Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2283 - Diseño y Análisis de Algoritmos - 2$^{\text{do}}$ semestre 2022

# PAUTA INTERROGACIÓN 1

## Pregunta 1

Para esta pregunta asuma que las funciones son de tipo $f : \mathbb{N} \to \mathbb{R}_0^+$. Responda lo siguiente:

1. Demuestre que si $f_1 \in \mathcal{O}(g_1)$ y $f_2 \in \mathcal{O}(g_2)$ luego $f_1 + f_2 \in \mathcal{O}(\max\{g_1(n), g_2(n)\})$.
2. Se define:
$$\omega(f) = \{g : \mathbb{N} \to \mathbb{R}_0^+ \mid \forall c \in \mathbb{R}^+.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ g(n) \geq c \cdot f(n)\}$$
Demuestre que $3^n \in \omega(2^n)$.

**Solución.** A continuación se muestra una posible demostración para cada inciso:

1. Suponga que $f_1 \in \mathcal{O}(g_1)$ y $f_2 \in \mathcal{O}(g_2)$. Luego se cumple
$$\exists c_1 \in \mathbb{R}^+.\ \exists n_0^1 \in \mathbb{N}.\ \forall n \geq n_0^1.\ f_1(n) \leq c_1 \cdot g_1(n)$$
$$\exists c_2 \in \mathbb{R}^+.\ \exists n_0^2 \in \mathbb{N}.\ \forall n \geq n_0^2.\ f_2(n) \leq c_2 \cdot g_2(n)$$
Si sumamos las desigualdades:
$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) \\ &\leq c_1 \cdot \max\{g_1(n), g_2(n)\} + c_2 \cdot \max\{g_1(n), g_2(n)\} \\ &= (c_1 + c_2) \cdot \max\{g_1(n), g_2(n)\} \end{aligned}$$
por lo que con $c = c_1 + c_2$ y $n_0 = \max\{n_0^1, n_0^2\}$ se demuestra lo pedido.
2. Se debe demostrar que
$$\forall c \in \mathbb{R}^+.\ \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ 3^n \geq c \cdot 2^n$$
Basta entonces con encontrar $n_0$ para cada $c > 0$. Si despejamos la desigualdad:
$$\begin{aligned} 3^n &\geq c \cdot 2^n \\ \left(\frac{3}{2}\right)^n &\geq c \\ n &\geq \log_{3/2}(c) = \frac{\log_2(c)}{\log_2(3) - 1} \end{aligned}$$
con $n_0 = \lceil \log_{3/2}(c) \rceil$ es fácil ver que se cumple lo pedido.

**Rúbrica.** Dado lo anterior la atribución de puntaje es la siguiente:

- (**1 Punto**) En ítem 1. Por sumar las cotas de ambas funciones y llegar a una desigualdad.
- (**1 Punto**) En ítem 1. Por acotar usando el máximo entre $g_1(n)$ y $g_2(n)$.
- (**1 Punto**) En ítem 1. Por indicar que se encontró un $c$ y un $n_0$ tal que se demuestra lo pedido.
- (**1 Punto**) En ítem 2. Por formular la demostración e indicar que se busca un $n_0$ para cada valor de $c$.
- (**1 Punto**) En ítem 2. Por despejar la desigualdad usando el logaritmo.
- (**1 Punto**) En ítem 2. Por encontrar el valor de $n_0$ que satisface la definición.

# Pregunta 2

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 1 \\ T\left(\left\lfloor \frac{n\sqrt{3}}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2 & n > 1 \end{cases}$$

Demuestre que $T(n) \in \mathcal{O}(n^2 \cdot \log_2(n))$ usando inducción constructiva.

**Solución.** Iniciamos la inducción constructiva a partir del caso inductivo. Se asume que existen $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tal que para todo $n_0 \leq n' < n$ se cumple que $T(n') \leq c \cdot n'^2 \cdot \log_2(n')$.

