Ignacio Méndez Pérez
IIC2283 - 2022 - 2

Programación dinámica + FFT

Sea $\mathcal{A}$ un algoritmo. Asociamos a $\mathcal{A}$ una **función de tiempo de ejecución**

$$\text{tiempo}_{\mathcal{A}}: \Sigma^* \to \mathbb{N}$$

tal que:

$$\text{tiempo}_{\mathcal{A}}(w) \coloneqq \text{ número de pasos realizados por } \mathcal{A} \text{ con entrada } w \in \Sigma^*$$

Definición:

Sea $f: \mathbb{N} \to \mathbb{R}_0^+$ una función. Se define el **conjunto $\mathcal{O}(f)$** tal que:

$$\boxed{\mathcal{O}(f) = \{\, g: \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ g(n) \leq c \cdot f(n) \,\}}$$

Decimos entonces que $g \in \mathcal{O}(f)$.

- También usamos la notación g es $\mathcal{O}(f)$, lo cual es formalizado como $g \in \mathcal{O}(f)$.

Definición:

Sea $f: \mathbb{N} \to \mathbb{R}_0^+$ una función. Se definen los **conjuntos $\Omega(f)$ y $\Theta(f)$** tal que:

$$\begin{aligned} \Omega(f) &= \{\, g: \mathbb{N} \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}^+. \exists n_0 \in \mathbb{N}.\ \forall n \geq n_0.\ c \cdot f(n) \leq g(n) \,\} \\ \Theta(f) &= \mathcal{O}(f) \cap \Omega(f) \end{aligned}$$

Búsqueda binaria:

```
BúsquedaBinaria(a, L, i, j)
    if i > j then return no
    else if i = j then
        if L[i] = a then return i
        else return no
    else
        p := ⌊(i+j)/2⌋
        if L[p] < a then return BúsquedaBinaria(a, L, p + 1, j)
        else if L[p] > a then return BúsquedaBinaria(a, L, i, p − 1)
        else return p
```

Llamada inicial al algoritmo: **BúsquedaBinaria**$(a, L, 1, n)$

Si contamos solo las comparaciones, entonces la complejidad del algoritmo se define como:

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

donde $c \in \mathbb{N}$ y $d \in \mathbb{N}$ son constantes tales que $c \geq 1$ y $d \geq 1$.
Esta es una **ejecución de recurrencia**.

Solucionando la ecuación:

Técnica básicaL sustitución de variables.
Para la ecuación anterior usamos la sustitución $n = 2^k$.

- Suponemos que n es potencia de 2.
- Utilizaremos inducción.

$$T(2^k) = \begin{cases} c & k = 0 \\ T(2^{k-1}) + d & k > 0 \end{cases}$$

Expandimos:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + d \\ &= (T(2^{k-2}) + d) + d \\ &= T(2^{k-2}) + 2d \\ &= (T(2^{k-3}) + d)2d \\ &= T(2^{k-3}) + 3d \end{aligned}$$

Decimos que la expresión general para $k - i \geq 0$:

$$T(2^k) = T\left(2^{k-i}\right) + i \cdot d$$

Considerando $i = k$ obtenemos:

$$T(2^k) = T(1) + k \cdot d$$

Dado que $k = \log_2(n)$ (por cambio de variable), obtenemos que:

$$T(n) = c + d \cdot \log_2(n)$$

para n potencia de 2.

Queremos demostrar que $T(n) \in \mathcal{O}(\log_2(n))$

- Vale decir, tenemos que demostrar que existen $e \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $T(n) \leq e \cdot \log_2(n)$ para todo $n \geq n_0$.

Inducción constructiva:

Dado que $T(1) = c$ y $\log_2(1) = 0$ no es posible encontrar un valor para e tal que $T(1) \leq e \cdot \log_2(1)$.

Dado que $T(2) = c + d$, si consideramos $e = (c + d)$ tenemos que $T(2) \leq e \cdot \log_2(2)$.

- Definimos entonces $e = (c + d)$ y $n_0 = 2$.

Tenemos que demostrar lo siguiente:

$$\forall n \geq 2.\ T(n) \leq e \cdot \log_2(n)$$

Para esto, utilizaremos inducción fuerte.

- $n = 2$ es el punto de partida y el primer caso base.
- $n = 3$ también es un caso base ya que $T(3) = T(1) + d$ y para $T(1)$ no se cumple la propiedad.
- Para $n \geq 4$ tenemos que $T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + d$ y $\left\lfloor \frac{n}{2} \right\rfloor \geq 2$, por lo que resolvemos este caso de manera inductiva.
    - Suponemos que la propiedad se cumple para todo $k \in \{2, \dots, n-1\}$.

