UNIVERSIDAD DE MÁLAGA	Análisis y Diseño de Algoritmos Parcial 2 Diciembre 2020
Alumno:	Grupo:

1.-(7.5 ptos.) Pertenecemos a un grupo de una red social que se dedica a realizar sorteos de lotes de objetos de segunda mano todas las semanas. Para participar hay que pagar 2€ y elegir un número que no haya sido elegido aún. En caso de ser el ganador nos llevaremos el lote completo, que después venderemos para obtener beneficio. El grupo proporciona la información de los lotes que se van a sortear durante las siguientes *n* semanas y conocemos el valor *v(i)* del lote que se sortea en la semana *i*. El sorteo se realiza con una aplicación de generación de números aleatorios y nos han puesto en contacto con un hacker que nos puede proporcionar el siguiente valor aleatorio que proporcionará dicha aplicación. La tarifa del hacker *t(i)* no es fija y depende de la semana *i* en que le pidamos el número. Para no levantar sospechas, decidimos que no podremos ganar más de una vez cada $K > 0$ semanas (en otras palabras, deben transcurrir al menos $K - 1$ semanas entre dos premios consecutivos). Determinar qué semanas hay que participar en los sorteos para maximizar el beneficio total (valor de los lotes menos los gastos ocasionados).

a)(1) Enfocar el problema como una secuencia de decisiones y demostrar que se satisface el principio de subestructura óptima.

Vamos a recorrer de izquierda a derecha la lista de lotes (desde el correspondiente al sorteo más próximo hasta el más lejano) y decidiremos si participamos en el sorteo o no.

Sea S la solución de obtener el máximo beneficio total cuando tenemos 1..n lotes(denotado por B(1)), en el primer paso de decisión habremos considerado el lote de la semana 1. Tendremos dos opciones:

a) No participar en el sorteo. En ese caso, no gastamos nada pero tampoco ganamos nada la semana 1 y vemos qué hacer el resto de semanas (2..n.). Por eso B(1) = 0+ B(2). Es evidente que el subproblema B(2) debe encontrar el beneficio óptimo, puesto que B(1) tiene el mismo beneficio que dicho subproblema.
b) Participar en el sorteo. En ese caso, pagamos 2€ y, también, la tarifa del hacker t(1). Vamos a ganar seguro, así que tendremos un lote por valor v(1). Dado que no queremos levantar sospechas, vamos a estar sin ganar K-1 semanas, en las cuales no habrá gastos ni beneficios. A partir de ahí vemos qué hacer. Por ello, B(1) = v(1) – 2 – t(1) + B(1+K). Si la solución óptima se creó considerando que había que jugar el día 1, entonces v(1), 2 y t(1) son constantes y hay que resovler B(1+k) de la forma mejor posible (máxima) para obtener el mejor beneficio. De lo contrario la solución S, con beneficio B(1) no sería óptima.

Razonando de igual manera para los subproblemas se demuestra el principio de subestructura óptima.

b)(3) Encontrar la recurrencia que calcula el beneficio máximo.

Llamemos B(i) al beneficio máximo que se puede conseguir participando en sorteos entre la semana i y la semana n.

$$B(i) = \begin{cases} 0 & i > n \\ B(i+1) & i \leq n \wedge v(i) - 2 - t(i) \leq 0 \\ \max\{B(i+1), v(i) - 2 - t(i) + B(i+K)\} & i \leq n \wedge v(i) - 2 - t(i) > 0 \end{cases}$$

c)(0.5) Definir la estructura de datos necesaria para el algoritmo de programación dinámica (dimensiones, cómo se rellena, en qué celda se encuentran los casos base y la solución al problema).

Vamos a necesitar un array de longitud n+1, para los valores de i desde 1 hasta n+1 (Si el índice del array empieza en 0, se corresponderán con los valores desde 0 hasta n).

La celda que representa el caso base es aquella más a la derecha (n). La celda que resuelve el problema es la primera (0), que representa B(1).Por tanto, el array se llena de derecha a izquierda.

d)(1.5) Implementar un algoritmo de programación dinámica que el beneficio máximo que podemos conseguir.

```
public static double sorteo(int K, double []v, double[] t) {
            int n = v.length;
            double [] B = new double[n+1];

            //Caso base: n
            B[n] = 0;
            for (int i = n-1; i>=0; i--) {
                  double beneficioDia = v[i]-t[i]-2;
                  if (beneficioDia <= 0) {
                        B[i] = B[i+1];
                  }else {
                        B[i] = Math.max(B[i+1], beneficioDia +
B[i+K]);
                  }
            }

            return B[0];
      }
```

e) (0.5) Indicar el orden de crecimiento temporal y espacial del algoritmo propuesto.

