UNIVERSIDAD DE MÁLAGA

Análisis y Diseño de Algoritmos
Parcial 2 (Octubre 2025)

Nombre: **Grupo:**

1.(10 puntos) Parte teórica. Supongamos que hemos registrado durante un año los precios diarios en varias tiendas online de un producto que nos interesa. Para cada tienda disponemos de una lista de precios $P = [p_0, \dots, p_{n-1}]$, donde $P[i]$ representa el precio en el día i.
Sabemos, por la experiencia de años anteriores, que en cada tienda el precio sigue uno y solo uno de los siguientes patrones (ver ejemplos):

1) No cambia durante todo el año,
2) Forma de V: el precio baja, alcanza un mínimo único y, al día siguiente, sube.
3) El precio baja se mantiene en el valor mínimo durante varios días y vuelve a subir.
4) Baja constantemente hasta su valor mínimo.
5) Comienza a subir sin parar.

Nuestro objetivo es diseñar un algoritmo que se pueda aplicar a la lista de precios de cada tienda, P, para detectar el primer momento en que se alcanza ese precio mínimo en dicha tienda.
Ejemplos:

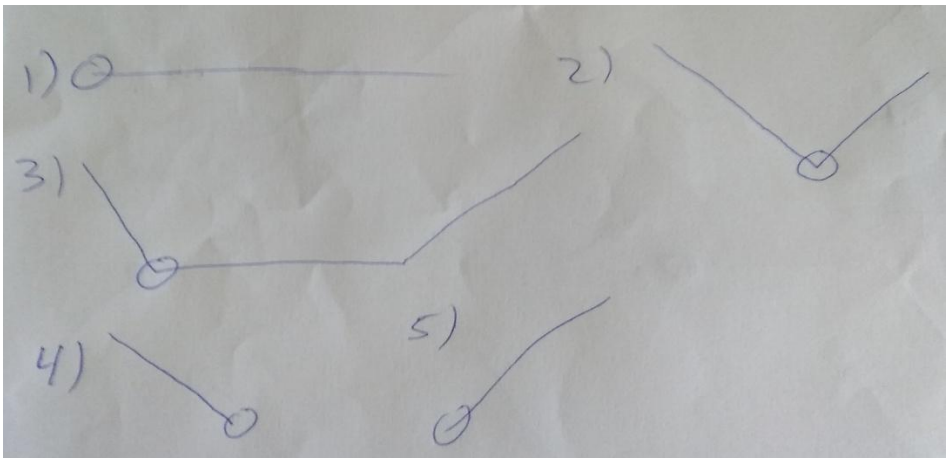
Si P= {10,10,10,10}, entonces hay que devolver 0 porque es el primer momento en que se encuentra el mejor precio.
Si P = {10,5,7,12}, entonces hay que devolver 1.
Si P = {10,9,7,7,8}, entonces hay que devolver 2.
Si P = {10,8,7,6}, entonces hay que devolver 3.
Si P = {10,11,12,13}, entonces hay que devolver 0.

a) *(2 ptos)* Diseña un algoritmo por Fuerza Bruta que resuelva el problema.
b) *(1 pto.)* Indica justificadamente el orden de crecimiento en el peor caso del algoritmo de Fuerza Bruta.
c) *(6 pto)* Diseña un algoritmo Divide y Vencerás, que resuelva el problema y cuya complejidad en el peor caso sea *O(log n).* Indicar claramente el caso general y los casos base.
d) *(1 pto)* Analiza la complejidad del algoritmo Divide y Vencerás y demuestra que se ajusta al orden de crecimiento indicado en el apartado c).

a) Para resolver por Fuerza Bruta el problema necesitaremos evaluar todos los elementos y ver cuál es el que tiene el valor mínimo.

```
int encontrarMinimo_FB(int[] P) {
      int posMin = 0;
      for (int i = 1; i< P.length; i++) {
            if (P[i] < P[posMin]) {
                  posMin = i;
            }
      }
      return posMin;
}
```

Escenarios:



Otra opción sería darnos cuenta de que en los escenarios planteados el mínimo se encuentra cuando el precio se repite o empieza a subir. Es decir, la primera vez que ocurra que P[i+1] >= P[i]. El escenario 4 podría dar problemas, pero, en ese caso, el contador llegará a la última posición que es la que tiene el mínimo.

```
int encontrarMinimo_FB(int[] P) {
      int pos = 0;
      while (pos < P.length - 1 && P[pos] > P[pos + 1]) {
            pos++;
      }
      return pos;
}
```

b) En el peor caso, el array tendrá que evaluar todos o prácticamente todos los elementos en el array P. Dado que siempre se hacen operaciones constantes, tendremos un número O(n) de iteraciones que repiten operaciones O(1). Por tanto, por la regla de la multiplicación la complejidad de nuestra solución será O(n).

c) Dado que me exigen que la complejidad de la solución sea O(log n), sabemos que la única forma de conseguirlo cuando nos dan n elementos es realizar comprobaciones en tiempo constante e ir descartando parte (una mitad, dos tercios, tres cuartos,…) de esos n elementos tras cada comprobación (estructura similar a la de la búsqueda binaria).
Por ello, vamos a evaluar el elemento de la mitad del array e intentar identificar en qué mitad puede estar el mínimo. Si el elemento medio está en la parte del array que decrece, entonces el mínimo estará en la mitad derecha. Si embargo, si está en la parte que crece o se mantiene constante, el mínimo estará en la izquierda (incluyendo al punto medio).
Suponiendo que la signatura de la función es `int encontrarMinimo(int[] P, int ini, int fin)` podríamos plantear el siguiente pseudocódigo del caso general:

1.- Calcular punto medio m. m = (ini + fin)/2
2a).- SI P[m] > P[m+1], ENTONCES buscamos el mínimo a la derecha, en el rango [m+1,fin]
2b).- SI P[m] <= P[m+1], ENTONCES buscamos el mínimo a la izquierda, en el rango [ini,m]

Nuestro algoritmo debería terminar cuando sólo quede un elemento, que será el mínimo. Este sería el caso base.

Para tamaños mayores que uno, la división del problema que hacemos nos asegura que terminaremos trabajando con un rango de un elemento como poco.

d) Analicemos la complejidad para una lista de elementos lo suficientemente grande. El caso general sólo realiza operaciones constantes y, como mucho, una llamada recursiva para la mitad de los valores de los que partíamos, por lo que $T(n) = T\left(\frac{n}{2}\right) + A,\ A \in O(1)$
Dado que *a = 1, b = 2, A* es un polinomio de grado *d = 0*, y *a = 1 = 2^0 = b^d*, podemos afirmar por el Teorema Maestro Reducido que $T(n) \in \Theta(n^0 \cdot \log n) \equiv \Theta(\log n)$

2. (*10 puntos*) Parte práctica. Implementa el algoritmo Divide y Vencerás diseñado en el ejercicio anterior.

```
public static int encontrarMinimo(int[] P) {
     return encontrarMinimo_DyV(P, 0, P.length - 1);
}

// Precondición: ini<=fin
private static int encontrarMinimo_DyV(int[] P, int ini, int fin) {
     int res = -1;
     if (ini == fin) {// n=1
          res = ini;
     } else {
          int m = (ini + fin) / 2;
          if (P[m] > P[m + 1]) {
               res = encontrarMinimo_DyV(P, m + 1, fin);
          } else { // P[m] <= P[m+1]
               res = encontrarMinimo_DyV(P, ini, m);
          }
     }
     return res;
}
```