Casos base:

- $T(2) = c + d = e \cdot \log_2(2)$
- $T(3) = c + d < e \cdot \log_2(3)$

Caso inductivo:

Suponemos que $n \geq 4$ y para todo $k \in \{2, \dots, n-1\}$ se tiene que $T(k) \leq e \cdot \log_2(k)$.

Usando la definición de $T(n)$ y la hipótesis de inducción concluimos que:

$$\begin{aligned} T(n) &= T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + d \\ &\leq e \cdot \log_2\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + d \\ &\leq e \cdot \log_2\left(\frac{n}{2}\right) + d \\ &= e \cdot \log_2(n) - e \cdot \log_2(2) + d \\ &= e \cdot \log_2(n) - (c + d) + d \\ &= e \cdot \log_2(n) - c \\ &< e \cdot \log_2(n) \end{aligned}$$

El Teorema Maestro

Muchas de las ecuaciones de recurrencia que vamos a usar en este curso son de la forma:

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T\left(\left\lfloor\frac{n}{b}\right\rfloor\right) + f(n) & n \geq 1 \end{cases}$$

El **Teorema Maestro** nos dirá cual es el **orden** de $T(n)$ dependiendo de ciertas condiciones sobre a, b y $f(n)$.

Antes...

Sea $f: \mathbb{N} \to \mathbb{R}_0^+$ una función y $a, b \in \mathbb{R}$ constantes tales que $a \geq 1$ y $b > 1$.

Definición:

La función f es **(a, b)-regular** si existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$\forall n \geq n_0 . a \cdot f\left(\left\lfloor\frac{n}{b}\right\rfloor\right) \leq c \cdot f(n)$$

Ejercicio:

Demuestre que la función $\log_2(n)$ no es $(1,2)$-regular.

Solución:

Por contradiccón, supongamos que $\log_2(n)$ es $(1,2)$-regular.
Entonces existen constantes $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ tales que $c < 1$ y

$$\forall n \geq n_0 . \log_2 \left\lfloor \frac{n}{2} \right\rfloor \leq c \cdot \log_2(n)$$

En particular, podemos concluir que para todo $k \geq n_0$:

$$\log_2 \left\lfloor \frac{2 \cdot k}{2} \right\rfloor \leq c \cdot \log_2(2 \cdot k)$$

Vale decir:

$$\log_2(k) \leq c \cdot (\log_2(k) + 1)$$

Dado que $0 < c < 1$:

$$\log_2(k) \leq \frac{c}{1 - c}$$

Lo cual nos lleva a una contradicción.

Teorema Maestro:

Sea $f: \mathbb{N} \to \mathbb{R}_0^+$ una función, $a, b, c \in \mathbb{R}_0^+$ constantes tales que $a \geq 1$ y $b > 1$, y $T(n)$ una función definida por la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} c & n = 0 \\ a \cdot T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n) & n \geq 1 \end{cases}$$

Se tiene que:

1. Si $f(n) \in \mathcal{O}(n^{\log_b(a) - \varepsilon})$ para $\varepsilon > 0$, entonces $T(n) \in \Theta\left(n^{\log_b(a)}\right)$.

2. Si $f(n) \in \Theta(n^{\log_b(a)})$, entonces $T(n) \in \Theta\left(n^{\log_b(a)} \cdot \log_2(n)\right)$.

3. Si $f(n) \in \Omega(n^{\log_b(a) + \varepsilon})$ para $\varepsilon > 0$ y f es (a, b)-regular, entonces $T(n) \in \Theta\left(f(n)\right)$.

Ejemplo:

Considere la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} 1 & n = 0 \\ 3 \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c \cdot n & n \geq 1 \end{cases}$$

Dado que $\log_2(3) > 1.5$, tenemos que $\log_2(3) - 0.5 > 1$
Deducimos que $c \cdot n \in \mathcal{O}\left(n^{\log_2(3)-0.5}\right)$, por lo que usando el Teorema Maestro concluimos que$T(n) \in \Theta\left(n^{\log_2(3)}\right)$.

