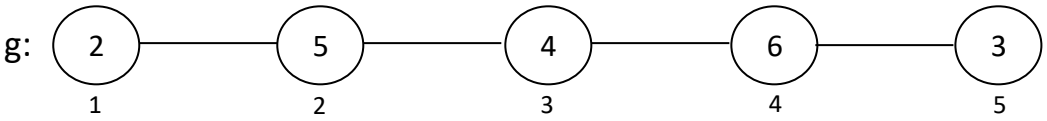
Ficha 15

Programación Dinámica [Aplicaciones]

1.) El problema del Subconjunto Independiente de Suma Máxima[1].

Un *Grafo Lineal* (designado como *Path Graph* en la terminología básica en inglés), se entiende como un grafo *g* no ponderado con *n* vértices numerados entre *1* y *n*, pero tal que sólo existen los arcos *(g[i], g[i+1])* para todo vértice *i* en *[1..n]* y además, cada vértice tiene un valor o peso representado como un número entero que puede ser negativo, cero o positivo. A los efectos del problema que analizaremos en esta primera sección, no importa si *g* es dirigido o no dirigido.

En otras palabras, un *grafo lineal* es aquel en el que cada vértice *v[i]* tiene como único antecesor a *v[i-1]* y como único sucesor a *v[i+1]*, y el valor contenido en *v[i]* es el peso del vértice *i*. El siguiente es un ejemplo de un grafo lineal *g* con *n = 5* vértices:



Un conocido problema referido al ámbito de los grafos lineales, se conoce como el problema del *Máximo Subconjunto Independiente (MSI)* de un grafo lineal *g*. Dado el grafo lineal *g*, se trata de encontrar el subconjunto *s* de *g* cuyos vértices no sean adyacentes (no estén unidos por un arco) y tenga la *mayor suma de pesos*. Por ejemplo, para el grafo de la figura anterior, el *MSI* contiene a los vértices *g[2] = 5* y *g[4] = 6*, ya que la suma *11* de los pesos de esos vértices es la máxima posible tomando cualquier combinación de vértices que no estén unidos por un arco en forma directa.

Una solución posible para el problema del *MSI*, es aplicar un algoritmo de fuerza bruta y probar toda posible combinación de los n vértices contra todos aquellos que no estén directamente conectados, e ir tomando el máximo valor que se vaya calculando, pero el tiempo de ejecución será de *orden exponencial* (la cantidad posible de combinaciones de los *n* vértices para los distintos subconjuntos independientes es $O(2^n)$), lo cual es inaceptable incluso para *n* tan pequeño como 30, 40 o 50.

Una solución mucho mejor puede plantearse con programación dinámica, usando una tabla para almacenar los valores de las sumas máximas para subconjuntos de menor tamaño del grafo de partida.

Sea un grafo lineal *g* con n vértices, en el que cada vértice *i* está representado por su peso *g[i]*. Se trata de encontrar un subconjunto *s* de g tal que *s* sea un *MSI* para *g*. Sea *g[n]* el peso o valor del último vértice *n*

[1] La fuente esencial en la que está basada la explicación de los problemas tratados en toda esta ficha (problema del *Máximo Subconjunto Independiente* y problema de la *Mochila*), es el material del curso "*Design and Analysis of Algorithms II*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

de *g*. Si se estudia el problema con detenimiento, surge que pueden aparecer dos casos en relación a *g[n]*:

a.) Que *g[n] no pertenezca* a *s*, siendo *s* un *MSI* para *g* (esto es: suponemos que g[n] $\notin$ s).

Supongamos que *g1 = g – {g[n]}* (o sea, supongamos que *g1* es el mismo grafo *g* pero sin el último vértice). Entonces, podemos probar que *s* sigue siendo un *MSI* para *g1* puesto que ya lo era para *g*.

Demostración: *s* debe ser un *MSI* para *g1*, pues si no lo fuese, entonces habría *otro* conjunto *s1* mejor que *s* para *g1*, el cual también sería mejor que *s* en *g*, lo cual implica una contradicción (pues partimos de la suposición que *s* era un *MSI* para *g*).

b.) Que *g[n] pertenezca* a *s*, siendo *s* un *MSI* para *g* (esto es: suponemos que g[n] $\in$ s).

Si *g[n]* pertenece a *s*, entonces *g[n-1]* ***no*** pertenece a *s* (por definición de subconjunto independiente). Supongamos ahora que *g2 = g – {g[n], g[n-1]}* (o sea, supongamos que *g2* es el mismo grafo *g* pero sin los dos últimos vértices). Entonces, podemos probar que el conjunto *s – {g[n]}* es un *MSI* para *g2*.

