Taller 1
DALGO

1. Soluciones las siguientes ecuaciones de recurrencia y determine el orden de complejidad:

(a) (13 points) $T(n) = 3T(\frac{n}{2}) + n$.

Punto 1:

a) $T(n) = 3T(\frac{n}{2}) + n$ $\quad n = 2^m = g(m) \;//\; T(g(m)) = Y(m)$

$m = \log_2(n)$

$T(2^m) = 3T(\frac{2^m}{2}) + 2^m$

$T(2^m) = 3T(2^{m-1}) + 2^m$

$T(g(m)) = 3T(g(m-1)) + 2^m$

$Y(m) = 3Y(m-1) + 2^m \;//\; \{Y(m) - 3Y(m-1) = 2^m\}$

Homogenea: $Y(m) - 3Y(m-1) = 0$

$\lambda - 3 = 0 \;//\; \lambda = 3 \;//\; \{h(m) = C_1 3^m\}$

Particular: $P(m) - 3P(m-1) = 2^m \;//\; P(m) = C_2 2^m$

$Y(m) - 3Y(m-1) = 2^m$

$(C_2 2^m) - 3(C_2 2^{m-1}) = 2^m \;//\; C_2(2^m - 3 \cdot 2^{m-1}) = 2^m$

$C_2(2^m - 3 \cdot \frac{2^m}{2}) = 2^m \;//\; C_2 = 2^m / (2^m(1 - \frac{3}{2})) \;//\; C_2 = 2^m / (2^m(-\frac{1}{2}$

$C_2 = 1/(-1/2) \;//\; C_2 = -2 \;//\; P(m) = -2 \cdot 2^m \;//\; P(m) = -2^{m+}$

Completa: $Y(m) = h(m) + P(m)$

$Y(m) = C_1 3^m - 2^{m+1} \;//\; T(n) = C_1 3^{\log_2(n)} - 2^{\log_2(n)+1}$

Complejidad: $O(3^{\log_2(n)})$

(b) (13 points) $T(n) = 2T(\frac{n}{3}) + log_3(n^2)$.

b) $T(n) = 2T\left(\frac{n}{3}\right) + \log_3(n^2)$ $\quad n = 3^m \quad m = \log_3(n) \quad T(g(m)) = Y(m)$

$T(n) = 2T\left(\frac{n}{3}\right) + 2\log_3(n)$

$T(3^m) = 2T\left(\frac{3^m}{3}\right) + 2m$

$T(3^m) = 2T(3^{m-1}) + 2m$

$T(g(m)) = 2T(g(m-1)) + 2m$

$\{Y(m) = 2Y(m-1) + 2m\}$ // $Y(m) - 2Y(m-1) = 2m$

Homogénea: $Y(m) - 2Y(m-1) = 0$

$\lambda - 2 = 0$ // $\lambda = 2$ // $\{$forma: $h(m) = C_0 2^m\}$

Particular: $Y(m) - 2Y(m-1) = 2m$ forma $C_1 n + C_2$

$P(m) = 2mC_1 + C_2$

$(2mC_1 + C_2) - 2(2(m-1)C_1 + C_2) = 2m$

$(2mC_1 + C_2) - 2((2m-2)C_1 + C_2) = 2m$

$(2mC_1 + C_2) - 4mC_1 + 4C_1 - 2C_2 = 2m$

$(2mC_1 - 4mC_1) + (C_2 - 2C_2) + 4C_1 = 2m$

$-2mC_1 + (-C_2 + 4C_1) = 2m$ // $C_1(4 - 2m) - C_2 = 2m$ //

$C - 2mC_1 = 2m$ // $C_1 = -1$ // $-C_2 + 4(-1) = 0$ //

$\hookrightarrow -C_2 - 4 = 0$ // $C_2 = -4$ // $\{P(m) = -2m - 4\}$

Completa: $Y(m) = C_0 2^m - 2m - 4$

$\Longrightarrow Y(m) = C_0 2^{\log_3(n)} - 2\log_3(n) - 4$

Complejidad: $O(2^{\log_3(n)})$

(c) (14 points) $T(n) = 4T(\frac{n}{2}) + n^2$.

c) $T(n) = 4T\left(\frac{n}{2}\right) + n^2$ $\quad n = 2^m \quad m = \log_2(n) \quad T(g(m)) = Y(m)$

$T(2^m) = 4T\left(\frac{2^m}{2}\right) + (2^m)^2$

$T(2^m) = 4T(2^{m-1}) + 2^{2m}$

$T(g(m)) = 4T(g(m-1)) + 2^{2m}$

$Y(m) = 4Y(m-1) + 2^{2m}$ // $Y(m) - 4Y(m-1) = 2^{2m}$

Homogénea: $Y(m) - 4Y(m-1) = 0$

$\lambda - 4 = 0$ // $\lambda = 4$ // $h(m) = C_1 4^m$

Particular: $Y(m) - 4Y(m-1) = 2^{2m}$ // $P(m) = C_2 2^{2m}$

$(C_2 2^{2m}) - 4(C_2 2^{2m-1}) = 2^{2m}$

$C_2 2^{2m} - \frac{4C_2 2^{2m}}{2} = 2^{2m}$ // $C_2 2^{2m} - 2C_2 2^{2m} = 2^{2m}$

