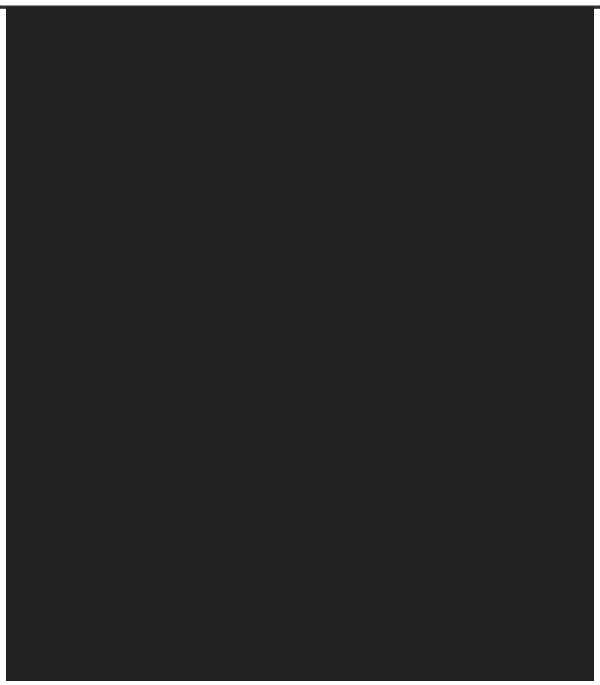


Práctica 3

Programación dinámica

ALEJANDRO RODRIGUEZ MORENO
ÁNGEL LÓPEZ CAPEL
JOSÉ LUIS PÉREZ LÓPEZ

1 ESTUDIO DE LA IMPLEMENTACIÓN

Todos los algoritmos de esta práctica (menos el último) están basados en el esquema de programación dinámica, este esquema se podría considerar como una mezcla de los esquemas de divide y vencerás y el voraz. Usa recursividad como divide y vencerás, pero no siempre hace la llamada recursiva, esto lo dicta una función que detecta si la solución a un problema ya ha sido calculada anteriormente dado a que tenemos una tabla donde guardamos todos los valores para su posterior uso.

La complejidad de estos algoritmos es de O(n*p) donde n es el peso del elemento y p la capacidad de la mochila.

El último algoritmo lo hemos hecho con el esquema voraz, ya que programación dinámica era increíblemente ineficiente y el guion también nos pedía una solución voraz. Este ordena los objetos según el valor hasta que se queda sin espacio para llevar un objeto entero, en ese momento, lo fraccionará para y llenará por completo la mochila. Este método es más rápido y bastante más sencillo de implementar.

1.1 IMPLEMENTACIONES

1.1.1 Mochila 0-1

```
static int knapSack(int W, int wt[], int val[], int n)
    {
        int i, w;
        int K[][] = new int[n + 1][W + 1];

        for (i = 0; i<= n; i++) {
            for (w = 0; w<= W; w++) {
                if (i == 0 || w == 0)
                    K[i][w] = 0;
                else if (wt[i - 1]<= w)
                    K[i][w] = max(val[i - 1] + K[i - 1][w -
wt[i - 1]], K[i - 1][w]);
                else
                    K[i][w] = K[i - 1][w];
            }
        }

        return K[n][W];
    }
```

1.1.2 Mochila 0-1 (pesos no exactos)

```
static double knapsackCaso2(int[] val, double[] weight, int w) {

        double[][] k = new double[val.length + 1][w + 1];

        boolean sorted = false;
        double temp;
        int temp2;
        while(!sorted) {
            sorted = true;
            for(int i=0; i<weight.length-1; i++) {
                 if (weight[i] > weight[i+1]) {
                        temp = weight[i];
                        weight[i] = weight[i+1];
                        weight[i+1] = temp;

                        temp2 = val[i];
                        val[i] = val[i+1];
                        val[i+1] = temp2;

                        sorted = false;
                 }
            }
        }

        for (int i = 0; i <= val.length; i++) {
            for (int j = 0; j <= w; j++) {

                if (i == 0 || j == 0) {
                    k[i][j] = 0;
                    continue;
                }

                if (j - weight[i - 1] >= 0) {
                    k[i][j] = Math.max(k[i - 1][j], k[i - 1][j-1] + val[i - 1]);
                }

                else {
                    k[i][j] = k[i - 1][j];
                }
            }
        }
        for(int i = 0; i < w+1; i++) {
            System.out.print(i+"\t");
        }
        System.out.println();
        for(int i = 0; i < w+1; i++) {
            System.out.print("--------");
        }
        System.out.println();


        for(int i = 0; i < k.length; i++) {
            for(int j = 0; j < k[0].length; j++) {
                System.out.print(k[i][j]+"\t");
            }
            System.out.println("");

        }
        System.out.println();
        return k[val.length][w];
    }
```

