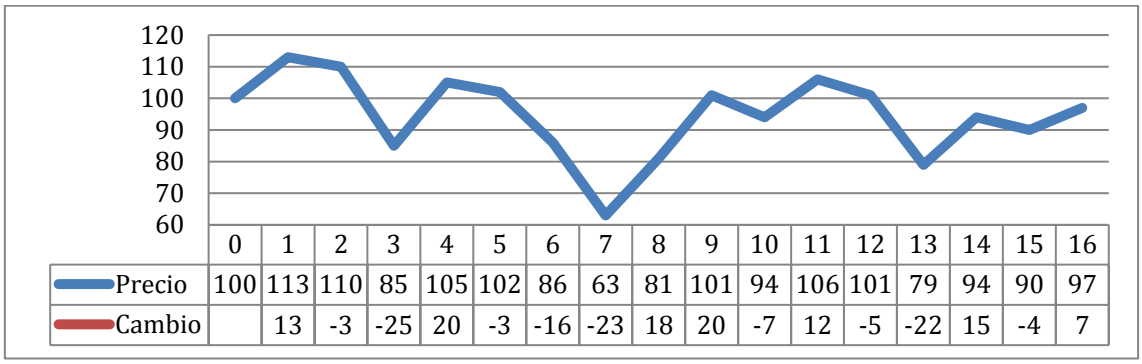
Análisis y Diseño de Algoritmos

SubArray Máximo

Definición del problema[1]

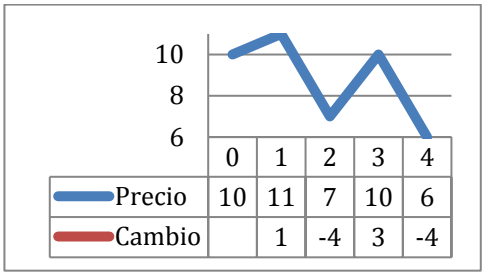
Supongamos que le ofrecen la oportunidad de invertir en bolsa en acciones una sociedad anónima. El precio de las acciones es muy volátil y solo se puede comprar un día y vender otro día, pero gracias a un conocimiento previo del mercado, y a técnicas sofisticadas de inteligencia artificial somos capaces de predecir con exactitud el precio de las acciones durante un periodo de *n* días.[2] El objetivo es maximizar el beneficio. La figura 1 muestra el precio de la acción durante un periodo de 17 días. Se puede comprar cualquier día desde el día 0 al 16. Por supuesto, lo que interesa es comprar "bajo" y vender "alto", es decir comprar al menor precio posible y vender después al mayor precio posible, a fin de maximizar el beneficio. En la figura 1, el mínimo precio se alcanza el día 7, pero el máximo se alcanza el día 1, por lo que no es posible esta opción.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Precio	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Cambio		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Figura 1. En esta gráfica se indica el precio de la acción cada día desde el día 0 al 16. La fila "Cambio", indica la variación de precio entre un día y el siguiente.

Podría pensarse que el máximo beneficio se obtiene o bien buscando primero el precio mínimo y a continuación el máximo en un día posterior; o bien buscando el precio máximo y a continuación buscando el mínimo en algún día anterior y quedándose con el mayor de los dos valores. Por ejemplo, en la figura 1, el máximo se obtiene comprando el día 7 y vendiendo el día 11. Pero esta estrategia no siempre funciona. La figura 2 muestra un contraejemplo en donde esta estrategia no funciona. El máximo beneficio se obtiene comprando el día 2 y vendiendo el día 3; pero ni el día 2 es el mínimo absoluto de la serie (que se alcanza el día 4), ni el día 3 es el máximo absoluto (que se alcanza el día 1).



	0	1	2	3	4
Precio	10	11	7	10	6
Cambio		1	-4	3	-4

Figura 2.

Solución por "fuerza bruta".

Para resolver este problema se puede diseñar una estrategia de "fuerza bruta", simplemente tomado todos los posibles pares de días de manera que el día de la compra sea anterior al de la venta, calculando el beneficio en cada caso y seleccionando el máximo. Un periodo de n días generaría $\binom{n}{2}$ pares. Puesto que $\binom{n}{2}$ es $\Theta(n^2)$; y la evaluación del beneficio se puede hacer en tiempo constante, esta solución implica un tiempo $\Omega(n^2)$.