El Teorema Maestro sigue siendo válido pero con $T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$ reemplazado por $T\left(\left\lceil \frac{n}{b} \right\rceil\right) + f(n)$

Sea $\mathcal{A}: \Sigma^* \to \Sigma^*$ un algoritmo:

Definición

Decimos que $\mathcal{A}$ en el **peor caso** es $\mathcal{O}\left(f(n)\right)$ si:

$$t_{\mathcal{A}}(n) \in \mathcal{O}\left(f(n)\right)$$

Recordar que $t_{\mathcal{A}}(n)$ es el mayor número de pasos realizados por $\mathcal{A}$ sobre las entradas $w \in \Sigma^*$ de largo n.

Dividir para conquistar

Algoritmo genérico:

```
DividirParaConquistar(w)
    if |w| ≤ k then return InstanciasPequeñas(w)
    else
        Dividir w en w_1, ..., w_ℓ
        for i := 1 to ℓ do
            S_i := DividirParaConquistar(w_i)
        return Combinar(S_1, ..., S_ℓ)
```

Algoritmo de multiplicación de Karatsuba

Sean $a, b \in \mathbb{Z}$ con n dígitos cada uno, donde $n = 2^k$ para algún $k \in \mathbb{N}$.
Se puede representar a y b de la siguiente forma:

$$a = a_1 \cdot 10^{\frac{n}{2}} + a_2$$
$$b = b_1 \cdot 10^{\frac{n}{2}} + b_2$$

Tenemos entonces que:

$$\boxed{a \cdot b = a_1 \cdot b_1 \cdot 10^n + (a_1 \cdot b_2 + a_2 \cdot b_1) \cdot 10^{\frac{n}{2}} + a_2 \cdot b_2}$$

Para calcular $a \cdot b$ debemos calcular las siguiente multiplicaciones:

1. $a_1 \cdot b_1$
2. $a_1 \cdot b_2$
3. $a_2 \cdot b_1$
4. $a_2 \cdot b_2$

Podemos calcular $a \cdot b$ realizando las siguiente multiplicaciones:

1. $c_1 = a_1 \cdot b_1$
2. $c_2 = a_2 \cdot b_2$
3. $c_3 = (a_1 + a_2) \cdot (b_1 + b_2)$

Tenemos entonces que:

$$\boxed{a \cdot b = c_1 \cdot 10^n + \left(c_3 - (c_1 + c_2)\right) \cdot 10^{\frac{n}{2}} + c_2}$$

Esta expresión se conoce como el **algorimto de Karatsuba**.

Tiempo de ejecución del algoritmo de Karatsuba:

$$T(n) = \begin{cases} 1 & n = 1 \\ 3 \cdot T\left(\frac{n}{2}\right) + e \cdot n & n > 1 \end{cases}$$

Supuesto:

- n es una potencia de 2 y $(a_1 + a_2)$ y $(b_1 + b_2)$ tienen $\frac{n}{2}$ dígitos cada uno.

¿Qué representa la constante e?

- Calcular $(a_1 + a_2), (b_1 + b_2), (c_1 + c_2)$ y $\left(c_3 - (c_1 + c_2)\right)$.
- Construir $a \cdot b$ a partir de $c_1,\ c_2$ y $\left(c_3 - (c_1 + c_2)\right)$, lo cual puede tomar tiempo lineal en el pero caso.

Utilizando el Teorema Maestro obtenemos que $T(n)$ es $\Theta\left(n^{\log_2(3)}\right)$, pero este resultado es válido bajo los supestos realizados anteriormente.

Caso general:

Representamos las entradas a y b de la siguiente forma:

$$a = a_1 \cdot 10^{\left\lfloor \frac{n}{2} \right\rfloor} + a_2$$
$$b = b_1 \cdot 10^{\left\lfloor \frac{n}{2} \right\rfloor} + b_2$$

La siguiente ecuación de recurrencia para $T(n)$ captura la cantidad de operaciones relaizadas por el algoritmo (para constantes e_1, e_2):

$$T(n) = \begin{cases} e_1 & n \leq 3 \\ T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) + e_2 \cdot n & n > 3 \end{cases}$$

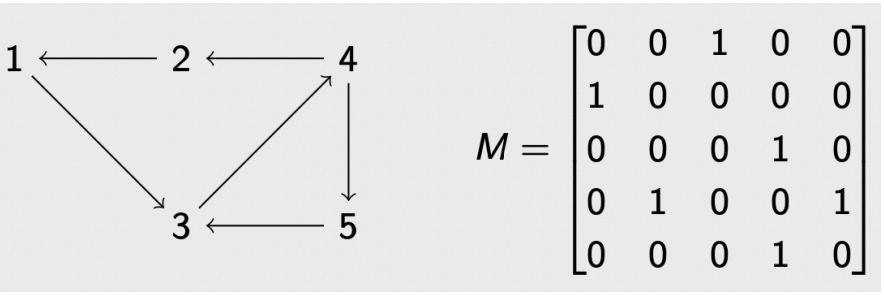
Programación dinámica: Grafos

Sea $G = (V, E)$, un par de nodos $v_i, v_f \in V$, y un número l, queremos desarrollar un algoritmo que cuente el número de caminos desde v_i a v_f en G cuyo largo es igual a l.