1.1.3 Mochila 0-1 (Objetos infinitos)

```
static int unboundKnapsack(int pesoMochila, int n, int[]
val, int[] peso){

        int dp[] = new int[pesoMochila + 1];

        for(int i = 0; i <= pesoMochila; i++){
            for(int j = 0; j < n; j++){
                if(peso[j] <= i){
                    dp[i] = max(dp[i], dp[i - peso[j]] +
                                val[j]);
                }
            }
        }
        return dp[pesoMochila];
    }
```

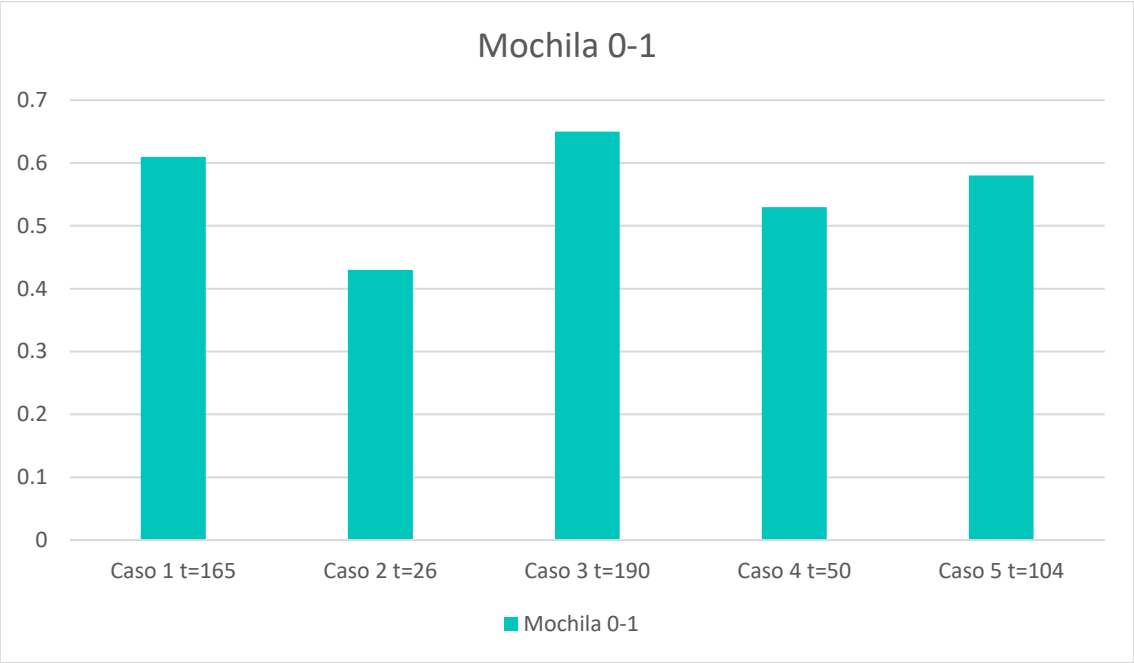
1.1.4 Mochila fraccional

```
static double knapsack(int[] peso, int[] valor, int cap){
        ValorObjeto[] iVal = new ValorObjeto[peso.length];

        for (int i = 0; i < peso.length; i++) {
            iVal[i] = new ValorObjeto(peso[i], valor[i], i);
        }

        Arrays.sort(iVal, new Comparator<ValorObjeto>() {
            @Override
            public int compare(ValorObjeto o1, ValorObjeto o2)
            {
                return o2.coste.compareTo(o1.coste);
            }
        });

        double valorTotal = 0;

        for (ValorObjeto i : iVal) {

            int pesoAct = (int)i.wt;
            int valorAct = (int)i.val;

            if (cap - pesoAct >= 0) {
                cap = cap - pesoAct;
                valorTotal += valorAct;
            }
            else {
                double fraction = ((double)cap / (double)pesoAct);
                valorTotal += (valorAct * fraction);
                cap = (int)(cap - (pesoAct * fraction));
                break;
            }
        }

        return valorTotal;
    }
```

