Actividad 8. Esquemas Algorítmicos

Objetivo

Resolver problemas que siguen diferentes estrategias, voraz, divide y vencerás, vuelta atrás.

Procedimiento

1. Leer y comprender los apuntes de esquemas algorítmicos que están disponibles en **Tema**, sección **Documentos e Ligazóns** / **Teoría** / *EsquemasAlgoritmicos.pdf*
2. Emplear técnicas de aprendizaje colaborativo para resolver los *ejercicios de clase* que se indican en esta actividad, utilizando el lenguaje java. En concreto se utilizará la **técnica del rompecabezas.**
    a. Se formarán grupos de 4 personas.
    b. Cada miembro del grupo será responsable de la resolución de un ejercicio (trabajo individual no presencial).
    c. En clase se reunirán los expertos de cada ejercicio para puesta en común.
    d. Después cada persona volverá a su grupo base para explicarles a cada compañero lo que aprendieron.
3. Los *otros ejercicios propuestos* se trabajarán de manera presencial en horas de grupo grande y pequeño. Para probar su correcto funcionamiento se puede hacer uso del test disponible en Tema, sección Documentos e Ligazóns / Actividades / Test / **Actividad8Test.java**
4. Para resolver todos los ejercicios es necesario conocer la interfaz del TAD Map y TAD Grafo, disponible en el anexo.

Evaluación

El trabajo colaborativo será evaluado de manera grupal el día 10 de diciembre.

Estos contenidos serán evaluados mediante una prueba teórica que tendrá lugar el 17 de diciembre de 2019.

Tiempo estimado

8 horas.

Ejercicios para resolver en clase

1. Supongamos que sólo están disponibles los siguientes billetes y monedas: 100, 50, 20, 10, 5 y 1 euro (con un número limitado de cada tipo de moneda). Nuestro problema consiste en diseñar un algoritmo para pagar una cierta cantidad a un cliente, utilizando el *menor* número posible de billetes y monedas. Por ejemplo, si tenemos que pagar 289 euros y tenemos 5 billetes y monedas de cada tipo, la mejor solución consiste en dar al cliente los siguientes

billetes/monedas: 2 de 100, 1 de 50, 1 de 20, 1 de 10, 1 de 5 y 4 de 1 euro. Implementa este algoritmo siguiendo una estrategia **voraz**.

```
public static Map<Integer,Integer> darCambio(int importeDevolver, Map<Integer,Integer> mapCanti)
```

2. Problema de dar el cambio. Supongamos que el cajero sólo tiene billetes de 200 y 500 euros y nos debe dar 2100 euros. Con una estrategia voraz nunca llegaría a la solución correcta (daría 4 de 500 y no podría dar el resto), mientras que con vuelta atrás podemos ir dando billetes y si llegamos a una combinación sin solución, volvemos atrás intentándolo con otro billete y así sucesivamente. En este caso la solución sería 3 billetes de 500 y 3 de 200. Implementa este algoritmo siguiendo una estrategia de **vuelta atrás**.

```
public static boolean darCambio(int importeDevolver, Map<Integer,Integer> mapCanti, Map<Integer,Integer> mapSol)
```

3. Considere que tenemos n programas distintos para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^{n} m_i$

   Diseñe un algoritmo, utilizando un esquema **voraz**, que calcule una colección de estos programas para grabar en el disco, de manera que se utilice la *máxima* capacidad posible del disco en la grabación de estos programas.

   El método principal tendrá esta cabecera:

   ```
   public static Vector<String> llenarCDVoraz (int capacidadMaxima, Map<String,Integer> espacioProgramas)
   ```

   donde:

   capacidadMaxima: es la capacidad del disco
   espacioProgramas: es un Map que contiene el nombre de cada programa que queremos almacenar (identificado por un String) y el espacio que ocupa (Integer)

   El método retornará un Vector que contiene el nombre de cada programa que se eligió para almacenar en el disco.

