Nº matrícula:_____ **Nombre:** _____

Apellidos: _____

Sesión 1.4. Divide y Vencerás

Sean A y B dos vectores de N elementos enteros no ordenados y que pueden estar repetidos, y sea v un valor entero. Se pide implementar un procedimiento que determine si existen o no dos posiciones i y j tales que A[i] + B[j] = v. El procedimiento sólo devolverá un valor lógico (true/false) en función de si existen esas posiciones. El procedimiento debe tener una complejidad O(N log N).

a) (3.5 puntos). Implementar la función `check2pos` basado en Divide y Vencerás con complejidad O(N log N) en el caso peor (donde N es el tamaño del vector) que devuelva un booleano en función de si se cumple, o no, la condición previamente descrita.

Ejemplo:

0	1	2	3	4	5	6	7	8
-5	**-2**	**-9**	**-4**	**1**	**5**	**7**	**0**	**-3**

0	1	2	3	4	5	6	7	8
8	**-1**	**2**	**4**	**2**	**-2**	**1**	**0**	**1**

check2pos(A, B, -1)=true
check2pos(A, B, 10)=false

```
boolean check2pos(int[] vector1, int[] vector2, int v){

}
```

Pregunta 2 (3 puntos). Justifica que la complejidad del algoritmo desarrollado en el apartado anterior para el caso peor es O(N log N), indicando qué valores deben darse en los datos de entrada para estar en ese caso peor.

Examen de Algorítmica y Complejidad (Plan 2014) — **24 de junio de 2020**

Nº matrícula:_____ **Nombre:** _____

Apellidos: _____

1) **Problema 1 (5 puntos).** Sean A y B dos vectores de *N* elementos enteros no ordenados y que pueden estar repetidos, y sea *v* un valor entero. Se pide implementar un procedimiento que determine si existen o no dos posiciones *i* y *j* tales que *A[i] + B[j] = v*. El procedimiento sólo devolverá un valor lógico (true/false) en función de si existen esas posiciones. El procedimiento debe tener una complejidad O(N log N).

   a) **(3.5 puntos)** Diseñar el procedimiento basado en Divide y Vencerás con complejidad O(N log N) en el caso peor[1] (donde N es el tamaño del vector) que devuelva un booleano en función de si se cumple, o no, la condición previamente descrita.

   Ejemplo:

0	1	2	3	4	5	6	7	8
-5	-2	-9	-4	1	5	7	0	-3

0	1	2	3	4	5	6	7	8
8	-1	2	4	2	-2	1	0	1

```
boolean Check2pos (int[] vector1, int[] vector2, int v)

Check2pos (A, B, -1) = true
Check2pos (A, B, 10) = false
```

Es esencial ordenar uno de los vectores para guiar la búsqueda y reducir la complejidad del algoritmo

```
public class EjercicioDyV {

    private static boolean checkPosAux(int v1, int[] v2, int i02, int iN2, int valor) {
                boolean encontrado = false;
        if (i02==iN2) {
            return v1+v2[i02]==valor; //O(1)
        } else {
            int k2 = (i02 + iN2) / 2; //O(1)
            //O(1 * T(N/2)+1, N>1, 2º caso TM, O (log N)
            if(v1+v2[k2]>valor) //busco lado izqdo
                return checkPosAux(v1, v2, i02, k2, valor);
            else if (v1+v2[k2]==valor)
                return true;
            else
                return checkPosAux(v1, v2, k2+1, iN2, valor); //busco lado dcho.
        }
    }
```

[1] Desarrollar un algoritmo que tenga una complejidad diferente a O(n log N) en el caso peor conllevará una puntuación de 0 en la pregunta.

```
    private static boolean checkPos(int[] v1, int[] v2, int valor) {
        boolean stop=false, aux=false;
        int i=0;
        ordenarMergeSort(v2); //se ordena vector 2, O(n LogN).
        while (!stop) {  //O(N): v1
            aux = checkPosAux(v1[i], v2, 0, v2.length - 1, valor);
            if (i==v1.length-1 || aux)
                stop=true;
            i++;
        }
        return aux;
    }

    void mergeSortAux (int[] vector, int i0, int iN){
      if (i0 >= iN)
          return;
      else {
         int k= (i0 + iN) /2;
         mergeSortAux(vector, i0, k);
         mergeSortAux(vector, k+1, iN);
         merge(vector, i0, k, iN);
      }
    }

    void ordenarMergeSort (int[] vector) {
      mergeSortAux(vector,0, vector.length-1);
    }

}
```

b) (**1.5 puntos**) Justifica que la complejidad del algoritmo desarrollado en el apartado anterior para el caso peor es O(N log *N*), indicando qué valores deben darse en los datos de entrada para estar en ese caso peor.

El caso peor se producirá cuando dados dos vectores de longitud variable no exista ninguna combinación tal que la suma de cuales quiera de sus dos elementos sea la del valor buscado, lo que originará que sea necesario recorrer todos los elementos de ambos vectores.

La demostración es sencilla, la solución está formada por la llamada a la función ***checkPos***(int[] v1, int[] v2, int valor), que previamente a recorrer el vector 1, ordena el vector 2 utilizando un algoritmo de ordenación, por ejemplo el Merge Sort, que tiene una complejidad de O(N LogN).

checkPos a su vez llama a la función auxiliar ***checkPosAux***(int v1, int[] v2, int i02, int iN2, int valor), que pasando el valor de cada uno de los elementos del primer vector (v1), simplemente hace una búsqueda binaria en la parte izquierda o derecha del vector, comprobando la condición: *A[i] + B[j] = v*

En el caso de que la suma de los dos elementos sea mayor que el valor 'objetivo' se busca en el lado izquierdo del vector y, en caso contrario, se buscaría en el derecho. Como puede verse en la solución, la búsqueda se detiene en cuanto se cumple la condición. Si aplicamos el TM a la función auxiliar (2º caso del TM) nos genera una complejidad O(log N).

$$T(N) = \begin{cases} \Theta(1) & N = 1 \\ T\left(\frac{N}{2}\right) + \Theta(1) & N > 1 \end{cases}$$

$$T(N) = \begin{cases} \Theta(1) & N = 1 \\ 1 \cdot T\left(\frac{N}{2}\right) + \Theta(N^0) & N > 1 \end{cases}$$

Teorema Maestro:

(2º Caso) $\text{Log}_2(1)=0$	**Complejidad: $\Theta(\log N)$**

Por lo tanto, la función ***checkPosAux*** tendrá una complejidad de O(log N).
Respecto a la función ***checkPos***, consta de una llamada al método de ordenación ***MergeSort***, que tiene una complejidad O(N log N), y a continuación llama desde un bucle a la función ***checkPosAux***. La complejidad del bucle será de O(N log N), dado que en el peor caso se realizan N iteraciones del bucle y cada una tiene complejidad O(log N). Dado que la ejecución de la llamada a MergeSort y el bucle se realiza en secuencial, la compleidad final de la función ***checkPosAux*** será max(O(NlogN), O(NlogN)) = O(NlogN)

El caso peor se produce cuando no existen dos posiciones i y j tales que A[i] + B[j] = v.