$$\begin{aligned}
T(n) &= T\left(\left\lfloor \frac{n\sqrt{3}}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2 \\
&\leq c \cdot \left\lfloor \frac{n\sqrt{3}}{2} \right\rfloor^2 \log_2\left(\left\lfloor \frac{n\sqrt{3}}{2} \right\rfloor\right) + c \cdot \left\lfloor \frac{n}{2} \right\rfloor^2 \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n^2 \qquad \text{(HI)} \\
&\leq c \cdot \left(\frac{n\sqrt{3}}{2}\right)^2 \log_2\left(\frac{n\sqrt{3}}{2}\right) + c \cdot \left(\frac{n}{2}\right)^2 \log_2\left(\frac{n}{2}\right) + n^2 \\
&= c \cdot \frac{3n^2}{4}\left(\log_2(n) + \log_2\left(\sqrt{3}\right) - \log_2(2)\right) + c \cdot \frac{n^2}{4}\left(\log_2(n) - \log_2(2)\right) + n^2 \\
&= c \cdot \frac{3n^2}{4}\left(\log_2(n) + \frac{1}{2}\log_2(3) - 1\right) + c \cdot \frac{n^2}{4}(\log_2(n) - 1) + n^2 \\
&= c \cdot n^2 \log_2(n) + \left(c\left(\frac{3}{8}\log_2(3) - 1\right) + 1\right) \cdot n^2
\end{aligned}$$

Para lograr el paso inductivo entonces, necesitamos que

$$\begin{aligned}
c\left(\frac{3}{8}\log_2(3) - 1\right) + 1 &\leq 0 \\
c\left(1 - \frac{3}{8}\log_2(3)\right) &\geq 1 \\
c \cdot (8 - 3\log_2(3)) &\geq 8 \\
c &\geq \frac{8}{8 - 3\log_2(3)}
\end{aligned}$$

Así, podríamos elegir $c$ igual a la fracción encontrada. Sin embargo, nótese que

$$\frac{8}{8 - 3\log_2(3)} \leq \frac{8}{8 - 3\log_2(4)} = \frac{8}{8 - 3 \cdot 2} = \frac{8}{8 - 6} = \frac{8}{2} = 4$$

Por lo que eligiendo $c = 4$ se cumplirá el paso inductivo.

Ahora queda verificar los casos base. Claramente $n = 1$ no será cierto ya que $\log_2(1) = 0$. Así los casos base serán $n = 2$ y $n = 3$:

$$\begin{aligned}
T(2) &= T(1) + T(1) + 2^2 = 1 + 1 + 4 = 6 \leq 4 \cdot 2^2 \cdot \log_2(2) = 4 \cdot 4 \cdot 1 = 16 \\
T(3) &= T(2) + T(1) + 3^2 = 6 + 1 + 9 = 16 \leq 4 \cdot 3^2 \cdot \log_2(3) = 36\log_2(3)
\end{aligned}$$

donde la última desigualdad es cierta dado que $\log_2(3) > 1$. No hay más casos base ya que $T(4) = T(3) + T(2)$ ya puede formar parte del paso inductivo. De esta manera eligiendo $n_0 = 2$ será suficiente. Así entonces se demuestra que para todo $n \geq 2$ se cumple que

$$T(n) \leq 4 \cdot n^2 \log_2(n)$$

y entonces $T(n) \in \mathcal{O}(n^2 \cdot \log_2(n))$.

**Rúbrica.** Dado lo anterior la atribución de puntaje es la siguiente:

(**0.5 Puntos**) Por enunciar correctamente la inducción constructiva.

(**3 Puntos**) Por desarrollar correctamente el paso inductivo.

(**1 Punto**) Por encontrar correctamente un $c$ que permita realizar el paso inductivo.

(**1.5 Puntos**) Por completar todos los casos base que corresponda correctamente.

# Pregunta 3

Sea $\Sigma$ un alfabeto. Para una palabra $w \in \Sigma^*$ se tiene que una partición palindrómica de $w$ es una secuencia de palabras $u_1, \ldots, u_\ell \in \Sigma^*$ tal que $u_1 \cdots u_\ell = w$ y $u_i$ es un palíndromo para cada $1 \leq i \leq \ell$ (recuerde que un palíndromo es una palabra que es igual a su reverso).

En esta pregunta, queremos utilizar programación dinámica para construir un algoritmo **MinPart** que reciba una palabra $w = a_1 \cdots a_n$ y retorne el largo mínimo $\ell$ posible para una partición palindrómica de $w$. Para este objetivo utilizamos la siguiente definición recursiva de $\textbf{MinPart}(a_1 \cdots a_n)$:

$$\textbf{MinPart}(a_1 \cdots a_n) = \begin{cases} 1 & a_1 \cdots a_n \text{ es un palíndromo} \\ \min\limits_{1 \leq k < n} \{\textbf{MinPart}(a_1 \cdots a_k) & \\ \qquad + \textbf{MinPart}(a_{k+1} \cdots a_n)\} & a_1 \cdots a_n \text{ NO es un palíndromo} \end{cases}$$

En esta pregunta usted debe explicar por qué se puede aplicar programación dinámica para resolver este problema, en particular, por qué la recursión anterior es correcta. Además, debe implementar un algoritmo que utilice esta recursión para calcular **MinPart** en tiempo polinomial, suponiendo que la operación básica a contar es la comparación entre símbolos pertenecientes a $\Sigma$.