Suponemos que $V = \{1, \dots, n\}, 1 \leq l \leq n$ y represenatmos G a través de su **matiz de adyacencia** M tal que:

Si $(i, j) \in E$, entonces $M[i, j] = 1$, en caso contrario $M[i, j] = 0$.

Ejemplo:



Queremos calcular el número de caminos de largo l desde un nodo v_i, a un nodo v_f en un grafo G (representado por una matriz de adyacencia M)
Para evitar hacer llamadas recursivas repetidas, y así disminuir el número de llamadas recursivas, definimos una **secuencia de matrices** $H_1, \dots, H_l$ tales que:

$$H_k[v_i, v_f] \quad := \quad \text{número de caminos de } v_i \text{ a } v_j \text{ de } largo\ k$$

La secuencia $H_1, \dots, H_k$ s puede definir recursivamente de la siguiente forma:

1. $H_1 = M$.
2. $H_{k+1} = M \cdot H_k$ para $k \in \{1, \dots, l-1\}$.

Luego, la cantidad de caminos de largo l desde v_i a v_f será $\boxed{H_l[v_i, v_f]}$.

ContarTodosCaminos$(M[1 \dots n][1 \dots n],\ \ell)$
  if $\ell = 1$ **then return** M
  else
    $H :=$ **ContarTodosCaminos**$(M,\ \ell - 1)$
    return $M \cdot H$

ContarCaminos$(M[1 \dots n][1 \dots n],\ v_i,\ v_f,\ \ell)$
  $H :=$ **ContarTodosCaminos**$(M,\ \ell)$
  return $H[v_i, v_f]$

Programación dinámica: Palabras

Midiendo la distancia entre dos palabras.
Vamos a utilizar la **distancia de Levenshtein** para medir cuán similares son dos palabras.

Dadas dos palabras $w_1, w_2 \in \Sigma^*$, utilizamos la notación $\text{ed}(w_1, w_2)$ para la edit distance entre w_1 y w_2.

Tres operadores sobre palabras:

Para $i \in \{1, \dots, n\}$ y $b \in \Sigma$ tenemos que:

1. $\text{eliminar}(w, i) = a_1 \cdots a_{i-1} \cdot a_{i+1} \cdots a_n.$
2. $\text{agregar}(w, i, b) = a_1 \cdots a_i \cdot b \cdot a_{i+1} \cdots a_n.$
3. $\text{cambiar}(w, i, b) = a_1 \cdots a_{i-1} \cdot b \cdot a_{i+1} \cdots a_n.$

Dadas palabras $w_1, w_2 \in \Sigma^*$, definimos $\text{ed}(w_1, w_2)$ como el menor número de operaciones eliminar, agregar y cambiar que aplicadas desde w_1 generan w_2. Para cualcular esta distancia utilizamos **programación dinámica**.

Definimos el **infijo** (substring) de w entre las posiciones i y j como:

$$w[i,j] = \begin{cases} w[i] \cdots w[j] & 1 \leq i \leq j \leq n \\ \varepsilon & \text{en caso contrario} \end{cases}$$

Fije dos strings $w_1, w_2 \in \Sigma^*$ tales que $|w_1| = m$ y $|w_2| = n$.

Dados $i \in \{0, \dots, m\}$ y $j \in \{0, \dots, n\}$, definimos:

$$\text{ed}(i,j) = \text{ed}(w_1[1,i], w_2[1,j])$$

Observe que $\text{ed}(w_1, w_2) = \text{ed}(m, n)$

Además, definimos el valor $\text{dif}(i,j)$ como 0 si $w_1[i] = w_2[j]$, y como 1 en caso contrario.

Definición recursiva de la distancia de Levenshtein

Del principio de optimalidad para sub=secuencias obtenemos la siguiente **definición recursiva** para la función ed:

$$\mathrm{ed}(i,j) = \begin{cases} \max\{i,j\} & i = 0 \text{ o } j = 0 \\ \min \begin{array}{l} \{1 + \mathrm{ed}(i-1,j), \\ 1 + \mathrm{ed}(i,j-1), \\ \mathrm{dif}(i,j) + \mathrm{ed}(i-1,j-1)\} \end{array} & i > 0 \text{ y } j > 0 \end{cases}$$

Tenemos entonces una forma de calcular la función ed que se basa en resolver sub-problemas más pequeños.