Demostración: si *s – {g[n]}* no fuese un *MSI* para *g2*, entonces habría otro conjunto *s2* tal que *s2* es mejor que *s – {g[n]}* para *g2.* Pero esto también implicaría que *s2* $\cup$ *{g[n]}* es mejor que *s* en *g*, lo cual es una contradicción ya que supusimos que s era un *MSI* para *g*.

Por lo tanto, un *MSI* para *g* necesariamente ***debe*** ser uno (y sólo uno) de los dos siguientes:

i.) Un *MSI* para *g1*.

ii.) El vértice *g[n]* más un *MSI* para *g2*.

Llegados a este punto, se podría hacer un planteo recursivo que calcule el mejor **i.)** y el mejor **ii.)** y retorne el máximo valor entre ellos (ya que no sabemos si *g[n]* pertenecerá o no al *MSI*). Pero este planteo tendría tiempo exponencial, pues debería analizar ***todas*** las alternativas hacia atrás.

Sin embargo, podemos ver que muchos cálculos para muchos subgrafos (o subproblemas) aparecerían más de una vez y no sería necesario volver a realizarlos. Se puede guardar el valor del MSI del grafo g_i en una tabla, y luego consultar la tabla si el subproblema para g_i vuelve a aparecer. De esta forma, como el grafo *g* de partida tiene *n* vértices, habrá *n* subproblemas (o subgrafos) y el tiempo total de ejecución será *O(n)*. La recurrencia básica a emplear, sale de tomar el máximo entre los dos casos **i** y **ii**.

La tabla *mw* para almacenar los resultados de los subgrafos anteriores tendrá entonces *n+1* casilleros, para incluir el caso *mw[0]* que representa al subconjunto vacío (cuyo máximo *MSI* obviamente vale 0) y para incluir la casilla *mw[n]* donde quedará almacenado el valor final del *MSI* para el grafo *g* completo.

Finalmente, el planteo del algoritmo definitivo sería entonces:

Entradas:

- Un arreglo (o una lista) *g* de *n* componentes de tipo *int*, que representa al grafo lineal.

Salidas:

- ✓ Un número entero que indique el valor de la suma para un *MSI* de *g*.

Proceso (algoritmo básico – recurrencia a emplear):

int maxIndependentSet(int g[]):

- Sea *n* = *g*.length;
- Sea la tabla *mw[]* con *n* + 1 casilleros: mw = new int[n + 1]
- Sea *mw[0]* = 0 y mw[k] = g[k-1] con k = 1, 2, … n

- Para i en [2 .. n]:
  - Sea v1 = mw[i - 1] // primer caso
  - Sea v2 = mw[i - 2] + mw[i]; // segundo caso
  - Sea mw[i] = max(v1, v2)

- Retorne *mw[n]*

El modelo *DLC-Path-Graph* que acompaña a esta ficha implementa el algoritmo anterior en forma directa, como método de la clase *PathGraph* que mostramos a continuación. Se deja el análisis de los detalles de código fuente para el alumno:

```
import java.util.ArrayList;
public class PathGraph
{
        // el grafo lineal...
        private ArrayList<Integer> g = new ArrayList<>();

        public PathGraph()
        {
        }

        public int getVertex(int i)
        {
           return g.get(i);
        }

        public void setVertex(int i, int w)
        {
           g.add(i, w);
        }

        public void addArray(int v[])
        {
           for(int i = 0; i < v.length; i++)
           {
              g.add(i, v[i]);
           }
        }

        public int maxIndependentSet()
        {
            // la cantidad de vértices...
            int n = g.size();
```

```
            // la tabla con los pesos máximos de los subconjuntos anteriores...
            int mw[] = new int[n + 1];

            // iniciar la tabla con los pesos de cada vértice (con mw[0] = 0...)
            for(int i = 1; i <= n; i++)
            {
               mw[i] = g.get(i-1);
            }

            // lanzar la recurrencia básica (llenado de la tabla)...
            for(int i = 2; i <= n; i++)
            {
               int v1 = mw[i-1];
               int v2 = mw[i-2] + mw[i];
               mw[i] = (v1 > v2)? v1 : v2;
            }

            // retornar el peso máximo...
            return mw[n];
        }
}
```

2.) El problema de la Mochila.