4. Considere que tenemos n programas distintos para grabar en un disco, pero el espacio de memoria que necesitan excede la capacidad del disco. Cada programa P_i requiere m_i kilobytes de memoria, la capacidad del disco es de C kilobytes y $C < \sum_{i=1}^{n} m_i$

   Resuelve este algoritmo siguiendo un esquema de **vuelta atrás**, de tal forma que si existe solución consigue ocupar el espacio del CD completamente, sino devuelve el CD vacio.

   El método principal tendrá esta cabecera:

   ```
   public static boolean llenarCDVueltaAtras (int capacidadMaxima, Map<String,Integer> espacioProgramas, Vector<String> CD)
   ```

   donde:

   capacidadMaxima: es la capacidad del disco

espacioProgramas: es un Map que contiene el nombre de cada programa que queremos almacenar(identificado por un String) y el espacio que ocupa (Integer).
CD: es un Vector que contiene el nombre de los programas grabados.

Otros ejercicios propuestos

5. Algoritmo del **Viajante**. Consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. "Un viajante debe recorrer una serie de ciudades interconectadas entre sí, de manera que recorra todas ellas con el menor número de kilómetros posible". Emplea una estrategia **voraz** para resolver el ejercicio.

   ```
   public static <E> Grafo<E,Integer> viajante(Grafo<E,Integer> g, Vertice<E> v)
   ```

6. Algoritmo de **Prim**. Es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, **no** dirigido y cuyas aristas están etiquetadas. En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo. Consideremos un mapa de carreteras, con dos tipos de componentes: las ciudades (*nodos*) y las carreteras que las unen. Cada tramo de carreteras (*arco*) está señalado con su longitud. Se desea implantar un tendido eléctrico siguiendo los trazos de las carreteras de manera que conecte todas las ciudades y que la longitud total sea mínima. Emplea una estrategia **voraz** para resolver el ejercicio.

   ```
   public static <E> Grafo<E,Integer> prim(Grafo<E,Integer> g, Vertice<E> v)
   ```

7. Escribe el **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un algoritmo para la determinación del camino más corto dado un vértice origen al resto de vértices en un grafo dirigido y con pesos en cada arista.

   La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a los demás vértices; cuando se obtiene el camino más corto desde el vértice origen, al resto de vértices que componen el grafo, el algoritmo se detiene. Emplea una estrategia **voraz** para resolver el ejercicio.

   public static <E> **Map<Vertice<E>,Integer>** dijkstra(Grafo<E,Integer> g, Vertice<E> v)

8. Problema de la mochila. Se desea llenar una mochila hasta un volumen máximo V, y para ello se dispone de n objetos, en cantidades limitadas c_1, ..., c_n y cuyos valores por

unidad de volumen son v_1, ..., v_n, respectivamente. Debe seleccionarse de cada objeto una cantidad máxima k_i con tal de que $k_i * v_i <=$ Volumen que resta para llenar la mochila. El problema consiste en determinar las cantidades k_1, ..., k_n que llenan la mochila *maximizando* el valor total $\Sigma\ v_i * c_i$, i ϵ {1, …, n}. Emplea una estrategia **voraz** para resolver el ejercicio.

```
public static Map<String,Integer> llenarMochila(int volumenMaximo, Map<String,Integer> cantidades, Map<String,Integer> volumenes)
```

9. Modifica el algoritmo anterior para que cumpla la siguiente restricción: Se desea llenar una mochila hasta un volumen máximo *V*, y con un peso máximo *P*. Para ello se dispone de *n* objetos, en cantidades limitadas c_1, ..., c_n y cuyos valores por unidad de volumen son v_1, ..., v_n, respectivamente, y los valores por unidad de peso son p_1, ..., p_n, respectivamente. Debe seleccionarse de cada objeto una cantidad máxima k_i con tal de que $k_i * v_i <=$ *Volumen que resta para llenar la mochila*, además $k_i * p_i <=$ *Peso que resta para alcanzar lo máximo que soporta la mochila*.

   El problema consiste en determinar las cantidades k_1, ..., k_n que llenan la mochila *maximizando* el valor total $\Sigma\ v_i * k_i$, $i \epsilon \{1, …, n\}$, pero sin pasarse del peso soportado por la mochila. Emplea una estrategia **voraz** para resolver el ejercicio.