- Estos sub-problemas están traslapados y se tiene un número polinomial de ellos, podemos aplicar entonces programación dinámica.

```
EditDistance(w1, i, w2, j)
    if i = 0 then return j
    else if j = 0 then return i
    else
        r := EditDistance(w1, i − 1, w2, j)
        s := EditDistance(w1, i, w2, j − 1)
        t := EditDistance(w1, i − 1, w2, j − 1)
        if w1[i] = w2[j] then d := 0
        else d := 1
        return min{1 + r, 1 + s, d + t}
```

¿Es esta una buena implementación de **EditDistance**?

R: No porque tenemos muchas llamadas recursivas repetidas, es mejor evaluar esta función utilizando un **enfoque bottom-up**.

Evaluación bottom-up:

Para determinar los valores de la función ed construimos una tabla siguiendo un orden lexicográfico para los pares (i,j):

$$(i_1,j_1) < (i_2,j_2) \quad \text{si y solo si} \quad i_1 < i_2 \quad \text{o} \quad (i_1 = i_2 \text{ y } j_1 < j_2)$$

Por ejemplo, para determinar el valor de $\text{ed}(\text{casa}, \text{asado})$ construimos la siguiente tabla:

		a	s	a	d	o
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	2	3	4
s	3	2	1	2	3	4
a	4	3	2	1	2	3

		a	s	a	d	o
	0	1	2	3	4	5
c	1	1	2	3	4	5
a	2	1	2	2	3	4
s	3	2	1	2	3	4
a	4	3	2	1	2	3

Estas operaciones son la siguientes:

1. $\text{eliminar}(\text{casa}, 1) = \text{asa}$
2. $\text{agregar}(\text{asa}, 3, d) = \text{asad}$
3. $\text{agregar}(\text{asad}, 4, 0) = \text{asado}$

Corolario:

Utilizando programación dinámica es poible construir un algoritmo para calcular $\text{ed}(w_1, w_2)$ que en el peor caso es $\Theta(|w_1| \cdot |w_2|)$.

Algoritmos codiciosos

Almacenamiento de Datos:

Sea Σ un alfabeto. Dada una palabra $w \in \Sigma^*$ suponga que queremos **almacenar** w **utilizando los símbolos 0 y 1**.

Definimos entonces una función $\tau: \Sigma \to \{0,1\}^*$ que asigna a cada símbolo en $a \in \Sigma$ una palabra en $\tau(a) \in \{0,1\}^*$ con $\tau(a) \neq \varepsilon$.

- Vamos a almacenar w reemplazando cada símbolo $a \in \Sigma$ que aparece en w por $\tau(a)$.
- Llamamos a τ una **Σ-codificación**.

La **extensión $\hat{\tau}$** de una Σ-codificación τ a todas las palabras $w \in \Sigma^*$ se define como:

$$\hat{\tau}(w) = \begin{cases} \varepsilon & w = \varepsilon \\ \tau(a_1) \cdots \tau(a_n) & w = a_1 \cdots a_n \text{ con } n \geq 1 \end{cases}$$

Vamos a almacenar w como $\hat{\tau}(w)$.

- Si la Σ-codificación τ esta fija (como en ASCII) entonces no es necesario almacenarla.
- Si τ no está fija, entocnes debemos almacenarla junto con $\hat{\tau}(w)$
    - En general $|w|$ es mucho más grande que $|\Sigma|$, por lo que el costo de almacenar τ es despreciable.

La función $\hat{\tau}$ debe especificar una traducción no ambigua:

$$\forall w_1, w_2 \in \Sigma^*. w_1 \neq w_2 \Rightarrow \hat{\tau}(w_1) \neq \hat{\tau}(w_2)$$

De esta forma podemos reconstruir el texto original dada su traducción
Vale decir, $\hat{\tau}$ debe ser una **función inyectiva**.

Codificaciones de largo fijo

Es claro que para lograr una traducción no ambigua necesitamos que $\tau(a) \neq \tau(b)$ para cada $a, b \in \Sigma$ tal que $a \neq b$.

Para obtener la misma propiedad para $\hat{\tau}$ podemos imponer la siguiente condición:

$$\forall a, b \in \Sigma. |\tau(a)| = |\tau(b)|$$

Si se cumple esta condición entonces diremos que τ es una **Σ-codificación de largo fijo**.

