Ejercicio 1: Determinar el costo del algoritmo (en número de operaciones básicas) y expresar su eficiencia en notación Big-O (Notación asintótica)

a)

```
Algoritmo CuadradoPositivosMatriz
    Definir M, N, i, j Como Entero
    Definir matriz, matrizResultado Como Entero
    Escribir "Ingrese la cantidad de columnas (N): "
    Leer N
    Dimension matriz[N,N]
    Dimension matrizResultado[N,N]

    Para i ← 0 Hasta N-1 Con Paso 1 Hacer
        Para j ← 0 Hasta N-1 Con Paso 1 Hacer
            Escribir "Ingrese el valor en la posición [", i, "][", j, "]: "
            Leer matriz[i,j]
        FinPara
    FinPara
    Para i ← 0 Hasta N-1 Con Paso 1 Hacer
        Para j ← 0 Hasta N-1 Con Paso 1 Hacer
            Si matriz[i,j] > 0 Entonces
                matrizResultado[i,j] ← matriz[i,j] * matriz[i,j]
            Sino
                matrizResultado[i,j] ← matriz[i,j]
            FinSi
        FinPara
    FinPara
    Escribir "Matriz resultante:"
    Para i ← 0 Hasta M-1 Con Paso 1 Hacer
        Para j ← 0 Hasta N-1 Con Paso 1 Hacer
            Escribir Sin Saltar matrizResultado[i,j], " "
        FinPara
        Escribir "" // salto de línea
    FinPara
FinAlgoritmo
```

b)

```
Algoritmo notasConPara
    Definir cuenta, cantidadNotas,suma,nota Como Entero
    Definir promedio Como Real
    cantidadNotas = 0
    suma = 0
    promedio = 0
    Escribir "Cuantas notas desea ingresar?"
    leer cantidadNotas
    Para cuenta Desde 1 Hasta cantidadNotas Hacer
        Escribir "Ingrese la nota ",cuenta
        Leer nota
        suma = suma + nota
    FinPara
    promedio = suma / cantidadNotas
    Escribir "Obtuviste un promedio de ",promedio
FinAlgoritmo
```

Ejercicio 2:

- Se deberá elaborar un algoritmo en pseudocódigo y en Java que calcule las raíces reales de una ecuación cuadrática. Verificar que el coeficiente $a \neq 0$ y el discriminante $(b^2 - 4ac) \geq 0$, de modo que existan raíces reales.
  Una vez obtenidas las raíces, indicar si cada una de ellas se encuentra dentro de un vector de números reales dado.
- Determinar el costo del algoritmo (en número de operaciones básicas) y expresar su eficiencia en notación Big-O.

Ejercicio 3:

Lea detenidamente cada problema, por cada uno de ellos deberá responder los siguientes puntos:

a) Estrategias de la solución Divide y Conquista, indicando cada uno de los elementos que se requieren para la resolución de esa técnica.
b) Pseudocódigo del Algoritmo de la resolución del problema, el cual debe seguir la estrategia definida en el punto a)
c) Calculo de la Complejidad Temporal, justificando la misma en forma detallada.

1. Diseñar un algoritmo que determine si una secuencia de n caracteres está ordenado alfabéticamente.
2. Diseñar un algoritmo que calcule a^n cuando n es una potencia de 2.
3. Sea A[1..n], n>= , un vector de enteros diferentes y ordenados crecientemente, tal que alguno de los valores pueden ser negativos. Diseñar un algorimo que devuelva un índice natural k, 1<= k <= n, tal que A[k] = k, siempre que tal índice exista.

Ejercicio 4: Plantear un algoritmo que ejemplifique cada uno de los siguientes casos del método de resolución de recurrencias:

1. Sustracción:
   - Caso 1: a=1
   - Caso 2: a>1
   - Caso 3: a<1
2. División:
   - Caso 4: a = b^k
   - Caso 5: a < b^k
   - Caso 6: a > b^k

Para cada caso:

- Describir la recurrencia.
- Plantear un algoritmo en pseudocódigo que represente el proceso.
- Implementar el algoritmo en Java, manteniendo la misma lógica.
- Determinar el orden de complejidad en notación asintótica (O).

Ejercicio 5 – Graficar los siguientes órdenes de complejidad:

   - O(1)
   - O(log n)
   - O(n)
   - O(nlog n)
   - O(n^2)
   - O(2^n)
- Usar valores positivos de n (tamaño de muestra de datos) para observar el crecimiento de cada función.
- Extraer conclusiones:

- ¿Qué algoritmos escalan mejor a medida que crece n?
- ¿Qué órdenes resultan inviables para grandes volúmenes de datos?

Ejercicio 6: - Analizar el funcionamiento y la eficiencia de los algoritmos Merge Sort y Quick Sort recursivos.

a) Implementar los algoritmos Merge Sort y Quick Sort en Java asegurándose de respetar la recursión característica de cada método.
b) Seleccionar un caso prueba

1. Un arreglo de tamaño par (por ejemplo 8 elementos).
2. Un arreglo de tamaño impar (por ejemplo 7 elementos).

c) Ejecutar ambos algoritmos con los casos prueba y mostrar:

3. Cada paso de la división y combinación (Merge Sort).
4. Cada paso de la partición y elección del pivote (Quick Sort).

d) Analizar el funcionamiento paso a paso de cada algoritmo y discutir las diferencias observadas entre tamaño par e impar.

a) Resolver la recurrencia y expresar la complejidad temporal en notación Big-O.

Ejercicio 7:

1. Proponer un algoritmo alternativo para seleccionar el pivote, distinto de simplemente elegir el primer elemento.
2. Implementar Quick Sort con esta nueva estrategia de pivote y comparar el comportamiento con el Quick Sort original:
   - Número de pasos o comparaciones
   - Casos prueba par e impar