   ```
   public static Map<String,Integer> llenarMochila(int volumenMaximo, int pesoMaximo, Map<String,Integer> cantidades, Map<String,Integer> volumenes, Map<String,Integer> pesos)
   ```

10. Problema de las 8 reinas. El problema consiste en colocar ocho reinas en un tablero de ajedrez sin que se den jaque (dos reinas se dan jaque si comparten fila, columna o diagonal). Puesto que no puede haber más de una reina por fila, podemos replantear el problema como "colocar una reina en cada fila del tablero de forma que no se den jaque". En este caso, para ver si dos reinas se dan jaque basta con ver si comparten columna o diagonal (emplea el método *buenSitio*). Por lo tanto, toda solución del problema se puede representar como una 8-tupla (X_0,...,X_7) en la que X_i es la columna en la que se coloca la reina que está en la fila *i* del tablero. Implementa el algoritmo *colocarReinas*, siguiendo una estrategia de **vuelta atrás** para resolver el ejercicio.

```
private static boolean buenSitio (int r, int[] tabl)
    {
    // ¿Es amenaza colocar la reina “r” en A[r], con las
anteriores?
     int i;
     for(i = 0; i < r; ++i)
     {
          if (tabl[i] == tabl[r]) return false;
          if (Math.abs(i-r) == Math.abs(tabl[i]-tabl[r])) return
false;
     }
     return true;
    }

 public static boolean colocarReinas (int reina, int[] tablero)
```

11. Problema de atravesar un laberinto. Nos encontramos en una entrada de un laberinto y debemos intentar atravesarlo. Dentro del algoritmo nos encontraremos con muros que no podremos atravesar, sino que habrá que rodear, lo que hará que nos encontremos a veces en un callejón sin salida. Es necesario tener en cuenta las siguientes condiciones:

    - Representación: Array N x N, de casillas marcadas como libre (**indicadas por el carácter ' '**) u ocupada por una pared (**indicadas por el carácter 'X'**).
    - Es posible pasar de una casilla a otra moviéndose solamente en vertical u horizontal.
    - El problema se soluciona cuando desde la casilla (0,0) se llega a las casilla (n-1, n-1).
    - Para resolver este problema se diseñará un algoritmo de búsqueda con retroceso de forma que se marcará en la misma matriz del laberinto un camino solución si existe (**el camino se marcará con el carácter 'C'**).
    - Si por el camino recorrido se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino (**la casilla imposible se marcará con el carácter 'I'**).
    - Por tanto, hay que marcar las casillas por donde ya se ha pasado (**con el carácter 'C' o 'I'**), de esta forma se evita meterse varias veces por el mismo callejón sin salida, dar vueltas alrededor de columnas....

    Emplea una estrategia de **vuelta atrás** para resolver el ejercicio.

    ```
    public static boolean ensayar(char [][] laberinto, int posicionX, int posicionY)
    ```

12. Considérese un "vector" que almacena n enteros positivos distintos: vector = {v_1, v_2, v_3, …, v_n}. El problema de la suma consiste en encontrar un subconjunto del vector cuya suma sea exactamente "resultado".

    La "solución" tendrá la forma de vector con valores 1 y 0, es decir, solución = (s1, s_2, s_3, …, s_n} con $s_i \in \{0,1\}$ y deberá cumplir: $\sum v_i * s_i$ = resultado.

    Para resolver este problema deberá utilizarse la estrategia de **vuelta atrás**. Que irá considerando todas las posibilidades (incluir / no incluir cada elemento en la solución).

    Una posible cabecera para el método que resuelve el problema sería:

    ```
    public static boolean subconjunto(int[] vector, int[] solucion, int resultado, int indice)
    ```

    siendo:

    int[] vector: el vector con n enteros positivos distintos

    int[] solución: el vector de 0 y 1 con la solución al problema

    int resultado: lo que tiene que sumar los enteros seleccionados del vector.

    int índice: posición del vector a considerar incluir o no incluir en la solución. Puede considerarse que vale 0 la primera vez que se invoca al método.

