Practica Backtracking

Participación

-Pablo Carbayo: 50%
-Sergio Cornejo: 50%

Algoritmo de backtraking que resuelve el problema

```java
private static void combinaciones(int[] soluciones, ArrayList<Vaca> listaVacas, double lecheMinima, int k,
        double lecheParcial, int espacio, int espacioParcial, ArrayList<int[]> listaSoluciones) {
    if (lecheParcial >= lecheMinima) { /* Teste de solucion */
        int[] copia = new int[soluciones.length];
        System.arraycopy(soluciones, srcPos:0, copia, destPos:0, soluciones.length);
        listaSoluciones.add(copia);
    } else {
        for (int i = k; i < listaVacas.size(); i++) {
            /* El indice comienza en K para iterar sobre las vacas restantes que no han sido exploradas */
            if (espacioParcial + listaVacas.get(i).getEspacio() <= espacio) { /* Test de Factibilidad/Fracaso */
                soluciones[i] = 1; /* Si es factible se agrega la vaca al vector de soluciones */
                /* Con el vector en el que hemos añadido una vaca para la solucion, volvemos a llamar al metodo para que busque
                mas vacas hasta que superemos la leche minima pero sin pasarnos del espacio que tenemos */
                /*Invocamos al metodo sumando en uno la etapa (i + 1), y poniendo el contador de leche parcial como la suma
                 de leche con la que hemos empezado mas la que produce la vaca que hemos introducido en el vector de soluciones.
                 Con el espacio lo mismo, le restamos al que tenemos el que ocupa la vaca introducida */
                combinaciones(soluciones, listaVacas, lecheMinima, i + 1,
                        (lecheParcial + listaVacas.get(i).getLeche()),
                        espacio, (espacioParcial + listaVacas.get(i).getEspacio()), listaSoluciones);
                soluciones[i] = 0; /* Eliminamos la vaca despues de la recursion para explorar mas soluciones */
            }
        }
    }
}
```

La complejidad de este algoritmo podemos decir que es exponencial, ya que la cantidad de subconjuntos que se pueden formar teniendo un conjunto principal de tamaño N es 2^N. Además, como en este algoritmo la recursión se hace en cada elemento de la lista de vacas, la llamada recursiva se hace un total de 2^N veces, con lo que en verdad, si hablamos en términos de complejidad asintótica, la complejidad del algoritmo es de **O(2^N * N)**, ya que como se mencionó anteriormente, la llamada recursiva se hace N veces. Por otro lado, como las copias que se hacen de ArrayList y el acceso a los elementos tienen un coste de tiempo constante, no afecta a la complejidad del algoritmo, con lo que se mantiene en una complejidad **O(2^N * N)**.

Con esto podemos llegar a la conclusión de que cuanto más aumentemos el tamaño de nuestra lista de vacas, mayor tardara el algoritmo en resolver el problema, siendo en nuestro caso el valor de la complejidad con un N = 30 de 3.22 * 10^10, en cambio si por ejemplo hubiera 10 vacas más, es decir, un N = 40, nuestra complejidad tendría el valor de 4.4 * 10^13, es decir, un aumento considerable.

