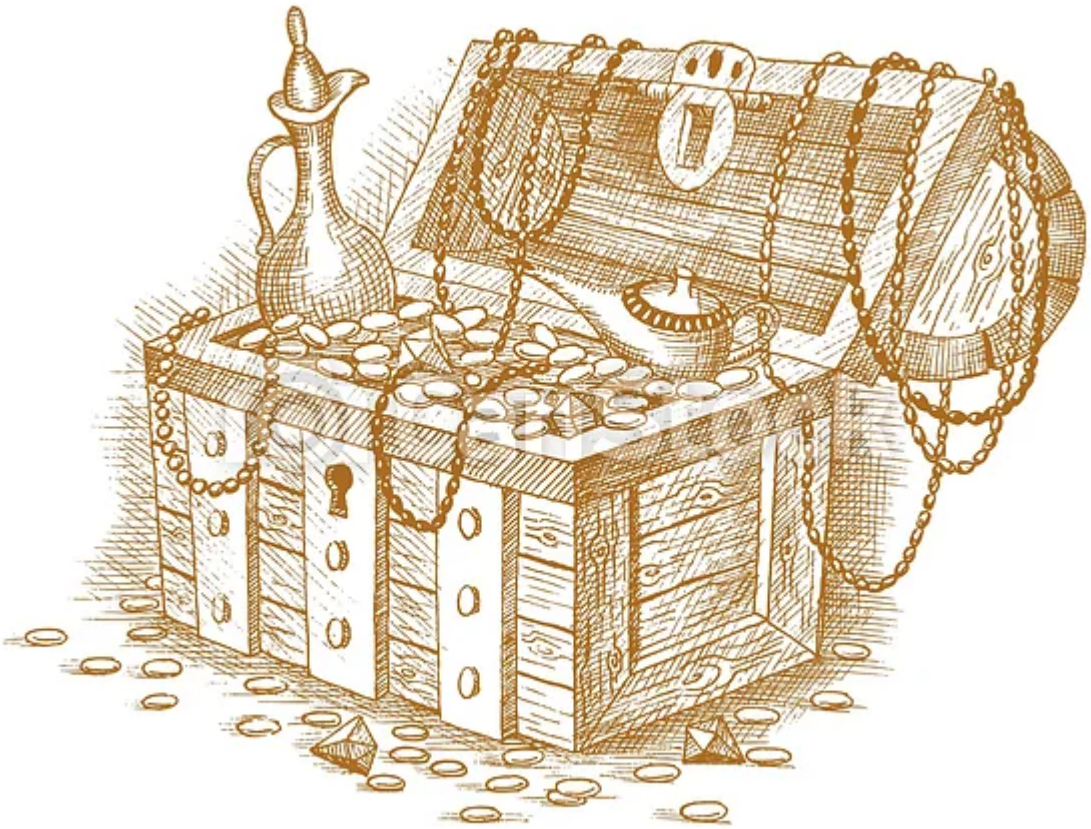
Práctica 3: Programación dinámica



Grupo:

- Alejandro Dámaso Pastor
- Francisco Escamilla Vizcaino

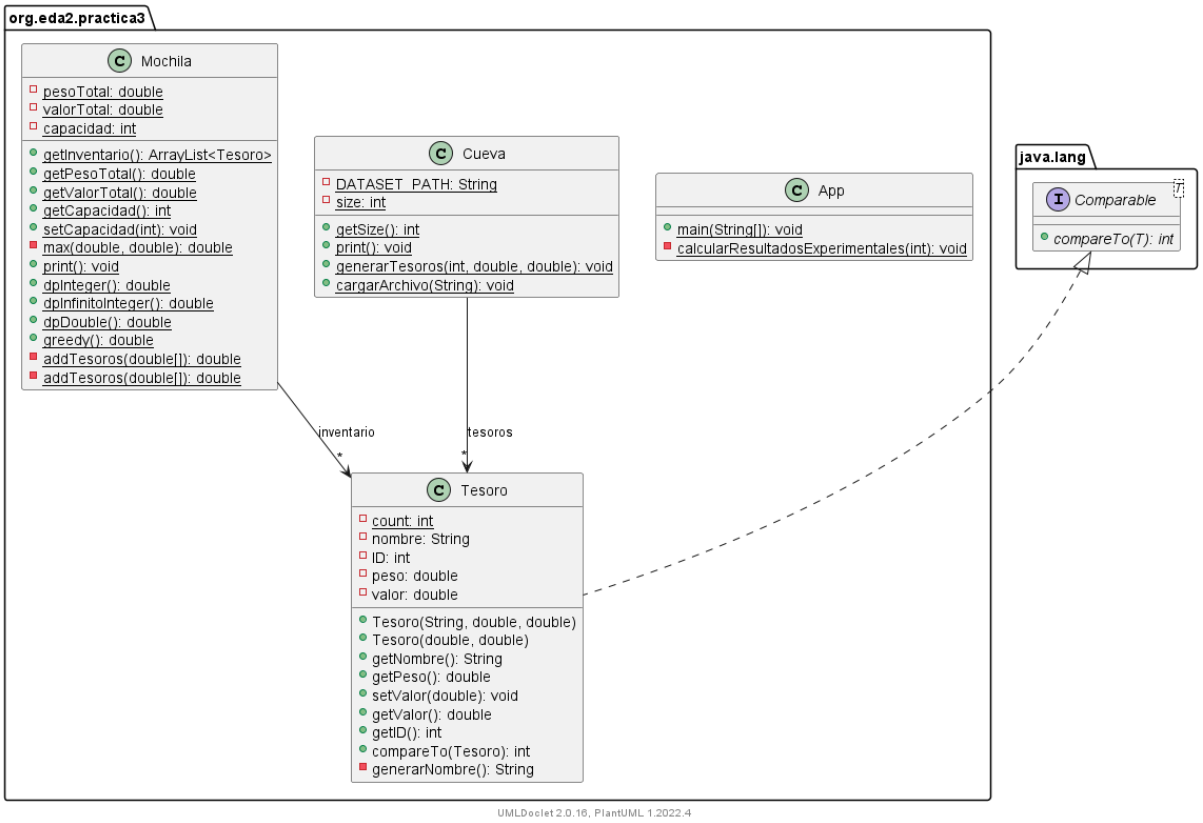
Índice

Introducción

La programación dinámica es un método usado principalmente para reducir el tiempo de ejecución de un algoritmo mediante subproblemas y subestructuras óptimas, para esto tendremos en cuenta los siguientes pasos:

- Dividimos el problema en subproblemas más pequeños.
- resolvemos estos problemas.
- unimos las soluciones óptimas para generar la solución final.

Además normalmente suele seguir uno de estos enfoques, **Top-down** (Divide el problema final en subproblemas y estos se solucionan) y **Bottom-up** (todos los problemas se resuelven por adelantado aunque resulta poco intuitivo)

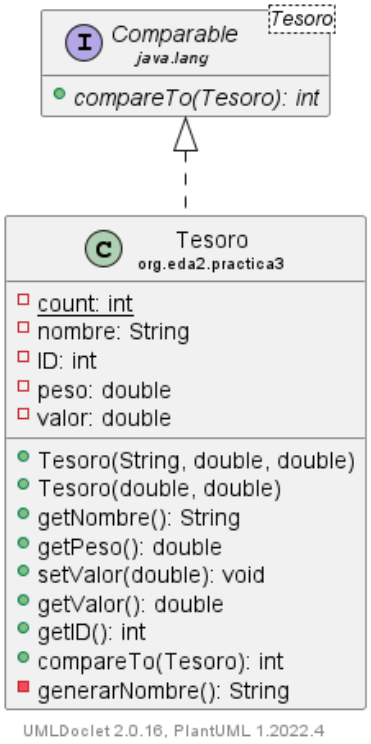


Estructura del proyecto

App

Clase principal donde ejecutaremos las órdenes.