Por otro lado, decimos que τ es una **Σ-codificación de largo variable** si

$$a, b \in \Sigma. \quad |\tau(a)| \neq |\tau(b)|$$

¿Por qué nos conviene utilizar codificaciones de **largo variable**?

R: Podemos obtener representaciones más cortas para la palabra que queremos almacenar.

Ejemplo:

Suponga que $w = aabaacaab$

- Para $\tau_1(a) = 00$, $\tau_1(b) = 01$ y $\tau_1(c) = 10$ tenemos que:

$$\hat{\tau}(w) = 000001000010000001$$

- Para $\tau_2(a) = 0$, $\tau_2(b) = 10$ y $\tau_2(c) = 11$ tenemos que:

$$\hat{\tau}(w) = 001000110010$$

Por lo tanto $|\hat{\tau}_2(w)| = 12 < 18 = |\hat{\tau}_1(w)|$.

Lema:

Si existen $w_1, w_2 \in \Sigma^*$ tales que $w_1 \neq w_2$ y $\hat{\tau}(w_1) = \hat{\tau}(w_2)$ entonces existen $a, b \in \Sigma$ tales que $a \neq b$ y $\tau(a)$ es un prefijo de $\tau(b)$.

Demostración

Suponga que $w_1 \neq w_2$, $\hat{\tau}(w_1) = \hat{\tau}(w_2)$ y

$$\begin{aligned} w_1 &= a_1 \dots a_m \qquad m \geq 1 \\ w_2 &= a_1 \dots a_n \qquad n \geq 1 \end{aligned}$$

Además, sin pérdida de generalidad suponga que $m \leq n$.

Si w_1 es **prefijo propio** de w_2 entonces $\hat{\tau}(w_1)$ es prefijo propio de $\hat{\tau}(w_2)$

- Puesto que $\tau(a) \neq \varepsilon$ para cada $a \in \Sigma$.

Dado que $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, tenemos entonces que w_1 no es prefijo propio de w_2.

Ahora, sea $k = \min_{1 \leq i \leq m} a_i \neq b_i$

k está bien definido puesto que $w_1 \neq w_2$ y w_1 no es prefijo propio de w_2.

Dado que $\hat{\tau}(a_1 \dots a_{k-1}) = \hat{\tau}(b_1 \dots b_{k-1})$ y $\hat{\tau}(w_1) = \hat{\tau}(w_2)$, concluimos que $\hat{\tau}(a_k \dots a_m) = \hat{\tau}(b_k \dots b_n)$.

Tenemos entonces que $\tau(a_k) \cdots \tau(a_m) = \tau(b_k) \cdots \tau(b_n)$, de lo cual se deduce que $\tau(a_k)$ es un prefijo de $\tau(b_k)$, o $\tau(b_k)$ es un prefijo de $\tau(a_k)$.

Lo cual concluye la demostración puesto que $a_k \neq b_k$. ■

Decimos que una Σ-codificación τ es **libre de prefijos** si para cada $a, b \in \Sigma$ tales que $a \neq b$, se tiene que $\tau(a)$ no es prefijo de $\tau(b)$.

Ejemplo:
Para $\Sigma = \{a, b, c\}$ y $\tau(a) = 0,\ \tau(b) = 10,\ \tau(c) = 11$, se tiene que τ es libre de prefijos.

Corolario:

Si τ es una codificación libre de prefijos, entonces $\hat{\tau}$ es una **función inyectiva**.

Frecuencias relativas de los símbolos

Palabra a almacenar: $w \in \Sigma^*$

Para $a \in \Sigma$ definimos $\mathrm{fr}_w(a)$ como la frecuencia relativa de a en w, vale decir, el número de apariciones de a en w dividido por el largo de w.

- Por ejemplo, si $w = aabaabaac$, entonces $\mathrm{fr}_w(a) = \frac{2}{3}$ y $\mathrm{fr}_w(c) = \frac{1}{9}$.

Para una Σ-codificación τ definimos el largo promedio para w como:

$$\mathrm{lp}_w(\tau) = \sum_{a \in \Sigma} \mathrm{fr}_w(a) \cdot |\tau(a)|$$

Tenemos que $|\hat{\tau}(w)| = \mathrm{lp}_w(\tau) \cdot |w|$

- Por lo tanto queremos una Σ-codificación τ que minimice $\mathrm{lp}_w(\tau)$.