Necesitamos un array auxiliar de tamaño n+1, por lo que $E(n) \in \Theta(n)$.

En cuanto al coste temporal, hay que rellenar el array (n+1 casillas) y en cada casilla realizaremos una comparación (O(1)) y unas cuantas operaciones aritméticas (coste O(1)). Por tanto, el coste temporal $T(n) \in \Theta(n)$.

f) (1) Indicar qué cambios habría que hacer al algoritmo del apartado **d** si queremos conocer las semanas que hay que participar en los sorteos.

El tipo de salida de la función va a ser un Conjunto de números de semana. Una vez creada la tabla que se basa en la recurrencia, la vamos a recorrer en orden inverso para reconstruir la solución.

```
public static Set<Integer> sorteo2(int K, double []v, double[] t) {
		int n = v.length;
		double [] B = new double[n+1];
		Set<Integer> sol = new HashSet<>();
		int i;

		//Caso base: n
		B[n] = 0;
		for (i = n-1; i>=0; i--) {
			double beneficioDia = v[i]-t[i]-2;
			if (beneficioDia <= 0) {
				B[i] = B[i+1];
			}else {
				B[i] = Math.max(B[i+1], beneficioDia +
B[i+K]);
			}
		}

		//Reconstruimos
		i = 0; //Semana 1
		while (i <= n+1) {
			if (B[i]==B[i+1]) {//No se apostó la semana i
				i++;
			}else {//Se apostó la semana i
				sol.add(i+1); //En la posición i está
representada la semana i+1.
				i += K;
			}
		}

		return sol;
	}
```

2.- (2.5) Resolver el problema del ejercicio 1 mediante algoritmos voraces.

a) (0.75) Definir todos los elementos necesarios para construir un algoritmo voraz que nos indique la colección de semanas que hay que participar para obtener máximo beneficio.

Solución parcial: será una lista de los números de semana en los que se va a apostar.

Candidatos factibles: los días que no han sido seleccionados que sean compatibles con los que ya están seleccionados. Dos días serán compatibles si la distancia entre ellos es al menos K-1.

Función objetivo: el beneficio total obtenido en los sorteos de las semanas seleccionados.

Función de terminación: no quedan semanas compatibles que no hayan sido seleccionadas.

Función de selección: Seleccionar la semana de máximo beneficio de los candidatos factibles.

b) (1.25) Implementar el algoritmo.

```
public static Set<Integer> sorteoVoraz(int K, double[]v, double []t){
            Set<Integer> sol = new HashSet<>();
            //Inicialmente los candidatos son todas las semanas
            Set<Integer> candidatos = new HashSet<>();
            for (int i = 1; i <= n; i++) {
                  candidatos.add(i);
            }

            //Construimos la solución
            while (!candidatos.isEmpty()) {
                  Integer mejorCandidato =
seleccionarMasValor(candidatos,v,t);
                  sol.add(mejorCandidato);
                  eliminarNoCompatibles(mejorCandidato,candidatos,K);
            }

            return sol;
      }
```

El método seleccionarMasValor(Set<Integer>, double [], double []) escoge el número de semana del conjunto que proporciona mayor beneficio.

El método eliminarNoCompatible(Integer, Set<Integer, int) elimina del conjunto todos los números de semana que no son compatibles con el número dado como primer parámetro. Es decir, si el número es i, elimina desde i-k+1 hasta i+k-1.

Otro enfoque voraz sería el que intenta participar todas las veces que pueda, es decir, apuesta el primer día, deja pasar K-1 días, vuelve a apostar, y así sucesivamente.

Tampoco es un enfoque correcto para toda instancia del problema, pero el algoritmo resultante sería más eficiente que el anterior.

```
public static Set<Integer> sorteoVoraz2(int K, double[]v, double []t){
            Set<Integer> sol = new HashSet<>();

            int i = 0;
            while (i < v.length) {
                  sol.add(i+1);
                  i = i+K;
            }

            return sol;
      }
```

c) (0.5) Estudiar la corrección del algoritmo.

El algoritmo no devuelve el óptimo en todos los casos.

Contraejemplo: Tenemos los lotes con valor [100, 150, 100]. El hacker nos cobra [10, 5,10] y K = 2.

El beneficio de cada semana sería [100-2-10, 150-2-5,100-2-10]= [88,143,88]

Según nuestro algoritmo escogeríamos participar la semana 2 y obtendríamos un beneficio de 143.

Sin embargo, la solución óptima escogería participar la primera y tercera semana, con un beneficio de 176.