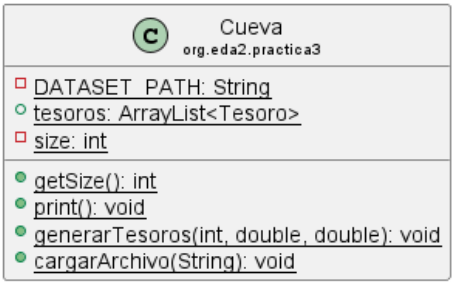
Tesoro



Clase que guarda los datos de nombre, peso y valor de cada tesoro generado o cargado en la clase Cueva.

- Implementa la interfaz ***comparable*** para permitir ordenación por medio de Collection.Sort().
- Posee un método privado llamado ***generarNombre*** que permite la generación de nombres aleatorios, aunque el nombre no tiene uso a parte del visual.

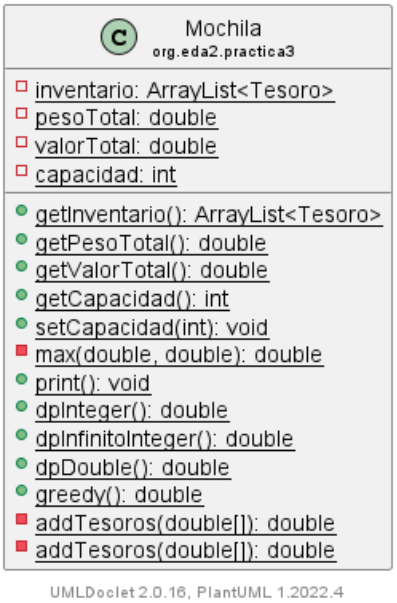
Cueva



Clase que se encarga de generar o cargar la lista de tesoros que podemos introducir en nuestra mochila.

- **generarTesoros**: Nos permite generar los tesoros que hay dentro de la cueva.
- **cargarArchivo**: Carga archivos en el formato dado por los datasets.

Mochila



Clase que contiene una lista de tesoros seleccionados, peso y valores totales y la capacidad actual de la mochila.

- Los 2 métodos ***addTesoros*** se encargan de añadir los tesoros según la matriz o array que devuelven los algoritmos de programación dinámica.
- El algoritmo greedy introduce en el inventario los tesoros sobre la marcha.

Estudio de implementación

Explicaremos la funcionalidad de los métodos que implementan los algoritmos que se nos piden en esta práctica.

Caso 1 - dpInteger()

Algoritmo básico de programación dinámica que implementa el primer caso que se nos pide en el trabajo a desarrollar para esta práctica.

Este método obtiene el resultado tratando los pesos de cada tesoro de forma exacta..

La primera parte del método verifica que la lista de tesoros a recorrer no esté vacía y limpia el inventario de tesoros que ya se encuentren dentro por posibles ejecuciones de código previas.

```
if (Cueva.tesoros.isEmpty())
        return -1;
inventario.clear();
```

Esta primera parte es igual para todos los métodos que implementan los algoritmos por lo que de ahora en adelante no la volveremos a mencionar.

En esta parte inicializamos una matriz llamada dp con la cantidad de tesoros y la capacidad de nuestra mochila. Esto se realiza de esta manera puesto que este método realmente crea una tabla de valores.

```
double[][] dp = new double[Cueva.getSize() + 1][capacidad + 1];
double valor = 0;
int peso = 0;
Tesoro t;
```

Además creamos variables auxiliares que aportan claridad al código.

Ciclo del algoritmo

Esta parte es la que define el algoritmo.

Se trata de un bucle for anidado que contiene varias condiciones que dan forma a la matriz resultante.

```
1       for (int i = 0; i <= Cueva.getSize(); i++) {
            if (i > 0) {
                t= Cueva.tesoros.get(i - 1);
                valor = t.getValor();
                peso = (int) t.getPeso();
            }
            for (int j = 0; j <= capacidad; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                    continue;
                }
2               if (peso <= j)
                    dp[i][j] = max(valor + dp[i - 1][j - peso], dp[i - 1][j]);
3               else
                    dp[i][j] = dp[i - 1][j];
            }
        }
```

Análisis del proceso:

1. La tabla dinámica que genera este método en dp[][] comienza su primera fila y columna por 0 y solo obtenemos un tesoro y su valor y peso a partir de 1>=1.