De todas formas, no se puede determinar de forma teórica a partir de que numero de vacas el problema se hace "intratable", ya que depende de las características de la maquina en la que se ejecuta la solución, con lo que se necesitan pruebas empí6ricas

```java
private static void mejorCombinacion(File archivo, double lecheMinima, int espacio) throws FileNotFoundException {

    ArrayList<Vaca> listaVacas = LeerArchivo.listaVacas(archivo); /* Lista donde estan todas las vacas */
    int[] soluciones = new int[listaVacas.size()]; /* Vector donde guardaremos las soluciones parciales */
    ArrayList<int[]> listaSoluciones = new ArrayList<int[]>(); /* ArrayList que contendra todos los vectores que son solucion */
    combinaciones(soluciones, listaVacas, lecheMinima, k:0, lecheParcial:0, espacio, espacioParcial:0, listaSoluciones);
    double lecheProducida = 0; /* Variable para almacenar la leche de cada solucion */
    int espacioGastado = 0; /* Variable para almacenar el espacio que se ocupa en cada solucion */
    int mejorEspacio = 99999; /* Variable que se usara para comprobar cual es la mejor solucion (Se incializa con un valor alto) */
    double mejorLeche = 0; /* Variable que se usara para comprobar cual es la mejor solucion */
    int[] mejorSolucion = new int[listaVacas.size()]; /* Vector que guardara la mejor solucion encontrada */
    int[] vector = new int[30]; /*Vector auxiliar donde se iran guardando las soluciones parciales */

    if (listaSoluciones.size() == 0) { /* Si la lista de soluciones esta vacia no hay soluciones posibles */
        System.out.println("No hay combinacion posible para " + lecheMinima + " litros de leche y " + espacio
                + " hectareas.\n");
    } else { /* Si la lista de soluciones no esta vacia, buscamos cual es la mejor */
        for (int i = 0; i < listaSoluciones.size(); i++) {
            /* Reiniciamos las variables de leche para cada solucion */
            lecheProducida = 0;
            espacioGastado = 0;
            vector = listaSoluciones.get(i); /* Obtenemos el vector en la posición i */
            /* Ahora recorremos el vector solucion que hemos obtenido de la lista y hacemos los calculos de la leche que
             produce y del espacio que ocupa */
            for (int j = 0; j < vector.length; j++) {
                if (vector[j] == 1) { /* Si el valor en esa posicion es 1, la vaca de dicha posicion es una vaca de nuestra solucion
                    parcial */
                    /* Hacemos la suma de la leche producida y del espacio gastado */
                    lecheProducida += listaVacas.get(j).getLeche();
                    espacioGastado += listaVacas.get(j).getEspacio();
                }
            }
            /* Una ve que tenemos cuanta leche produce y cuanto espacio se gasta en la solucion que hemos cogido, comprobamos
            si es la mejor comparando con las variables que hemos creado antes de solucion mejor */
            if (lecheProducida >= mejorLeche) { /* Si se cumple la condicion nuestor vector que guarda la mejor solucion
                se convierte en la solucion que acabamos de comprobar, y las variables de mejor solucion se actualizan */
                mejorLeche = lecheProducida;
                mejorEspacio = espacioGastado;
                mejorSolucion = vector;
            }
        }
    }
    /* Una vez que tenemos la mejor solucion la imprimimos junto la leche que se ha producido y el espacio que se ha gastado */
    System.out.println(x:"La mejor combinacion es:");
    for (int i = 0; i < mejorSolucion.length; i++) {
        if (mejorSolucion[i] == 1) {
            System.out.print("Vaca: " + listaVacas.get(i).getId() + " ");
        }
    }
    System.out.println("con " + String.format(format:"%.1f", mejorLeche) + " litros en " + mejorEspacio + " hectareas.\n");
}
```

En este caso, este método tiene una complejidad de **O(N^2 * N * N^2)**, o lo que es lo mismo **O(N^2 * N^3)**, ya que al principio invoca al método de combinaciones, el cual por sí mismo ya tiene la complejidad de O(N^2 * N), y una vez se han encontrado todas las combinaciones, se busca la mejor, aplicando dos bucles for anidados, lo que va a dar una complejidad de N^2, dando como resultado la complejidad total del algoritmo que hemos mencionado anteriormente, O(2^N * N^3)

```java
private static void primeraCombinacion(File archivo, double lecheMinima, int espacio) throws FileNotFoundException {

    ArrayList<Vaca> listaVacas = LeerArchivo.listaVacas(archivo); /* Lista donde estan todas las vacas */
    int[] soluciones = new int[listaVacas.size()]; /* Vector donde guardaremos las soluciones parciales */
    ArrayList<int[]> listaSoluciones = new ArrayList<int[]>(); /* ArrayList que contendra todos los vectores que son solucion */
    combinaciones(soluciones, listaVacas, lecheMinima, k:0, lecheParcial:0, espacio, espacioParcial:0, listaSoluciones);

    if (listaSoluciones.size() == 0) { /* Si la lista de soluciones esta vacia no hay soluciones posibles */
        System.out.println("No hay combinacion posible para " + lecheMinima + " litros de leche y " + espacio
                + " hectareas.\n");
    } else { /* Si la lista de soluciones no esta vacia, buscamos la primera de todas las posibles soluciones */
        int[] primeraSolucion = listaSoluciones.get(index:0); /* Guardamos la primera solucion que se ha introducido en la
        lista de soluciones en nuestro vector de primeraSolucion */
        System.out.print("Conjunto " + (1) + ":\n");
        /* Creamos las variables para visualizar cuanta leche se produce y cuanto espacio se gasta */
        double lecheProducida = 0;
        int espacioGastado = 0;
        /* Recorremos nuestro vecotr de primeraSolucion haciendo las cuentas de la leche que se produce y el espacio que se gasta */
        for (int j = 0; j < primeraSolucion.length; j++) {
            if (primeraSolucion[j] == 1) {
                lecheProducida += listaVacas.get(j).getLeche();
                espacioGastado += listaVacas.get(j).getEspacio();
                System.out.print("Vaca:" + listaVacas.get(j).getId() + " ");
            }
        }
        /* Imprimimos la primera solucion junto a sus datos */
        System.out.println(
                "\nLeche producida en el conjunto " + 1 + ": " + String.format(format:"%.1f", lecheProducida)
                        + " litros en " + espacioGastado + " hectareas");
        System.out.println();
    }
}
```

