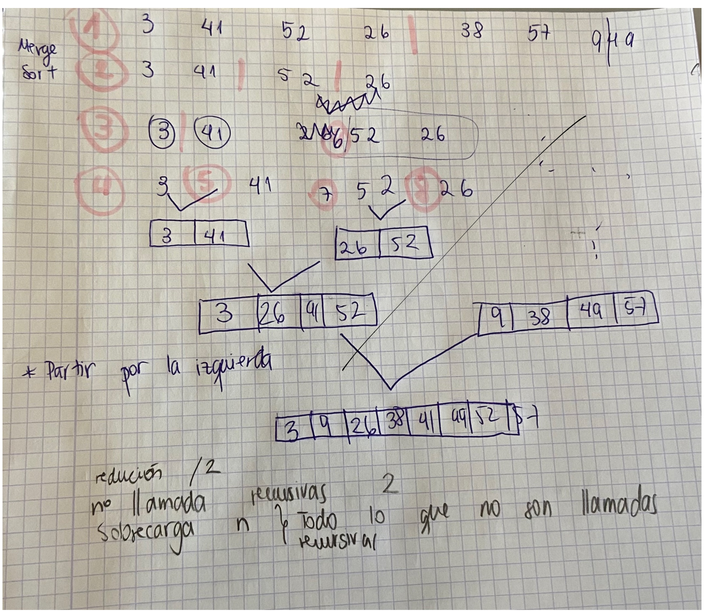
Merge-sort with Transylvanian-saxon (German) folk dance
https://www.youtube.com/watch?v=XaqR3G_NVoo

Ejercicio 5: Diseña **fusionDyV**, la modificación del método **fusion** que requiere la implementación en Java de la estrategia *Merge Sort*.

Para ello tener en cuenta que **fusionDyV**:

- En lugar de dos arrays **a** y **b**, recibe un único array **v** como parámetro; los restantes parámetros de su cabecera (**i**, **m** y **j**) permiten indicar el principio y final de los (sub)arrays de **v** a fusionar, ya ordenados en las llamadas a **mergeSort** (**v[i, m]** y **v[m + 1, j]**).
- En lugar de la suma de las longitudes de **a** y **b**, la talla del array **res** es **j – i + 1.**
- En lugar de **res**, el (sub)array resultado de la fusión es **v[i, j]**.
- El método devuelve **void** en vez de un array.
- Primero se realiza la fusión sobre **res** y luego se copian sus componentes en **v[i, j]**.

```java
public static <E extends Comparable<E>> E[] fusion(E[] a, E[] b) {
   E[] res = (E[]) new Comparable[a.length + b.length];
   int i = 0, j = 0, k = 0;
   while (i < a.length && j < b.length) {
      if (a[i].compareTo(b[j]) < 0) { res[k++] = a[i++]; }
      else { res[k++] = b[j++]; }
   }
   for (int r =  i; r < a.length; r++ ) { res[k++] = a[r]; }
   for (int r =  j; r < b.length; r++ ) { res[k++] = b[r]; }
   return res;
}
```



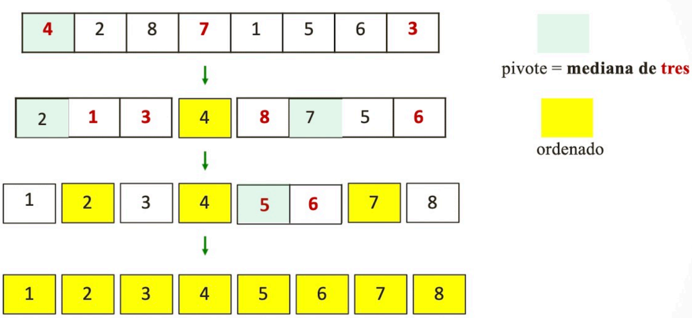
```java
// fusionDyV modifica el array v para que v[i,j] quede ordenadoAsc
private static <T extends Comparable<T>> void fusionDyV(
    T[] v, int i, int m, int j)
{
    T[] res = (T[]) new Comparable[j - i + 1];
    int i1 = i;
    int i2 = m;
    int k = 0;
    while (i1 < m && i2 <= j) {
        if (v[i1].compareTo(v[i2]) < 0 ) res[k++] = v[i1++];
        else res[k++] = v[i2++];
    }
    for (int r = i1; r < m; r++) res[k++] = v[r];
    for (int r = i2; r <= j; r++) res[k++] = v[r];
    for (int r = 0; r < res.length; r++) v[r + i] = res[r];
}

public static <T extends Comparable<T>> T[] fusion(T[] a, T[] b) {
    T[] res = (T[]) new Comparable[a.length + b.length];
    int i = 0;
    int j = 0;
```
Clase compilada - no hay errores de sintaxis

Quicksort -> Reducir problemas ya que cada vez ordeno todo a la izquierda del problema (o derecha)

El pivote al medio es lo que mejor me viene, la homogeniedad es lo mejor.
Si elijo el mínimo o máximo entonces es -1 y tiene coste n^2.
Sacamos la mediana de tres elementos para intentar reducir la probabilidad de un mal pivote

3. Aplicación de DyV ...

3.2. Quick Sort: la "vital" elección del pivote



¡¡ Cuesta menos ordenar ahora que cuando se elegía como pivote el primer elemento de cada (sub)array!!