2. Cuando el peso del tesoro actual no sobrepase la capacidad actual marcada por j procederemos a introducir en esta celda de la tabla el valor mayor entre el valor del tesoro actual sumado al valor de la celda que se encuentra en la anterior fila y la columna [j-peso] y el valor de la anterior fila en la misma columna de capacidad.

3. Si se rebasa la capacidad aceptamos el valor anterior directamente.

Una vez completado el ciclo se devuelve finalmente el método sobrecargado ***addTesoros(double matriz[][])*** que se encarga de añadir los tesoros seleccionados basándose en los valores de la matriz y nos devuelve el resultado final, que se encuentra ahora mismo en **dp[númeroTesoros][capacidad]**.

Caso 2 - dpInfinitoInteger()

Algoritmo de programación dinámica que implementa el segundo caso que se nos pide en el trabajo a desarrollar para esta práctica.

Este método obtiene el resultado en el caso de que los tesoros puedan volver a ser seleccionados aunque ya lo hayan sido previamente.

Inicializamos un array llamado dp con la capacidad de nuestra mochila. En este caso no es necesario una matriz para lograr el objetivo de este algoritmo.

```
double[] dp = new double[capacidad + 1];
double valor = 0;
int peso = 0;
Tesoro t;
```

Ciclo del algoritmo

Esta parte es la que define el algoritmo.

Se trata de un bucle for anidado que contiene varias condiciones que dan forma a la matriz resultante.

```
1                for (int i = 0; i <= capacidad; i++) {
                     for (int j = 0; j < Cueva.getSize(); j++) {
                              t= Cueva.tesoros.get(j);
                              valor = t.getValor();
                              peso = (int) t.getPeso();
2                             if (peso <= i)
                                      dp[i] = max(valor + dp[i - peso], dp[i]);
                     }
            }
```

Análisis del proceso:

1. Esta vez recorreremos las columnas de capacidad en busca de aquellos valores que cumplan nuestras condiciones.
2. La condición es igual que en el dpInteger() pero adaptado al array de 1D.

Devuelve el método sobrecargado ***addTesoros(double array[])*** que se encarga de añadir los tesoros seleccionados basándose en los valores en el array y nos devuelve el resultado final, que se encuentra ahora mismo en **dp[capacidad]**.

Caso 3 - dpDouble()

Se nos pregunta qué modificaciones podemos hacer en el primer caso para trabajar sobre pesos que sean números reales positivos (double).

En nuestro caso hemos cambiado la siguiente línea.

```
peso = (int) Math.round(t.getPeso());
```

Ahora la obtención de peso redondea el valor con el que trabaja.

Caso 4 - greedy()

Algoritmo voraz que implementa el cuarto caso que se nos pide en el trabajo a desarrollar para esta práctica.

Se nos pide hacer un algoritmo que pueda encontrar la solución a este problema pero con la posibilidad de fraccionar cada uno de los tesoros y se nos pregunta si una solución greedy sería más razonable en este caso.

En nuestro caso creemos que la solución voraz es mucho más sencilla de implementar y en este caso su complejidad es bastante baja puesto que ni siquiera requiere de un bucle anidado sino que un simple for basta para obtener el resultado deseado y no necesita usar memoria extra para mantener una tabla como en los otros 3 casos.

En esta parte inicializamos una matriz llamada dp con la cantidad de tesoros y la capacidad de nuestra mochila. Esto se realiza de esta manera puesto que este método realmente crea una tabla de valores.

```
double[] cantidad = new double[Cueva.getSize()];
pesoTotal = 0;
valorTotal = 0;
Tesoro t;
```

Además creamos variables auxiliares que aportan claridad al código.

Ciclo del algoritmo

Esta parte es la que define el algoritmo.

Se trata de un bucle for anidado que contiene varias condiciones que dan forma a la matriz resultante.