**Solución.** En esta pregunta se debe explicar primero la razón de porqué la recursión es correcta. Se podía realizar una demostración usando inducción. Sin embargo bastaba explicando con palabras la idea de que la solución óptima (el tamaño de la partición palindrómica más corta) debe provenir de alguna solución óptima tomada a partir de dividir la palabra en subpalabras. Esto ya que las particiones palindrómicas de ellas se pueden unir para obtener una partición palindrómica de la palabra original (y por lo tanto, basta con sumar el tamaño de ambas particiones palindrómicas). Así la recursión descrita elige el mínimo de todas las posibles divisiones de la palabra en dos sub-palabras.

Luego la razón de porqué se puede usar programación dinámica viene dada por este principio de optimalidad de los subproblemas (las sub particiones palindrómicas de la palabra original).

Por último, se debe dar una implementación que calcule **MinPart** en tiempo polinomial. Acá se pueden utilizar dos ideas: hacer un enfoque *top-down* recursivo y usar un diccionario para guardar las llamadas recursivas repetidas. O bien, usar un enfoque *bottom-up* o de memoización. A continuación se muestra una posible implementación según este último enfoque:

```
MinPart(a_1 ··· a_n)
    L := matriz de n × n con ceros en sus entradas
    P := matriz de n × n con ceros en sus entradas
    for i := 1 to n do
        L_{i,i} := 1
        P_{i,i} := 1
        if (i ≠ n) then
            if (a_i = a_{i+1}) then
                P_{i,i+1} := 1
                L_{i,i+1} := 1
            else
                L_{i,i+1} := 2
    for m := 2 to n − 1 do
        for i := 1 to n − m do
            j := i + m
            if (a_i = a_j ∧ P_{i+1,j−1} = 1) then
                P_{i,j} := 1
                L_{i,j} := 1
            else
                P_{i,j} := 0
                for k := i to j − 1 do
                    L_{i,j} := min{L_{i,j}, L_{i,k} + L_{k+1,j}}
    return L_{1,n}
```

La idea es que $L_{i,j}$ almacena **MinPart**$(a_i \cdots a_j)$ y $P_{i,j}$ almacena si $a_i \cdots a_j$ es un palíndromo o no. Se parte revisando todas las sub-palabras de tamaño menor o igual a 2, marcando si son palíndromos o no. Posteriormente se revisan, de forma iterativa, todas las sub-palabras de tamaño mayor a 2, utilizando los resultados guardados en la matriz, hasta llegar finalmente a la palabra completa. Esto es posible gracias a que un palíndromo de tamaño $n$ (con $n > 2$) se construye siempre a partir de un palíndromo de tamaño $n-2$ (agregando un mismo elemento a ambos extremos). Es fácil ver que el tiempo de ejecución del algoritmo será $\mathcal{O}(n^3)$.

Se acepta también una respuesta que asuma la disponibilidad de una función **EsPalindromo**$(a_i \cdots a_j)$ que decida en tiempo polinomial si acaso $a_i \cdots a_j$ es un palíndromo o no.

**Rúbrica.** Dado lo anterior la atribución de puntaje es la siguiente:

- (**2 Puntos**) Por explicar por qué la recursión es correcta.
- (**1 Punto**) Por explicar por qué se puede aplicar programación dinámica en este caso.
- (**3 Puntos**) Por hacer una implementación de tiempo polinomial.

# Pregunta 4

Considere un polinomio $p(x) = \sum_{k=0}^{n-1} a_k x^k$ no nulo de coeficientes racionales con $n \geq 1$. En clases vimos que si $n$ es par entonces podemos separar $p(x)$ en dos polinomios de grado $n/2 - 1$:

$$p(x) = \sum_{k=0}^{\frac{n}{2}-1} a_{2k} x^{2k} + x \cdot \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1} x^{2k} = q(x^2) + x \cdot r(x^2)$$

Lo que permite confeccionar el algoritmo **FFT** que realiza 2 llamadas recursivas sobre los polinomios $q(x)$ y $r(x)$. Dicho eso, no es difícil ver que si $n$ es divisible por 3 se va a cumplir que:

$$p(x) = \sum_{k=0}^{\frac{n}{3}-1} a_{3k} x^{3k} + x \cdot \sum_{k=0}^{\frac{n}{3}-1} a_{3k+1} x^{3k} + x^2 \cdot \sum_{k=0}^{\frac{n}{3}-1} a_{3k+2} x^{3k} = q(x^3) + x \cdot r(x^3) + x^2 \cdot s(x^3)$$

Es decir, podemos separar $p(x)$ en 3 polinomios de grado $n/3 - 1$.

