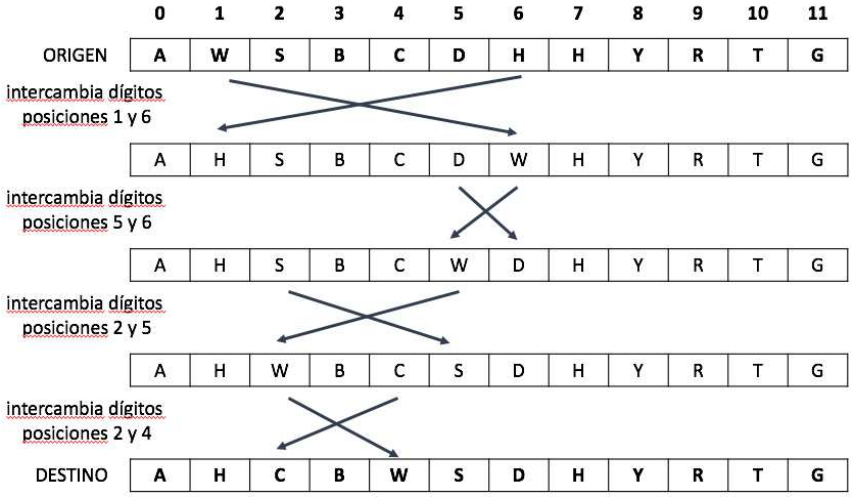
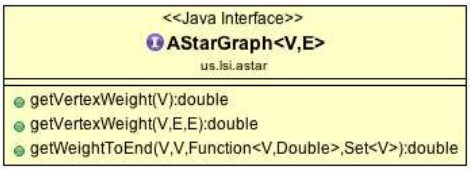
Ejercicio 3: Grafos Virtuales

Con el objetivo de encontrar una secuencia de ADN válida para un organismo, se propone utilizar un algoritmo que haciendo el menor número de intercambios de símbolos posibles encuentre la combinación correcta.
Para ello, considere:

1) Se sabe que la secuencia de ADN de un organismo está compuesta de 12 símbolos, siendo cada símbolo una de las 15 posibles letras según las reglas de la UIPAC ("A", "C", "G", "T", "R", "Y", "K", "M", "S", "W", "B", "D", "H", "V", "N").
2) Se parte de una secuencia origen desordenada con el mismo número de símbolos que la secuencia destino.
3) El objetivo es realizar el menor número de intercambios posibles para llegar a la secuencia destino, es decir, secuencia con el mismo número de símbolos que la original pero ordenada.
4) En cada paso se intercambian dos símbolos de la secuencia de ADN.
5) El espacio de búsqueda (es decir, el número de combinaciones posibles) es suficientemente grande como para rechazar la idea de representar el problema como un grafo tradicional.

Por ejemplo, se tiene la secuencia "AWSBCDHHYRTG", para que sea correcta, esa misma combinación de letras debería estar ordenada de la forma: "AHCBWSDHYRTG". Si aplicamos el algoritmo A* obtendríamos





Se pide:

1. Resuelva este problema como un problema de caminos mínimos usando grafos virtuales. Indique de forma justificada de qué tipo serían sus vértices y aristas, y qué propiedades tendrían. Indique justificadamente cuántos vecinos puede tener un vértice para este problema.
2. Dentro de la clase que corresponda a sus vértices (que debe implementar *VirtualVertex*), implemente el método *getNeighborListOf*() que debe devolver un *Set* con los vértices vecinos del objeto vértice actual.
3. Dentro de la clase que corresponda a su grafo, implemente los métodos que corresponden con la interfaz *AStarGraph*.

Nota: Si necesita utilizar algún método de la clase de Vértices o Aristas para resolver cualquier apartado, debe implementarlo.

Solución:

1. **Respuesta:**
   - Grafo: Grafo Virtual no dirigido.
   -Vértices: Secuencia implements VirtualVertex. Propiedad secuencia que es una lista de String.
   -Arista: SimpleEdge<Secuencia>
   -Vecindad: Un vértice tiene n(n-1)/2 vecinos (siendo n el número de símbolos) que se corresponde con todos los cruces posibles de un símbolo.
2. **Respuesta:**

```
public Set<Secuencia> getNeighborListOf() {
      return IntStream.range(0, this.sec.size()).boxed()
      .<ParInteger>flatMap(x->IntStream.range(x+1,this.sec.size())
      .boxed()
      .map(y->ParInteger.create(x,y)))
      .<Secuencia>map(p->getVecino(p.p1, p.p2))
      .collect(Collectors.toSet());

}

public Secuencia getVecino(int i, int j){
      Secuencia nuevasecuencia = new Secuencia(this);
      List<String> secnueva = nuevasecuencia.getSecuencia();
      String s1 = secnueva.get(i);
      String s2 = secnueva.get(j);
      secnueva.set(i, s2);
      secnueva.set(j, s1);
      return nuevasecuencia;
}
```

3. **Respuesta:**

```
   public double getVertexWeight(Secuencia vertex) {
         return 0;
   }
   public double getVertexWeight(Secuencia vertex,
   SimpleEdge<Secuencia> edgeIn, SimpleEdge<Secuencia> edgeOut) {
      return 0;

   }
   public double getWeightToEnd(Secuencia actual, Secuencia
   endVertex, Function<Secuencia, Double> goalDistance,
   Set<Secuencia> goalSet) {
         return actual.getDistancia(endVertex);
   }
   public double getDistancia(Secuencia end){
      int minsize = ath.min(end.getSecuencia().size(),this.size());
      Double d = Double.valueOf(Math.abs(end.getSecuencia().size()
                  - this.size()));
      for(int i=0;i<minsize;i++){
         if(!end.getSecuencia().get(i).equals(this.get(i)))
            d++;
      }
      return d;
   }
```