1

```
for (int i = 0; i < Cueva.getSize(); i++) {
    t = Cueva.tesoros.get(i);
    if (pesoTotal + t.getPeso() <= capacidad) {
        t.setCantidad(1);
        inventario.add(t);
        valorTotal += t.getValor();
        pesoTotal += t.getPeso();
        if (pesoTotal == capacidad)
            break;
```

2

```
    } else {
        t.setCantidad((capacidad - pesoTotal) / t.getPeso());
        inventario.add(t);
        valorTotal += t.getValor() * t.getCantidad();
        pesoTotal += t.getPeso() * t.getCantidad();
        break;

    }
}
```

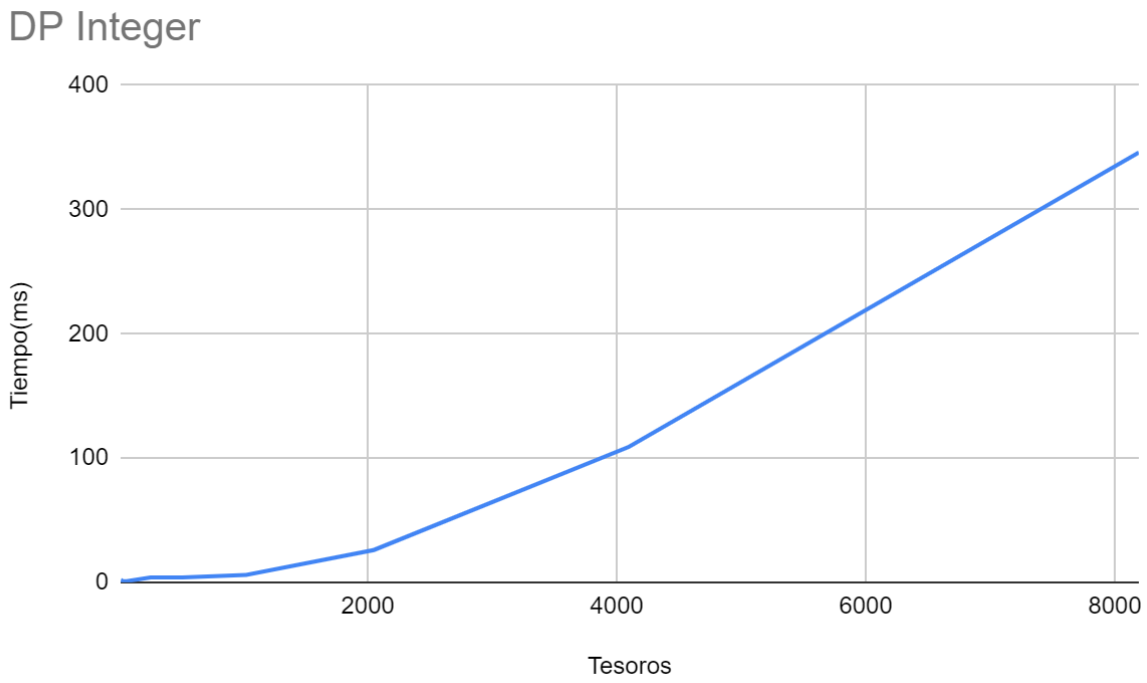
Análisis del proceso:

1. Se recorren todos los tesoros de la lista y si el pesoTotal actual más el peso del tesoro actual es menor a la capacidad de nuestra mochila se añadirá a la cantidad de ese tesoro. Cuando el pesoTotal sea igual a la capacidad se romperá el ciclo.
2. En caso contrario añadiremos una fracción de este.

Una vez completado el ciclo se devuelve directamente el valorTotal obtenido ya que a diferencia de los otros casos en el algoritmo voraz introducimos los tesoros directamente durante el ciclo en proceso.