$C_2(2^{2m} - 2 \cdot 2^{2m}) = 2^{2m}$ // $C_2 = 2^{2m} / (2^{2m} - 2^{2m+1})$

$\hookrightarrow C_2 = 2^{2m} / (-2^{2m})$ // $C_2 = -1$ // $P(m) = -2^{2m}$

Completa: $Y(m) = C_1 4^m - 2^{2m}$

$Y(\log_2(n)) = C_1 4^{\log_2(n)} - 2^{2\log_2(n)}$

Complejidad: $O(4^{\log_2(n)})$

Punto 2:

a) Genovia:

1) (archivo)

2) El programa utiliza la estrategia de Dividir y Conquistar (DC) para encontrar un evento cumbre en una lista de magnitudes sísmicas ya que divide el conjunto de datos en dos mitades en cada paso, lo cual hace que el programa reduzca el espacio de búsqueda de manera significativa, permitiendo una búsqueda más rápida y eficiente que si se examinara cada elemento secuencialmente.
Esta estrategia aprovecha la naturaleza recursiva del problema, donde la condición para ser un evento cumbre (ser mayor que los elementos adyacentes) se puede verificar de manera independiente en cualquier subsegmento del conjunto de datos, al aplicar la división en cada iteración, el programa converge rápidamente hacia la solución, encontrando un evento cumbre sin necesidad de revisar todos los elementos, lo cual sería más lento. Este enfoque no solo mejora la eficiencia sino que también simplifica el proceso de identificación del evento cumbre al enfocarse en subconjuntos más pequeños del problema original.

```
def encontrar_evento_cumbre(eventos, inicio, fin):
    if fin - inicio == 2: # n * (C2 + C3)
        if eventos[inicio] < eventos[inicio + 1] > eventos[fin]:
            return eventos[inicio + 1]
        return None
    if fin - inicio < 2:
        return None

    medio = inicio + (fin - inicio) // 2
    if eventos[medio] > eventos[medio - 1] and eventos[medio] >
eventos[medio + 1]:
        return eventos[medio]
    if eventos[medio - 1] > eventos[medio]:
        return encontrar_evento_cumbre(eventos, inicio, medio)

    return encontrar_evento_cumbre(eventos, medio, fin)
```

3) Por dividir y conquistar (se divide la lista de eventos), la ecuación de recurrencia sería:
$T(n) = T(n/2) + C$

4) La complejidad de la función sería de $O(\log(n))$

b) Melquiades:

1) (archivo)

2) Para encontrar la distancia mínima entre puntos, se divide el conjunto total de puntos en dos mitades, basándose en sus coordenadas X, generando así dos subproblemas más pequeños. Este proceso de división continúa recursivamente hasta alcanzar el caso base, que ocurre cuando el conjunto de puntos es reducido a tres o menos, permitiendo calcular la distancia mínima directamente mediante comparaciones simples.

Durante la fase de "conquistar", el algoritmo se concentra en encontrar la distancia mínima entre puntos que podrían estar en diferentes mitades pero cerca de la línea divisoria. Esto se logra examinando puntos dentro de una banda estrecha centrada en esta línea, donde solo se comparan puntos que están dentro de una distancia específica, optimizando así el proceso.

```
def distancia(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

def distancia_minima_div_conq(puntos):
    def distancia_min_rec(puntos_ordenados_x, puntos_ordenados_y):
        if len(puntos_ordenados_x) <= 3:
                            return  min(distancia(puntos_ordenados_x[i],
puntos_ordenados_x[j]) for i in range(len(puntos_ordenados_x)) for j in
range(i + 1, len(puntos_ordenados_x)))

        mitad = len(puntos_ordenados_x) // 2
        Qx = puntos_ordenados_x[:mitad]
        Rx = puntos_ordenados_x[mitad:]

        punto_medio = puntos_ordenados_x[mitad][0]
                  Qy  =  list(filter(lambda  p:  p[0]  <=  punto_medio,
puntos_ordenados_y))
                   Ry  =  list(filter(lambda  p:  p[0]  >  punto_medio,
puntos_ordenados_y))

        delta_q = distancia_min_rec(Qx, Qy)
        delta_r = distancia_min_rec(Rx, Ry)
        delta = min(delta_q, delta_r)

         banda = [p for p in puntos_ordenados_y if punto_medio - delta <
p[0] < punto_medio + delta]
        min_dist_banda = delta
        for i in range(len(banda)):
            for j in range(i+1, min(i + 7, len(banda))):
```

```
                d = distancia(banda[i], banda[j])
                if d < min_dist_banda:
                    min_dist_banda = d

        return min_dist_banda

    puntos_ordenados_x = sorted(puntos, key=lambda p: p[0])
    puntos_ordenados_y = sorted(puntos, key=lambda p: p[1])
    return distancia_min_rec(puntos_ordenados_x, puntos_ordenados_y)
```

3) T(n) es el tiempo necesario para resolver el problema para n puntos.
El término 2T(n/2) representa el coste de resolver recursivamente cada una de las dos mitades del conjunto de puntos. Esto indica que el problema se divide en dos subproblemas de tamaño n/2, cada uno resuelto de manera independiente.

El término O(n) representa el coste de combinar las soluciones de los subproblemas, incluida la búsqueda de la distancia mínima cruzada entre las dos mitades. Este paso implica ordenar los puntos según sus coordenadas Y y luego buscar la distancia mínima cruzada, lo cual se puede hacer en tiempo lineal respecto al número de puntos en la banda cercana a la línea divisoria

T(n) = 2T(2n)+O(n)

4) La complejidad entonces sería O(n log(n))