Estudio de complejidad

Los cálculos se harán bajo la suposición de que todas las operaciones, excepto las llamadas a otras funciones estudiadas tienen coste 1.
La complejidad es O(n), siendo n el número de líneas que hay en el fichero.

```java
public static void leerCoches(final String pahtname, final List <Coche> coches) throws FileNotFoundException {
    /* Declaración del scanner a usar para leer las líneas */
    final Scanner input = new Scanner(new File(pahtname));

    StringTokenizer st;

    while (input.hasNextLine()) {
        try {
            /* Vamos obteniendo los tokens los cuales están separados por "," */
            st = new StringTokenizer(input.nextLine(), ",");;

            /* Token modelo */
            final String modelo = st.nextToken();

            /* Intentamos obtener el tipo de combustible a partir del token */
            final TipoCombustible comb = TipoCombustible.parseTipoCombustible(st.nextToken());

            /* Pasamos a entero porque en el fichero se encuentra como doble, pero no tiene sentido que este no sea entero */
            final int asientos = (int) Double.parseDouble(st.nextToken());

            final TipoTansmisión trans = TipoTansmisión.parseTipoTransmisión(st.nextToken());

            final double capacidad = Double.parseDouble(st.nextToken());

            final double consumoMedio = Double.parseDouble(st.nextToken());

            /* Creamos y añadimos el coche a la lista */
            coches.add(new Coche(modelo, comb, asientos, trans, capacidad, consumoMedio));

        } catch (NumberFormatException e) {
            System.err.println("Error trying to read number: " + e.getMessage());
        } catch (IllegalArgumentException e) {
            System.err.println("Error trying to read enum: " + e.getMessage());
        } catch (NoSuchElementException e) {
            System.err.println("Error trying to read next token.");
        } catch (CarCreationException e) {
            System.err.println(e.getMessage());
        }
    }
```

El coste de este método es: número de veces que el usuario introduzca mal el dato numérico + número indicado por el usuario + número de coches en el fichero.

```java
public static double[] obtenerDatos(final String pathname, final Vector<Coche> coches) throws FileNotFoundException {
    /* leemos un entero el cual será el número de puntos de interés (numPOIs) */
    int leído;
    do {
        leído = leer.entero("¿Cuántos puntos de interés hay?");
        /* este entero debe ser positivo */
        if (leído <= 1) System.err.println("Debe haber al menos 2 puntos de interés.");
    } while (leído <= 1);

    /* una vez vemos que cumple las condiciones, nos lo quedamos como el número de POIs */
    final int numPOIs = leído;

    /* Generamos las distancias entre los POIs, entre numPOIs hay una distancia menos, entre 2 puntos hay 1 recta */
    final double[] distanciasPOIs = new double[numPOIs - 1];
    for (int i = 0; i < distanciasPOIs.length; i++)
        /* Seleccionamos una distancia en km aleatoria entre [20, 60) entre dos POIs (i - 1, i) */
        distanciasPOIs[i] = Constantes.DISTANCIAMÍNIMA + Math.random() * (Constantes.DISTANCIAMÁXIMA - Constantes.DISTANCIAMÍNIMA);

    /* leemos los coches */
    IO.leerCoches(pathname, coches);

    return distanciasPOIs;
}
```

El coste de este método es: número de coches * número de distancias + número de coches que terminan * log (número de coches que terminan). Donde número de coches que terminan <= número de coches.

```
 */
public static Vector<Coche> run(final Vector<Coche> coches, final double[] distanciasPOIs, final Vector<Coche> cochesQueNoTerminan) {
    final Vector<Coche> cochesQueTerminan = new Vector<>();

    /* Hacemos que los coches recorran la suma de todas las distancias */
    /* coste número de coches * número de distancias */
    for (Coche coche : coches) {
        /* coste n, siendo n el número de distancias (la longitud del array) */
        recorrer(distanciasPOIs, coche);

        /* si al coche le queda gasolina, ha conseguido terminar el recorrido */
        if (coche.getCapacidadActual() >= 0)
            cochesQueTerminan.add(coche);
        else
            cochesQueNoTerminan.add(coche);
    }

    /* ordenamos por consumo y devolvemos ese Vector */
    /* coste n log n, siendo n el número de coches que terminan (la longitud del Vector) */
    return Ordenar.ordenarPorConsumo(cochesQueTerminan);
}
```