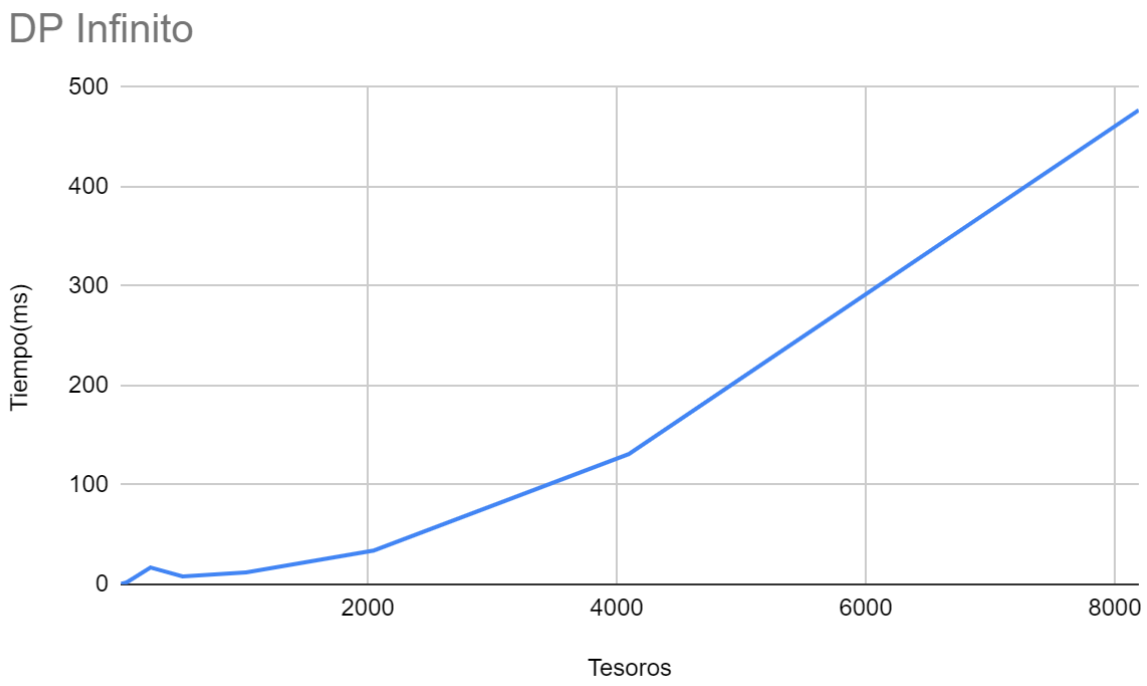
Estudio Teórico

DP Integer



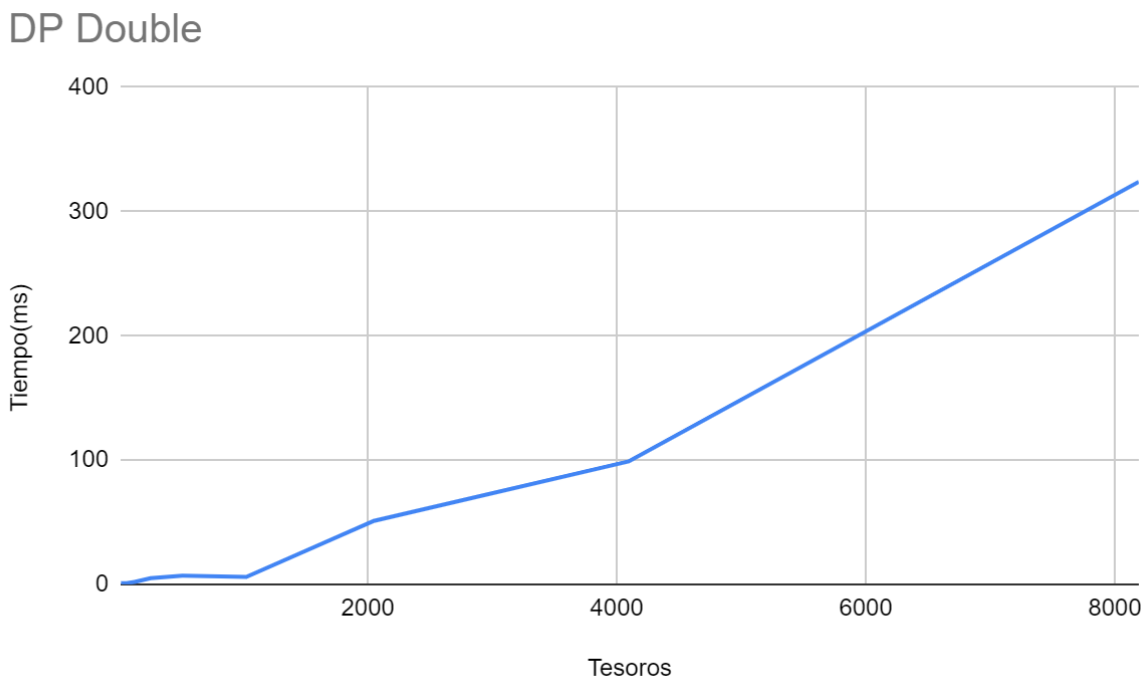
Resultados DP Integer:		
Tesoros	**Capacidad**	**Tiempo(ms)**
16	8	2
32	16	1
64	32	1
128	64	2
256	128	4
512	256	4
1024	512	6
2048	1024	26
4096	2048	109
8192	4096	346