Problema de optimización a resolver

Dado $w \in \Sigma^*$, encontrar una Σ-codificación τ libre de prefijos que minimice el valor $\mathrm{lp}_w(\tau)$.

La función objetivo del algoritmo codicioso es entonces $\mathrm{lp}_w(x)$.

- Queremos minimizar el valor de esta función.

La función $\mathrm{lp}_w(x)$ se define a partir de la función $\mathrm{fr}_w(y)$.

- No se necesita más información sobre w. En particular, no se necesita saber cuál es el símbolo de w en una posición específica.

Podemos entonces trabajar con funciones de frecuencias relativas en lugar de palabras.

- La entrada del problema no va a ser w sino que fr_w.

Decimos que $f: \Sigma \to (0,1)$ es una **función de frecuencias relativas** para Σ si se cumple que:

$$\sum_{a \in \Sigma} f(a) = 1$$

Dada una función f de frecuencias relaticas para Σ y una Σ-codificación τ, el **largo promedio de τ para f** se define como:

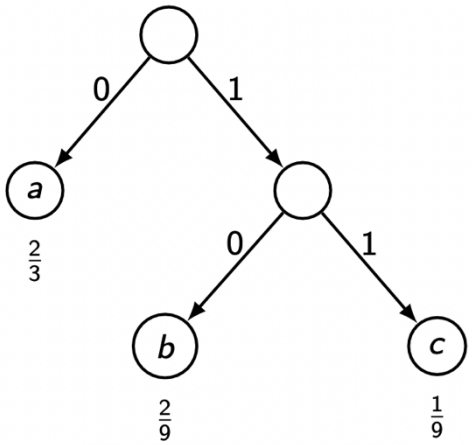
$$\mathrm{lp}_f(\tau) = \sum_{a \in \Sigma} f(a) \cdot |\tau(a)|$$

La entrada del problema es entonces una función f de frecuencias relativas para Σ, y la función objetivo a minimizar es $\mathrm{lp}_f(x)$.

Codiciaciones como árboles

Si una Σ-codificación τ es libre de prefijos, entonces el árbol que la representa satisface las siguientes propiedades.

- Cada hoja tiene como etiqueta un elemento de Σ, y estos son los únicos nodos con etiquetas.
- Cada símbolo de Σ es usado exactamente una vez como etiqueta.
- Cada arco tiene etiqueta 0 ó 1.
- Si una hoja tiene etiqueta e y las etiquetas de los arcos del camino desde la raíz hasta esta hoja forman una palabra $w \in \{0,1\}^*$, entonces $\tau(e) = w$.



Calculando el mínimo de $\mathrm{lp}_f(x)$

- Función objetivo a minimizar: $\mathrm{lp}_f(x)$.
- Función selección: elige los dos símbolos de Σ con menor frecuencia relativa, los coloca como hermanos en el árbol binario que representa la Σ-codificación óptima, y continua la construcción con el resto de los símbolos de Σ.

El algoritmo de Huffman

En el siguiente algoritmo representamos a las funciones como conjuntos de pares ordenados, y suponemos que f es una función de frecuencias relativas que al menos tiene dos elementos en el dominio.

CalcularCodificaciónHuffman(f)
  Sea Σ el dominio de la función f
  if $\Sigma = \{a, b\}$ **then return** $\{(a, 0), (b, 1)\}$
  else
    Sean $a, b \in \Sigma$ tales que $a \neq b$, $f(a) \leq f(b)$ y
      $f(b) \leq f(e)$ para todo $e \in (\Sigma \setminus \{a, b\})$
    Sea c un símbolo que **no** aparece en Σ
    $g := (f \setminus \{(a, f(a)), (b, f(b))\}) \cup \{(c, f(a) + f(b))\}$
    $\tau^\star :=$ **CalcularCodificaciónHuffman**(g)
    $w := \tau^\star(c)$
    $\tau := (\tau^\star \setminus \{(c, w)\}) \cup \{(a, w0), (b, w1)\}$
    return τ

Teorema

Si f es una codificación de frecuencias relativas, Σ es el dominio de f y $\tau =$**CalcularCodificaciónHuffman**(f)**,** entonces τ es una Σ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_f(x)$.

Demostración:

Vamos a realizar la demostración por inducción en $|\Sigma|$.

Si $|\Sigma| = 2$, entonces la propiedad se cumple trivialmente

Suponga que la propiedad se cumple para un valor $n \geq 2$, y suponga que $|\Sigma| = n + 1$.

Sean a, b, c, g, τ^* y τ definidos como en el código de la llamda **CalcularCodificaciónHuffman**(f), y sea Γ el dominio de g.