Este método tiene una complejidad de **O(2^N * N**), que es la complejidad que tiene el método de combinaciones. Las demas operaciones que se hacen en el método, como puede ser el uso de un bucle for, que tiene complejidad O(N) o la suma de variables, comprobación de condiciones, etc, tienen coste constante, con lo que no afectaran a nuestra complejidad.

```java
private static void combinacionVacas(File archivo, double lecheMinima, int espacio) throws FileNotFoundException {

    ArrayList<Vaca> listaVacas = LeerArchivo.listaVacas(archivo); /* Lista donde estan todas las vacas */
    int[] soluciones = new int[listaVacas.size()]; /* Vector donde guardaremos las soluciones parciales */
    ArrayList<int[]> listaSoluciones = new ArrayList<int[]>(); /* ArrayList que contendra todos los vectores que son solucion */
    combinaciones(soluciones, listaVacas, lecheMinima, k:0, lecheParcial:0, espacio, espacioParcial:0, listaSoluciones);

    if (listaSoluciones.size() == 0) { /* Si la lista de soluciones esta vacia no hay soluciones posibles */
        System.out.println("No hay combinacion posible para " + lecheMinima + " litros de leche y " + espacio
                + " hectareas.\n");
    } else { /* Si la lista no esta vacia, vamos imprimiendo una a una las soluciones junto a sus datos */
        /* Recorremos la lista de soluciones usando variables auxiliares para poder guardar los datos de la leche que se
         produce y el espacio que ocupa */
        for (int i = 0; i < listaSoluciones.size(); i++) {
            double lecheProducida = 0;
            int espacioGastado = 0;
            int[] vector = listaSoluciones.get(i); // Obtenemos un vector solucion, en este caso el de la posición i
            System.out.print("Conjunto " + (i + 1) + ":\n");
            for (int j = 0; j < vector.length; j++) { /* Recorremos el vector de solucion */
                /* Si la posicion i esta en 1, significa que la vaca i de la lista de vacas, es una vaca perteneciente a la
                solucion */
                if (vector[j] == 1) {
                    /* Vamos actualizando los valores de la leche y el espacio cada vez que haya una vaca solucion */
                    lecheProducida += listaVacas.get(j).getLeche();
                    espacioGastado += listaVacas.get(j).getEspacio();
                    System.out.print("Vaca:" + listaVacas.get(j).getId() + " ");
                }
            }
            /* Una vez tenemos todos los datos, imprimimos la solucion que hemos obtenido junto a sus datos */
            System.out.println(
                    "\nLeche producida en el conjunto " + i + ": " + String.format(format:"%.1f", lecheProducida)
                            + " litros en " + espacioGastado + " hectareas");
            System.out.println();
        }
    }
}
```

Aquí la complejidad del algoritmo vuelve a ser de **O(2^N * N^3)**, ya que como estamos invocando al método de "combinaciones" cuya complejidad ya es de O(2^N *N), y además tenemos dos bucles for anidados cuya complejidad es O(N^2), pues obtendríamos una complejidad de O(2^N * N * N^2), lo que daría de forma simplificada la complejidad que he dicho anteriormente. Las demas operaciones del método no influirían en la complejidad ya que son operaciones cuyo valor es constante, como puede ser la comprobación de condicionales, creación de variables, etc.