El coste de este método es: número de coches ordenados + número de coches que no terminan. Es decir, número de coches totales.

```
 */
public static void mostarResultados(final Vector<Coche> cochesOrdenados, final Vector<Coche> cochesQueNoTerminan) {
    System.out.printf("\n%-35s : Consumo\n", "Modelo");

    System.out.println("Los coches que sí han llegado al final son, ordenados por consumo:");
    for (Coche coche : cochesOrdenados)
        System.out.println(coche);

    System.out.println();

    System.out.println("Los coches que no han conseguido llegar al final son:");
    for (Coche coche : cochesQueNoTerminan)
        System.out.println(coche);
}
```

Asumimos que todas las operaciones tienen un coste constante (coste 1), menos las llamadas a funciones que tienen un coste variable.

a = 2, b = 2, k = 0

a > b^k → T(n) pertenece O($n^{log_b a}$) = O(n)
Donde n es el tamaño del array distancias.

```
private static void recorrer(final double[] distancias, int limInferior, int limSuperior, final Coche coche) {
    /* cuando sólo nos quede un elemento */
    if (limInferior == limSuperior) {
        /* intentamos recorrer esa distancia */
        coche.recorrer(distancias[limInferior]);
    } else {
        /* dividimos por la mitad, b = 2 */
        /* coste de división 1, k = 0 */
        final int mid = (limInferior + limSuperior) / 2;

        /* hacemos dos llamadas, a = 2 */
        recorrer(distancias, limInferior, mid, coche);

        /* si ha sido capaz de recorrer hasta aquí, intentamos seguir */
        if (coche.getCapacidadActual() > 0)
            recorrer(distancias, mid + 1, limSuperior, coche);
    }
}
```

Todas las operaciones básicas tienen un coste constante que se considera igual a 1. Mientras que las llamadas a funciones externas pueden tener un coste diferente.

a = 2, b = 2, k = 1

a = b^k →T(n) pertenece O(nlog n) = O(nlog n)

```java
private static Vector<Coche> mergeSort(final Vector<Coche> coches, final int limInferior, final int limSuperior) {
    Vector<Coche> cochesOrdenados;

    /* si sólo queda un elemento, ya está ordenado */
    if (limInferior == limSuperior) {
        cochesOrdenados = new Vector<>(1);
        cochesOrdenados.add(coches.get(limInferior));
    } else {
        /* debemos ordenar cada mitad
         * División del problema: b = 2
         * Coste de división: 1 */
        final int mitad = (limInferior + limSuperior) / 2;

        /* Coste de combinación: n + m, donde m = n, por lo que el coste es 2n, y la complejidad es O(n) */

        /* Devolvemos la combinación de */
        cochesOrdenados = combinar(
                            /* Hacemos 2 llamadas recursivas, las cuales se ejecutan ambas: a = 2 */
                            /* la primera mitad ordenada */
                            mergeSort(coches, limInferior, mitad),
                            /* la segunda mitad ordenada */
                            mergeSort(coches, mitad + 1, limSuperior)
                            );
    }

    /* g(n) pertenece a la complejidad O(n) -> k = 1 */
    return cochesOrdenados;
}
```

El coste de las operaciones básicas, como la asignación de valores a variables o la comparación de valores, es constante y tiene un valor unitario. No obstante, el coste de las llamadas a otras funciones es variable.
El coste de este es n + m, siendo n la longitud del primer vector y m la del segundo.

```java
private static Vector<Coche> combinar(final Vector<Coche> primerVector, final Vector<Coche> segundoVector) {
    /* nos guardamos los tamaños de los vectores, para que no tengamos que hacer muchas llamadas */
    final int end1 = primerVector.size(), end2 = segundoVector.size();

    /* creamos vector con tamaño igual a la suma de los tamaños de los dos vectores a unir */
    final Vector<Coche> cochesOrdenados = new Vector<>(end1 + end2);

    int índice1 = 0, índice2 = 0;

    while (índice1 < end1 && índice2 < end2) {
        Coche c1 = primerVector.get(índice1);
        Coche c2 = segundoVector.get(índice2);

        /* añadimos el coche con menor capacidad al vector ordenado
         * y avanzamos en ese vector
         */
        if (c1.compareTo(c2) < 0) {
            cochesOrdenados.add(c1);
            índice1++;
        } else {
            cochesOrdenados.add(c2);
            índice2++;
        }
    }

    /* sólo uno de estos se ejecutará, se encargan de terminar de añadir el resto de coches al vector */
    for (; índice1 < end1; índice1++)
        cochesOrdenados.add(primerVector.get(índice1));

    for (; índice2 < end2; índice2++)
        cochesOrdenados.add(segundoVector.get(índice2));

    return cochesOrdenados;
}
```