---

[1] Esta práctica esta adaptada del libro "Cormen T.H, Leiserson, C.E, Rives, R.L. & Stein,C. (2009): Introduction to Algorithms. MIT Press, Cambridge."

[2] Actualmente no hay ninguna forma fiable de hacer estas predicciones.

Solución usando "Divide y Vencerás".

Para buscar una solución más eficiente, lo primero que debemos hacer es transformar la matriz de entrada de manera que obtengamos el cambio o variación de precio entre dos días seguidos. Esta secuencia forma un array al que llamaremos *A*. Una vez hecho esto, el problema consiste en obtener la subsecuencia de días tales que la suma de los valores del vector *A* sea máxima. Por ejemplo, la figura 1 muestra la fila "Cambio", que corresponde al array *A* y contiene la variación de precio diaria. Si nos fijamos en la subsecuencia que va desde el día 8 hasta el día 11, la suma de los valores del array "Cambio" es 18+20-7+12 = 43. Ninguna otra subsecuencia da un valor mayor. A esta subsecuencia la llamaremos un "*subarray máximo*" (el subarray máximo no tiene porque ser único). Una vez obtenida, la solución al problema bursátil es comprar el día anterior, es decir, el día 7, y vender el mismo día en el que finaliza el subarray máximo (día 11).
A primera vista esta transformación parece que no ayuda mucho en la solución de fuerza bruta, pero veamos que ocurre si aplicamos una solución basada en "divide y vencerás". Ni que decir tiene que el problema solo es interesante cuando el array A tiene algunos valores negativos, ya que si todos los valores son positivos la solución trivial es que el subarray máximo coincide con el array completo.
Para resolver este problema mediante el método de "divide y vencerás" supongamos que tenemos un array *A[inf..sup]*. Supongamos que dividimos el array por un punto medio *med*, y formamos dos subarrays *A[inf..med]* (la parte izquierda) y *A[med+1..sup]* (la parte derecha). Cualquier subarray *A[i..j]* del array original *A[inf..sup]* o bien esta incluido en la parte izquierda; o bien esta en la parte derecha; o bien esta "a caballo" entre ambas, es decir, comienza en la parte izquierda y termina en la parte derecha. En concreto el *subarray máximo* de *A[inf..sup]*, estará en alguna de estas tres circunstancias:

- Completamente dentro de *A[inf..med]],* es decir*: inf* $\leq i \leq j \leq$ *med*.
- Completamente dentro de *A[med+1..sup],* es decir: *med* $< i \leq j \leq$ *sup*.
- Entre ambos, es decir: : $i \leq$ *med* $< j \leq$ *sup*.

De hecho, el máximo subarray del array completo debe tener la máxima suma de cualquier array que esté en alguno de estos tres casos. Para resolver el problema inicial, debemos obtener los *subarrays máximos* en cada uno de estos tres casos. En los dos primeros, se resuelve de forma recursiva, porque los subproblemas son instancias menores del problema inicial. En el tercer caso, en el que el array esta entre ambas partes la solución se puede obtener de forma lineal. Para ello basta con obtener partiendo del punto medio *med*, y hacia atrás hasta llegar al valor inicial *inf*, el valor i que hace máxima la suma por el extremo izquierdo, e igualmente, a partir del valor *med+1* y hasta llegar al final, el valor *j* que hace máxima la suma por el extremo derecho. Por ejemplo, para el array del ejemplo de la figura 1, supongamos que queremos hallar el *subarray máximo* del array *A[1..16],* y que tomamos el valor medio *med=8*. Recursivamente calculamos el subarray máximo de los arrays *A[1..8]* y *A[9..16]* (que resultan ser los subarrays *A[4..4]* y *A[9..11]* con sumas de *20* y *25* respectivamente). A partir del punto medio *8*, y recorriendo hacia atrás buscamos el valor *i* de la suma de *A[i,8],* este valor resulta ser *i=8*, con un valor de *suma(A[8..8]) = 18*. Igualmente se busca el valor *A[9..j]* que hace máximo el valor de la suma, y resulta ser *suma(A[9..11]) = 25*. Por lo que el tercer subarray máximo es *A[8..11],* con un valor de suma de *18+25=47*. Finalmente, el subarray máximo del array original *A[1..16]* es subarray cuya suma es mayor de entre los tres casos: *A[4..4]; A[9..11] y A[8..11]*. En este caso resulta ser el subarray *A[8..11]* con una suma de *47*.
Si analizamos esta solución nos damos cuenta de que la ecuación de recurrencia es:

$$T(n) = 2T(n/2) + \Theta(n) + \Theta(1)$$
$$T(1) = \Theta(1)$$

en donde, 2T(n/2) corresponden al coste de resolver recursivamente los subproblemas izquierdo y derecho; Θ(n) es el coste de resolver el subproblema en el caso de que el punto medio este en mitad del subarray máximo y Θ(1) corresponde a las operaciones de selección del máximo de los tres. Aplicando el teorema maestro la complejidad de esta solución, una vez obtenido el array A de diferencias, resulta ser $T(n) = \Theta(n \log(n))$.

Solución lineal.
Finalmente, vamos a ver que existe una solución de coste lineal a este problema. La idea consiste en empezar por el extremo izquierdo del array e ir avanzando progresivamente hacia la derecha, llevando la cuenta del valor de la suma del máximo subarray encontrado hasta el momento. Así por ejemplo, si conocemos el subarray máximo del array *A[1..j],* y queremos hallar el subarray máximo del array *A[1..j+1],* entonces solo hay dos posibilidades, o bien el *subarray máximo* de *A[1..j+1],* es el mismo que el de *A[1..j]*;

o bien es un subarray de la forma *A[i..j+1]*, siendo*: 1* ≤ *i* ≤ *j*+1. El valor de *i* se puede obtener en tiempo constante sabiendo cual es el *subarray máximo* de *A[1..j]*.

Se pide:

Completar la implementación en Java del problema descrito definiendo tres clases denominadas:

```
SubArrayMaximoN.java
SubArrayMaximoNLOGN.java
SubArrayMaximoN2.java
```

herederas de la clase denominada `SubArrayMaximo.java` (ver ficheros auxiliares de la práctica).

```
public abstract class SubArrayMaximo {
        public abstract SolucionSubArrayMaximo resolver(int[] array);
        ...
}
```

redefiniendo el método `resolver(int[])`. Este método tiene como entrada un array de *n* valores enteros y como salida el valor de los extremos de una secuencia máxima y el valor del beneficio obtenido.

Por ejemplo, la ejecución del programa correspondiente al ejemplo de la figura 1 dará la siguiente salida:

```
> java Bolsa
100 113 110 85 105 102 86 63 81 101 94 106 101 79 94 90 97
comprar dia 7; vender dia 11; beneficio = 43;
```

Se acompaña un fichero de ejemplos ya resueltos que pueden probarse y que una vez que se hayan implementado las tres clases correctamente dará los siguientes resultados:

```
> java ProbarEjemplos
El metodo de fuerza bruta supera las pruebas de los ejemplos
El metodo de Divide y Venceras supera las pruebas de los ejemplos
El metodo de Complejidad Lineal supera las pruebas de los ejemplos
```

Evaluación de la práctica

* No utilice paquetes de Java, coloque todas las clases en un mismo directorio.

* Para realizar este ejercicio NO ES NECESARIO UTILIZAR ECLIPSE, aunque se puede hacer si se quiere y se sabe como. Para compilar el código que se entrega solo es necesario poner los ficheros en una misma carpeta, abrir una terminal y utilizar las instrucciones:

```
>javac *.java
```

* En caso de utilizar Eclipse, el fichero `ejemplos.txt` debe colocarse en la carpeta raíz del proyecto.

* Para evaluar la practica se deben enviar solamente los tres ficheros que contienen las clases que implementan el método `resolver(int[])`.

* No basta con implementar los tres métodos iguales. Debe comprobarse empíricamente que la complejidad de la implementación por fuerza bruta es $\Theta(n^2)$, que la complejidad de la solución mediante "divide y vencerás" es $\Theta(n \log n)$ y que la mejor solución es de complejidad es $\Theta(n)$. Si no se cumple esta condición la implementación no se considerará correcta.