13. Un problema que sigue la estrategia **divide y vencerás** es encontrar el *k-ésimo* menor elemento de un array. Obviamente, se puede realizar ordenando el array, pero es de esperar que selección sea un proceso más rápido, al solicitarse menor información. Haciendo un pequeño cambio en el algoritmo *quicksort (revisar apuntes de AEDI)*, se puede resolver el problema de selección en *tiempo lineal*, en promedio. Llamamos a este algoritmo selección rápida. Los pasos a realizar son los siguientes:

    Siendo "array", la estructura donde se busca el k menor elemento.

    - Si el número de elementos en "array" es 1, entonces presumiblemente *k* también es 1, por lo que se puede devolver el único elemento en "array".
    - En otro caso, elegir un elemento *v* de "array" como pivote:
    - Hacer una partición de "array" – {*v*} en "arrayIzq" y "arrayDer", exactamente igual a como se hacía en el *quicksort*.
    - Si *k* es menor o igual que el número de elementos en "arrayIzq", entonces el elemento que estamos buscando debe estar en "arrayIzq", por lo que se llama recursivamente a *seleccionRapida*(arrayIzq, *k*). Si *k* es exactamente igual a uno más que el número de elementos de "arrayIzq", entonces el pivote es el *k-ésimo* menor elemento y se puede devolver como respuesta. En otro caso, el *k-ésimo* menor elemento estará en "arrayDer". De nuevo podemos hacer una llamada recursiva y devolver el resultado obtenido.

    Utilizando los siguientes métodos privados, implementa el algoritmo de selección rápida.

    ```
    public static int SeleccionRapida (int []array, int k_menor, int i, int j)
    ```

```
// Selecciona un elemento pivote: entre dos elementos diferentes devuelve el elemento mayor.

private static int buscaPivote (int [] aux, int inicio, int fin)
{
int primer = aux[inicio];
int k = inicio + 1;

while (k <= fin)
{
   if (aux[k] > primer) {
              return k;
   }
   else if (aux[k] < primer) {
                  return inicio;
   }
   else {
        k++;
   }
}
return -1;
}
```

```
//Divide el array en menores que pivote, y mayores e igual a pivote

// Devuelve la posición del primer elemento de los mayores o iguales a pivote (es decir el número de elementos que hay en la partición de la izquierda).

private static int particion (int [] aux, int inicio, int fin, int pivote)
{
  int derecha = inicio;
  int izquierda = fin-1;
  while (derecha <= izquierda)
  {
   while (aux[derecha] < pivote) {
                  derecha++;
   }
   while (aux[izquierda] >= pivote) {
                  izquierda--;
    }

   if (derecha < izquierda)
      intercambiar (aux, derecha, izquierda);
    }

    return derecha;
  }
```

```
private static void intercambiar (int
[] aux, int i, int j){
    int temp = aux[i];
    aux[i] = aux[j];
    aux[j]=temp;
}
```

Anexo:

```
public interface Grafo<E,T>{
        public boolean esVacio();
        public boolean estaVertice(Vertice<E> v);
        public boolean estaArco(Arco<E,T> a);
        public Iterator<Vertice<E>> vertices();
        public Iterator<Arco<E,T>> arcos();
        public Iterator<Vertice<E>> adyacentes (Vertice<E> v);
        public void insertarVertice (Vertice<E> v);
        public void insertarArco (Arco<E,T> a);
        public void eliminarVertice (Vertice<E> v);
        public void eliminarArco (Arco<E,T> a);
}

public class Vertice<E> {
        private E etiqueta;
        public Vertice(E o)
        public E getEtiqueta()
}

public class Arco <E,T>{
        private Vertice<E> origen, destino;
        private T coste;
        public Arco(Vertice<E> o, Vertice<E> d, T c)
        public Vertice<E>E getOrigen()
        public Vertice<E> getDestino()
        public T getEtiqueta()
}
```

```
public interface Map<K,V> {
          public int tamaño();
          public V get (K clave);
          public void insertar(K clave, V valor);
          public V eliminar(K clave);
          public Iterator<K> getClaves();
          public Iterator<V> getValores();
}

public interface Iterable<E>{
        public Iterator<E> iterator();
}

public interface Iterator<E>{
        public boolean hasNext();
        public E next() throws NoSuchElementException
}
```