Usando esta idea, explique como se podría modificar el algoritmo **FFT** de manera que se realicen 3 llamadas recursivas y se siga entregando correctamente $\mathbf{DFT}(a_0, \ldots, a_{n-1})$ en tiempo $\mathcal{O}(n \cdot \log n)$. Justifique su respuesta.

**Solución.** Primero, así como en el algoritmo **FFT** se asume que el tamaño del input es una potencia de 2, ahora asumimos que el tamaño $n$ del input es una potencia de 3. Tenemos entonces que para $k \in \{0, \ldots, n-1\}$ se cumple que

$$p(\omega_n^k) = q\left(\left(\omega_n^k\right)^3\right) + \omega_n^k \cdot r\left(\left(\omega_n^k\right)^3\right) + \left(\omega_n^k\right)^2 \cdot s\left(\left(\omega_n^k\right)^3\right)$$

Para que el algoritmo modificado funcione (llamémoslo $\mathbf{FFT}_3$) se debería cumplir que los términos $\left(\omega_n^k\right)^3$ correspondan a las $\frac{n}{3}$-raíces de la unidad. Esto es fácil de ver gracias a la regla de simplificación vista en clases:

$$\left(\omega_n^k\right)^3 = \omega_{3 \cdot n/3}^{3 \cdot k} = \omega_{n/3}^k$$

Además también ocurre que al elevar al cubo las raíces comienzan a repetirse. En **FFT** se repetían dos veces, y en $\mathbf{FFT}_3$ se repetirán tres veces. Sea $k \in \{0, \ldots, \frac{n}{3} - 1\}$, luego

$$\left(\omega_n^{n/3+k}\right)^3 = \exp\left(3 \cdot \frac{2\pi i}{n}\left(\frac{n}{3} + k\right)\right) = e^{2\pi i} \cdot \left(\omega_n^k\right)^3 = \omega_{n/3}^k$$
$$\left(\omega_n^{2n/3+k}\right)^3 = \exp\left(3 \cdot \frac{2\pi i}{n}\left(\frac{2n}{3} + k\right)\right) = e^{4\pi i} \cdot \left(\omega_n^k\right)^3 = \omega_{n/3}^k$$

Lo que nos falta entonces es asegurarnos de que podemos obtener los $y_k = p(\omega_n^k)$ combinando los $u_k = q(\omega_{n/3}^k)$, $v_k = r(\omega_{n/3}^k)$ y $w_k = s(\omega_{n/3}^k)$ en tiempo lineal. Ya que así el Teorema Maestro nos asegura que el nuevo algoritmo se ejecutará en tiempo $\mathcal{O}(n \cdot \log(n))$. Desarrollar los $y_k$ es directo para $k \in \{0, \ldots, \frac{n}{3} - 1\}$:

$$\begin{aligned} y_k = p(\omega_n^k) &= q\left(\left(\omega_n^k\right)^3\right) + \omega_n^k \cdot r\left(\left(\omega_n^k\right)^3\right) + \left(\omega_n^k\right)^2 \cdot s\left(\left(\omega_n^k\right)^3\right) \\ &= q\left(\omega_{n/3}^k\right) + \omega_n^k \cdot r\left(\omega_{n/3}^k\right) + \omega_n^{2k} \cdot s\left(\omega_{n/3}^k\right) \end{aligned}$$

$$\begin{aligned} y_{n/3+k} = p(\omega_n^{n/3+k}) &= q\left(\left(\omega_n^{n/3+k}\right)^3\right) + \omega_n^{n/3+k} \cdot r\left(\left(\omega_n^{n/3+k}\right)^3\right) + \left(\omega_n^{n/3+k}\right)^2 \cdot s\left(\left(\omega_n^{n/3+k}\right)^3\right) \\ &= q\left(\omega_{n/3}^k\right) + \omega_n^{n/3} \cdot \omega_n^k \cdot r\left(\omega_{n/3}^k\right) + \omega_n^{2n/3} \cdot \omega_n^{2k} \cdot s\left(\omega_{n/3}^k\right) \\ &= q\left(\omega_{n/3}^k\right) + e^{2\pi i/3} \cdot \omega_n^k \cdot r\left(\omega_{n/3}^k\right) + e^{4\pi i/3} \cdot \omega_n^{2k} \cdot s\left(\omega_{n/3}^k\right) \end{aligned}$$

