Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2283 - Diseño y Análisis de Algoritmos

# Ayudantía 10 - Repaso I2

Septiembre de 2025

Caetano Borges, Isabella Cherubini

---

## 1. Greedy

Tenemos $n$ charlas por asignar. La charla $i$ tiene hora de inicio $s_i$ y hora de término $t_i$, es decir, ocupa el intervalo $[s_i, t_i)$. Debemos ofrecer charlas de tal manera que en ningún momento haya dos charlas ocurriendo simultáneamente. Queremos maximizar la cantidad de charlas ofrecidas.

1. Diseñe un algoritmo codicioso que resuelva este problema eficientemente. Escriba pseudocódigo.
2. Demuestre la correctitud de su algoritmo.

**Solución**

**1. Algoritmo Codicioso**

**Idea principal:** Seleccionar siempre la charla que termina más temprano entre las disponibles.

**Algorithm 1** *

**Entrada:** lista de $n$ charlas con tiempos $(s_i, t_i)$.
**Salida:** conjunto $A$ de charlas compatibles de máxima cardinalidad.

```
Ordenar las charlas en orden no decreciente de sus tiempos de término: t_(1) ≤ ··· ≤ t_(n)
A ← ∅
u ← −∞                          ▷ marca el tiempo de término de la última charla elegida
for j = 1 to n do
    if s_(j) ≥ u then
        A ← A ∪ {(j)}
        u ← t_(j)
    end if
end for
return A
```

Aquí $(j)$ denota el índice de la $j$-ésima charla en la ordenación por tiempos de término.

**Complejidad:** Ordenar las charlas cuesta $O(n \log n)$. El recorrido posterior cuesta $O(n)$, por lo que la complejidad total es $O(n \log n)$.

**2. Demostración de correctitud**

**Teorema 1.** *Sea $G = \{g_1, g_2, \ldots, g_k\}$ el conjunto de actividades elegidas por el algoritmo que ordena por tiempo de término creciente y selecciona las compatibles en ese orden. Sea $O = \{o_1, o_2, \ldots, o_m\}$ cualquier solución óptima. Entonces $k \geq m$, es decir, el algoritmo greedy es óptimo.*

*Demostración.* Dado un intervalo $I$ cualquiera, usamos $f(I)$ para denotar el extremo derecho del mismo. Mostraremos que para todo $i \leq \text{mín}\{k, m\}$ se cumple

$$f(g_i) \leq f(o_i)$$

es decir, el algoritmo greedy nunca "se queda atrás" respecto a una solución óptima. Lo demostramos por inducción sobre $i$.

**Caso base $i = 1$:** $g_1$ es la actividad que termina más temprano entre todas, por lo tanto $f(g_1) \leq f(o_1)$.

**Paso inductivo:** Supongamos que $f(g_j) \leq f(o_j)$ para todo $j \leq t$. Como $o_{t+1}$ empieza luego de $f(o_t)$ y $f(o_t) \geq f(g_t)$, entonces $o_{t+1}$ también empieza no antes de $f(g_t)$. Por construcción, $g_{t+1}$ es la actividad compatible con $g_t$ que tiene el menor tiempo de término, de modo que

$$f(g_{t+1}) \leq f(o_{t+1})$$

Por inducción, $f(g_i) \leq f(o_i)$ para todo $i \leq \text{mín}\{k, m\}$. Si $k < m$, entonces $o_{k+1}$ empieza después de $f(o_k) \geq f(g_k)$ y sería compatible con todas las actividades en $G$, contradiciendo que $G$ es maximal. Por lo tanto, $k \geq m$ y el algoritmo es óptimo. □

# 2. FFT

Sean $a$ y $b$ dos números enteros positivos de $n$ dígitos cada uno (asuma que $n$ es una potencia de 2). Diseñe un algoritmo que permita calcular la multiplicación $a \times b$ usando $O(n \lg n)$ multiplicaciones y sumas de números complejos. Puede usar una descripción con palabras, siempre y cuando sea formal y clara. Se aconseja usar ejemplos para ilustrar su explicación. Indique claramente qué argumentos usará para invocar al algoritmo FFT visto en clases, y qué recibe como respuesta al aplicar la transformada inversa.

**Solución**

Sean $a = a_{n-1}a_{n-2}\cdots a_0$ los dígitos de $a$ y $b = b_{n-1}b_{n-2}\cdots b_0$ los dígitos de $b$. La multiplicación $a \times b$ puede calcularse de la siguiente manera: 1. Definimos los siguientes polinomios asociados a esos enteros:

$$p_a(x) = \sum_{i=0}^{n-1} a_i x^i,$$

y

$$p_b(x) = \sum_{i=0}^{n-1} b_i x^i$$

Ambos polinomios son representados de forma canónica usando los vectores $p_a(x) = (a_0, a_1, \ldots, a_{n-1})$ y $p_b(x) = (b_0, b_1, \ldots, b_{n-1})$. Por ejemplo, si $a = 8432$ y $b = 1054$, los polinomios son $p_a(x) = (2, 3, 4, 8)$ y $p_b(x) = (4, 5, 0, 1)$. 2. La multiplicación $a \cdot b$ se calculará usando $p_a(x) \cdot p_b(x)$ mediante la transformada rápida de Fourier (y su inversa), obteniendo el vector $c = (c_0, c_1, \ldots, c_{2n-1})$ que representa la multiplicación $p_a(x) \cdot p_b(x)$ usando $O(n \lg n)$ multiplicaciones y sumas de números complejos. Para el ejemplo anterior, el vector resultante es $p_a(x) \cdot p_b(x) = 8x^4 + 4x^5 + 43x^4 + 54x^3 + 31x^2 + 22x^1 + 8x^0$, representado por el vector $c = (8, 22, 31, 54, 43, 4, 8)$.

3. La última etapa consiste en la normalización del vector obtenido como resultado en el punto anterior, ya que éste todavía no puede ser interpretado como un número entero. Esto es porque las componentes del vector no necesariamente son dígitos entre 0 y 9. En nuestro ejemplo tenemos $22x^1$, que no corresponde a un dígito válido (tampoco $43x^4, 54x^3$, ni $31x^2$ ). Para solucionar esto, basta con recorrer el vector resultante, en cada posición quedándonos con el último dígito de cada coeficiente, y acarreando el restante a la siguiente posición (para ser sumado). El algoritmo es el siguiente:

```
1: r ← 0
2: for i ← 1 to 2n − 1 do
3:     c[i] ← c[i] + r
4:     r ← ⌊c[i]/10⌋
5:     c[i] ← c[i] mod 10
6: end for
```

Esto toma tiempo $O(n)$. Aplicando este algoritmo sobre el vector $c = (8, 22, 31, 54, 43, 4, 8)$ se obtiene $(8, 2, 3, 7, 8, 8, 8)$, que representa el entero $8887328 = 8432 \times 1054$.