2 ESTUDIO TEÓRICO Y EXPERIMENTAL

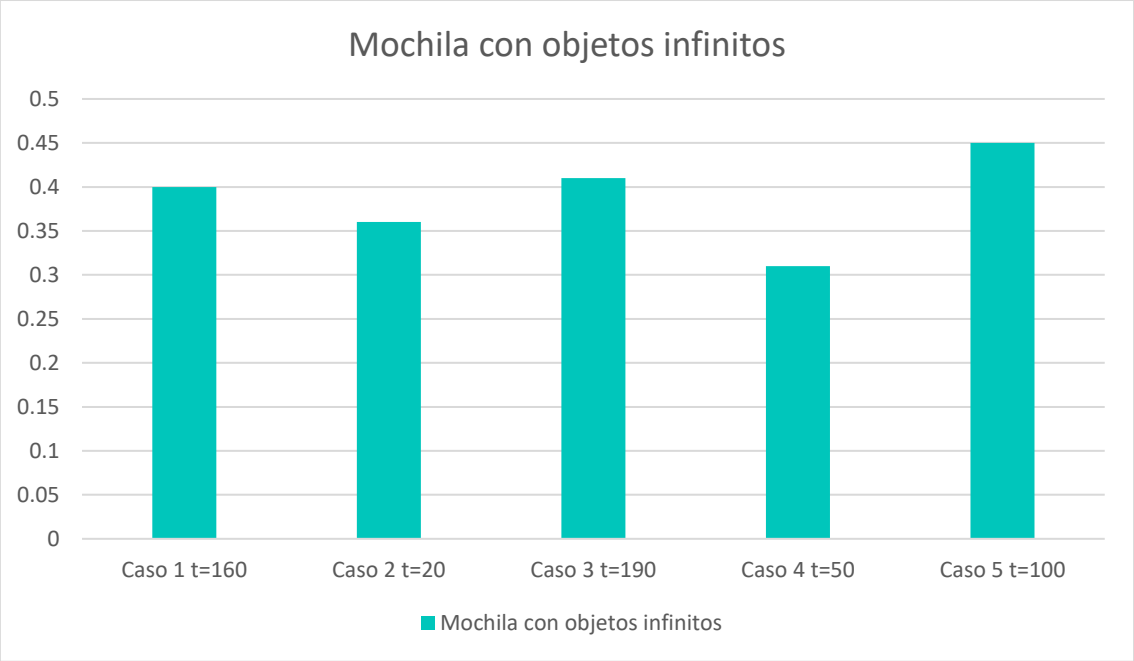
Para medir los tiempos del primer algoritmo hemos usado los sets de datos en https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

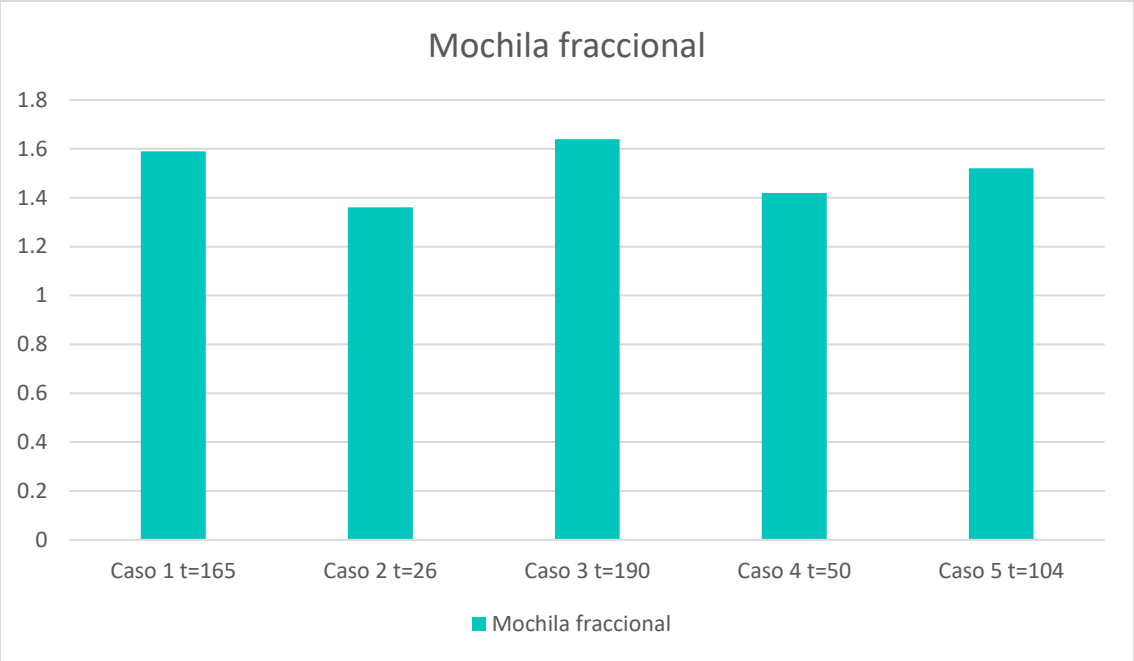
Cogimos cinco al azar y estos fueron nuestros resultados:



Como sería de esperar, los tiempos aumentan dependiendo del tamaño de la mochila ya que tiene que meter más objetos, y, por tanto, más operaciones.

Para el resto de los algoritmos, dado a que tienen características distintas y no testean lo mismo, tenemos que generar los datos de forma aleatoria.





Aquí también comprobamos que el tiempo aumenta con el tamaño de la mochila.

Los tres primeros algoritmos rellenan sus tablas de la misma manera ya que hay cambios mínimos en su funcionamiento. Y el algoritmo de la mochila fraccional no tiene tabla directamente ya que usa un esquema voraz.

3 BIBLIOGRAFÍA

Transparencias de clase

https://stackoverflow.com/questions/62200415/solving-fractional-knapsack-problem-with-dynamic-programming

https://stackoverflow.com/questions/63045448/unbounded-knapsack-and-classical-knapsack-comparison

https://stackoverflow.com/questions/50221844/recursive-solution-of-unbounded-knapsack-using-logic-of-0-1-knapsack