Un famoso problema muy estudiado en el ámbito de las ciencias de la computación es el *Problema de la Mochila*. Si bien suele enunciarse de manera informal casi como un juego de ingenio (cómo maximizar la carga que puede llevarse en una mochila), en realidad se trata de un problema de *optimización* que aparece frecuentemente en contextos de asignación de recursos (por ejemplo, organización de cursos y conferencias de forma de maximizar el uso de las salas disponibles, distribución de personas entre diversas categorías de tareas, aprovechamiento óptimo del espacio disponible en un depósito, etc.)

Básicamente, el problema consiste en lo siguiente: se dispone de una mochila (o un depósito, o un auditorio dividido en salas, etc.) que tiene cierta capacidad *W* de carga. A modo de ejemplo, si el problema se tipifica con una mochila, el valor *W* puede ser el peso máximo que esa mochila puede soportar, y si se trata de un depósito, el valor *W* puede ser el volumen de ese depósito. En lo que sigue de la explicación, simplemente supondremos la versión sencilla de la mochila.

Además, se dispone de un conjunto *C* de *n* items u objetos d_i tales que cada uno de esos ellos tiene dos atributos: un valor v_i no negativo y un peso w_i que se supone como número entero. El valor v_i puede representar el precio o valor monetario de cada item, o cualquier otro número no negativo que de alguna forma indique una apreciación de tal item. El peso w_i puede representar efectivamente el peso (en kilogramos o en cualquier otra unidad) o cualquier otro número entero que proporcione una idea de tamaño, peso, superficie, etc, para el item. El significado de estos dos atributos dependerá del contexto del problema.

El objetivo es encontrar un subconjunto *S* de *C* (o sea, $S \subseteq C$) tal que la suma de valores v_i de los items en *S* sea la máxima posible, pero sujeto a la restricción de que la suma de pesos w_i de los items en *S* no supere la capacidad máxima *W*. En otras palabras: encontrar la forma de llenar la mochila tanto como se pueda, *sin pasarse del peso límite*, de forma de *maximizar el valor de la carga total*. Formalmente:

Entradas:

Un conjunto de *n* items *C* = {d1, d2, d3, … dn} = {(v1, w1), (v2, w2), (v3, w3), … (vn, wn)}
W = capacidad máxima de la mochila

Salidas:

Un subconjunto S $\subseteq$ C [con S = {d1, d2, d3, …, dk} tal que 1 <= k <= n]
pero tal que:

En *S* sea máximo el valor de $\sum \mathbf{v_i}$
En *S* sea $\sum \mathbf{w_i}$ **<= W**

No se conocen algoritmos eficientes para resolver el problema de la mochila. Veremos más adelante, que esto significa que no se ha podido hasta la fecha plantear un algoritmo que pueda resolver el problema con un *tiempo de ejecución polinómico* respecto del tamaño de la entrada. En nuestro caso el tamaño de la entrada es *n* (la cantidad de items disponibles) y *W* (la capacidad de la mochila). Todos los algoritmos que se conocen tienen tiempo de ejecución super polinomial (es decir, el tiempo de ejecución es mayor al que se esperaría para cualquier función polinómica en *n* y *W*).

Una solución muy conocida se basa en *programación dinámica*, y logra un tiempo de ejecución *pseudo-polinómico O(nW)*. En fichas posteriores haremos un análisis más profundo de lo que significa el tiempo de ejecución polinómico o el tiempo pseudo-polinómico (entre otros aspectos de la complejidad computacional).

Para el desarrollo del algoritmo basado en programación dinámica debemos identificar los *subproblemas* que componen la *estructura de recurrencias óptima* final, en forma similar a como se hizo con el problema de Subconjunto Independiente de Suma Máxima en un Grafo Lineal. Para ello, supongamos que *S* es una solución óptima para una instancia de nuestro problema (es decir, supongamos que *S* es un subconjunto de items cuya suma de valores es máxima, sin pasar el límite *W* en la suma de pesos). Tenemos entonces dos casos posibles:

a.) Que el item *n* (el último item) *no pertenezca* a *S*: n $\notin$ S.

En este caso, es fácil probar que entonces *S* sigue siendo una solución óptima para los primeros *n-1* items (con la misma capacidad máxima *W*).

Demostración: Si hubiese otra solución *S1* mejor que *S* para los primeros *n-1* items, entonces esa solución *S1* seguiría siendo óptima para los *n* items, lo cual es una contradicción (supusimos que *S* era la óptima aún sin incluir a *n*).

b.) Que el item *n* *pertenezca* a *S*: n $\in$ S.

En este otro caso, podemos probar también en forma simple que *S – {n}* es una solución óptima para los primeros *n-1* items, pero considerando ahora una capacidad igual a $W - w_n$.