DP Infinito



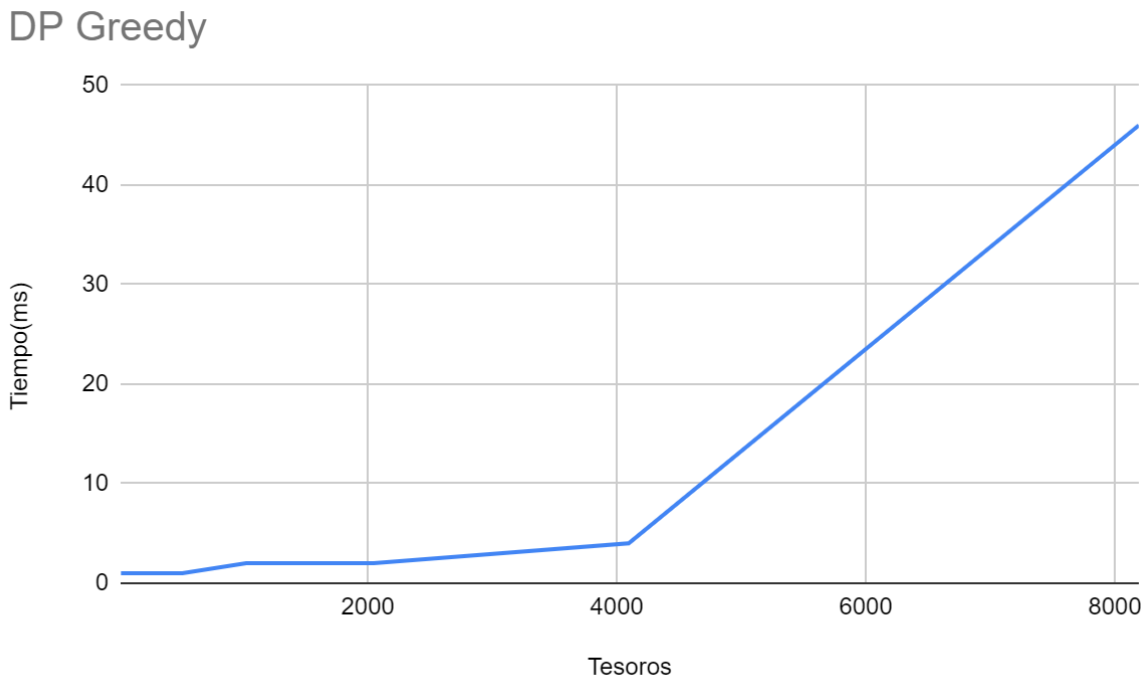
Resultados DP Infinito:		
Tesoros	**Capacidad**	**Tiempo(ms)**
16	8	1
32	16	1
64	32	2
128	64	7
256	128	17
512	256	8
1024	512	12
2048	1024	34
4096	2048	131
8192	4096	477

DP Double



Resultados DP Double:		
Tesoros	**Capacidad**	**Tiempo(ms)**
16	8	1
32	16	1
64	32	1
128	64	2
256	128	5
512	256	7
1024	512	6
2048	1024	51
4096	2048	99
8192	4096	324

DP Greedy



Resultados greedy:		
Tesoros	**Capacidad**	**Tiempo(ms)**
16	8	1
32	16	1
64	32	1
128	64	1
256	128	1
512	256	1
1024	512	2
2048	1024	2
4096	2048	4
8192	4096	46

Estudio Experimental

Hemos implementado test que verifican los resultados basándose en los datasets que se nos proporcionan obteniendo resultados correctos.