$$\begin{aligned} y_{2n/3+k} = p(\omega_n^{2n/3+k}) &= q\left(\left(\omega_n^{2n/3+k}\right)^3\right) + \omega_n^{2n/3+k} \cdot r\left(\left(\omega_n^{2n/3+k}\right)^3\right) + \left(\omega_n^{2n/3+k}\right)^2 \cdot s\left(\left(\omega_n^{2n/3+k}\right)^3\right) \\ &= q\left(\omega_{n/3}^k\right) + \omega_n^{2n/3} \cdot \omega_n^k \cdot r\left(\omega_{n/3}^k\right) + \omega_n^{4n/3} \cdot \omega_n^{2k} \cdot s\left(\omega_{n/3}^k\right) \\ &= q\left(\omega_{n/3}^k\right) + e^{4\pi i/3} \cdot \omega_n^k \cdot r\left(\omega_{n/3}^k\right) + e^{2\pi i/3} \cdot \omega_n^{2k} \cdot s\left(\omega_{n/3}^k\right) \end{aligned}$$

Así lo que nos queda es:

$$\begin{aligned} y_k &= u_k + \omega_n^k \cdot v_k + \omega_n^{2k} \cdot w_k \\ y_{n/3+k} &= u_k + e^{2\pi i/3} \cdot \omega_n^k \cdot v_k + e^{4\pi i/3} \cdot \omega_n^{2k} \cdot w_k \\ y_{2n/3+k} &= u_k + e^{4\pi i/3} \cdot \omega_n^k \cdot v_k + e^{2\pi i/3} \cdot \omega_n^{2k} \cdot w_k \end{aligned}$$

Por lo que el algoritmo modificado quedará así

**FFT**$_3(a_0, \ldots, a_{n-1})$
  **if** $n = 3$ **then**
    $y_0 = a_0 + a_1 + a_2$
    $y_1 = a_0 + e^{2\pi i/3} \cdot a_1 + e^{4\pi i/3} \cdot a_3$
    $y_2 = a_0 + e^{4\pi i/3} \cdot a_1 + e^{2\pi i/3} \cdot a_3$
    **return** $[y_0, y_1, y_2]$
  **else**
    $[u_0, \ldots, u_{\frac{n}{3}-1}] :=$ **FFT**$(a_0, \ldots, a_{n-3})$
    $[v_0, \ldots, v_{\frac{n}{3}-1}] :=$ **FFT**$(a_1, \ldots, a_{n-2})$
    $[w_0, \ldots, w_{\frac{n}{3}-1}] :=$ **FFT**$(a_2, \ldots, a_{n-1})$
    $\omega_n := e^{\frac{2\pi i}{n}}$
    $\omega_n^2 := \omega_n \cdot \omega_n$
    $\alpha := 1$
    $\beta := 1$
    **for** $k := 0$ **to** $\frac{n}{3} - 1$ **do**
      $y_k := u_k + \alpha \cdot v_k + \beta \cdot w_k$
      $y_{n/3+k} := u_k + e^{2\pi i/3} \cdot \alpha \cdot v_k + e^{4\pi i/3} \cdot \beta \cdot w_k$
      $y_{2n/3+k} := u_k + e^{4\pi i/3} \cdot \alpha \cdot v_k + e^{2\pi i/3} \cdot \beta \cdot w_k$
      $\alpha := \alpha \cdot \omega_n$
      $\beta := \beta \cdot \omega_n^2$
    **return** $[y_0, \ldots, y_{n-1}]$

**Rúbrica.** Dado lo anterior la atribución de puntaje es la siguiente:

(**2.5 Puntos**) Por argumentar que la recursión es correcta llamando a los polinomios $q$, $r$ y $s$ sobre las $\frac{n}{3}$-raíces de la unidad.

(**3 Puntos**) Por mostrar como combinar las **DFT** más pequeñas para obtener la más grande.

(**0.5 Puntos**) Por argumentar por qué el algoritmo es de tiempo $\mathcal{O}(n \cdot \log(n))$.

Se asignan (**2 ptos**) totales si sólo muestra un algoritmo correcto que haga lo pedido

Se añade (**1 pto**) al puntaje actual si analiza los casos base del algoritmo