Demostración: Supongamos que es *S2* una solución mejor que *S – {n}* para los primeros *n-1* items, tomando una capacidad $W - w_n$. Designemos como *p* al *peso total* de una solución. Como hemos supuesto inicialmente que *S* es óptima para los *n* items, entonces:

p(S) <= W [por definición de solución óptima]

$\Rightarrow$ p(S – {n}) <= W – w_n [ya que supusimos que n $\in$ S]

$\Rightarrow$ p(S2 $\cup$ {n}) <= W [con lo que *S2* $\cup$*{n}* sería mejor que *S* para los *n* ítems: contradicción]

En base a los dos casos que acabamos de detectar, la idea es usar una tabla bidimensional *a* en la cual cada casillero *a[i][x]* se use para almacenar el *valor acumulado* (suma de valores v_i) de la *mejor solución* en la cual:

- se usen sólo los primeros *i* items
- se tenga un peso total <= *x*

Con esto, la recurrencia básica para el armado de la tabla propuesta quedará como sigue:

Para todo i = [1, 2, 3, … n] y para cualquier peso x:

a[i][x] = máximo
- a[i-1][x] [Caso a: excluyendo al item *i*]
- v_i + a[i-1][x-w_i] [Caso b: incluyendo al item *i*]

pero considerando que si $w_i > x$ entonces *a[i][x] = a[i-1][x]*

Con todo esto, el algoritmo definitivo sería entonces (en pseudocódigo):

Supuestos previos:
- Sea la clase *Item* con dos atributos *vi* y *wi* tal como se expuso en estas notas.

Entradas:
- Un arreglo (o una lista) *d* de *n* objetos de la clase *Item*.
- El valor *W* que indica la capacidad máxima de la mochila.

Salidas:
- ✓ Un *número entero* que indique la suma de los valores *vi* de una *solución óptima* para el arreglo *d* para una mochila con capacidad *W*.

Proceso (algoritmo básico – recurrencia a emplear):

```
int knapsack(Item d[], int w):
    – Sea n = d.length la cantidad de items
    – Sea a[n+1][w+1] la matriz de recurrencias (inicialmente en cero)
    – Para i en [1..n]:
        ▪ Sea wi = el peso del objeto i [que en el vector d está en (i-1)]
        ▪ Sea vi = el valor del objeto i
        ▪ Para x en [1..w] (o sea, por cada posible capacidad x <= w):
            ○ Sea v1 = a[i-1][x];
            ○ Si(wi >x):
                  v2 = v1
              sino
                  v2 = a[i-1][x-wi] + vi
        ▪ a[i][x] = máximo{ v1, v2 }
    – Retornar a[n][x]
```

El modelo *DLC-Knapsack* que acompaña a esta ficha implementa el algoritmo anterior en forma directa, en el constructor de la clase *Knapsack* que mostramos a continuación. Se deja el análisis de los detalles de código fuente para el alumno:

```
public class Knapsack
{
   // la tabla de recurrencias...
   private int a[][] = null;

   // la capacidad máxima de la mochila usada...
   private int w;

   public Knapsack(Item d[], int w)
   {
      // cantidad de items...
      int n = d.length;

      // capacidad de la mochila...
      this.w = w;

      // la matriz de recurrencias, todos sus casilleros inicialmente en cero...
      // tamaño [n+1][w+1] para incluir a n y a w como indices válidos...
      // y mantener el cero.
      a = new int[n+1][w+1];

      // llenado de la matriz...
      for(int i = 1; i <= n; i++)
      {
         // peso del objeto numero i (que en el vector d está en (i-1)...
         int wi = d[i-1].getWeight();

         // valor de ese objeto...
         int vi = d[i-1].getValue();

         // para cada posible capacidad residual...
         for(int x = 0; x <= w; x++)
         {
            int v1 = a[i-1][x];
            int v2 = (wi > x) ? v1 : a[i-1][x - wi] + vi;
            a[i][x] = (v1 > v2)? v1 : v2;
         }
      }
   }

   public int[][] getMatrix()
   {
      // retornar la tabla completa... solución en a[n][w]...
      return a;
   }
```

```
        public int getOptimusValue()
        {
           return a[a.length - 1][w];
        }

        @Override
        public String toString()
        {
           StringBuilder r = new StringBuilder("[\n");
           for(int f = 0; f < a.length; f++)
           {
              r.append("\t[ ");
              for(int c=0; c < a[f].length; c++)
              {
                 r.append(a[f][c]).append(" ");
              }
              r.append("]\n");
           }
           r.append("]");
           return r.toString();
        }
}
```