Como $|\Gamma| = n$ y $\tau^* =$ **CalcularCodificaciónHuffman**(g), por hipótesis de inducción tenemos que τ^* es una Γ-codificación libre de prefijos que minimiza la función $\mathrm{lp}_g(x)$.

Dada la definición de τ es simple verificar las siguiente propiedades:

- τ es una codificaciónn libre de prefijos.
- Para cad $e \in (\Sigma \setminus \{a, b\})$ se tiene que $\tau(e) = \tau^*(e)$.
- $|\tau(a)| = |\tau(b)| = |\tau^*(c)| + 1$.

Por contradicción suponga que τ no minimiza el valor de la función $\mathrm{lp}_f(x)$,

- Vale decir, existe una Σ=codificación τ' libre de prefijos tal que τ' minimiza la función $\mathrm{lp}_f(x)$ y $\mathrm{lp}_f(\tau') < \mathrm{lp}_f(\tau)$.

Por la definición de a, b y los ejercicios anteriores podemos suponer que existe $w \in \{0,1\}^*$ tal que $\tau'(a) = w0$ y $\tau'(b) = w1$.

- Nótese que $w \neq \varepsilon$ puesto que $|\Sigma| \geq 3$.

A partir de τ' defina la siguiente Γ-codificación τ'':

$$\tau'' = \left(\tau' \setminus \left\{\left(a, \tau'(a)\right), \left(b, \tau'(b)\right)\right\}\right) \cup \{(c, w)\}$$

Tenemos que τ'' es una Γ-codificación libre de prefijos.

La relación entre $\mathbf{lp}_g(\tau^*)$ y $\mathbf{lp}_f(\tau)$

$$\mathrm{lp}_g(\tau^*) = \sum_{e \in \Gamma} g(e) \cdot |\tau^*(e)|$$

$$= \left(\sum_{e \in (\Gamma \setminus \{c\})} g(e) \cdot |\tau^*(e)| \right) + g(c) \cdot |\tau^*(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + \big(f(a) + f(b)\big) \cdot |\tau^*(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + f(a) \cdot |\tau^*(c)| + f(b) \cdot |\tau^*(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + f(a) \cdot (|\tau(a)| - 1) + f(b) \cdot (|\tau(b)| - 1)$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau(e)| \right) + (f(a) \cdot |\tau(a)|) + (f(b) \cdot |\tau(b)|) - f(a) - f(b)$$

$$= \left(\sum_{e \in \Sigma} f(e) \cdot |\tau(e)| \right) - \big(f(a) + f(b)\big)$$

$$= \mathrm{lp}_f(\tau) - \big(f(a) + f(b)\big)$$

La relación entre $\mathbf{lp}_g(\tau'')$ y $\mathbf{lp}_f(\tau')$

$$\mathrm{lp}_g(\tau'') = \sum_{e \in \Gamma} g(e) \cdot |\tau''(e)|$$

$$= \left(\sum_{e \in (\Gamma \setminus \{c\})} g(e) \cdot |\tau''(e)| \right) + g(c) \cdot |\tau''(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + \big(f(a) + f(b)\big) \cdot |\tau''(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + f(a) \cdot |\tau''(c)| + f(b) \cdot |\tau''(c)|$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + f(a) \cdot (|\tau'(a)| - 1) + f(b) \cdot (|\tau'(b)| - 1)$$

$$= \left(\sum_{e \in (\Sigma \setminus \{a,b\})} f(e) \cdot |\tau'(e)| \right) + (f(a) \cdot |\tau'(a)|) + (f(b) \cdot |\tau'(b)|) - f(a) - f(b)$$

$$= \left(\sum_{e \in \Sigma} f(e) \cdot |\tau'(e)| \right) - \big(f(a) + f(b)\big)$$

$$= \mathrm{lp}_f(\tau') - \big(f(a) + f(b)\big)$$

Tenemos entonces que

$$\begin{aligned} \mathrm{lp}_g(\tau'') &= \mathrm{lp}_f(\tau') - \big(f(a) + f(b)\big) \\ &< \mathrm{lp}_f(\tau) - \big(f(a) + f(b)\big) \\ &= \mathrm{lp}_g(\tau^*) \end{aligned}$$

Concluimos entonces que τ^* no minimiza la función $\mathrm{lp}_g(x)$, lo cual contradice la hipótsis de inducción.

La siguiente función calcula la codificación de Huffman teniendo como entrada una palabra w:

```
CalcularCodificaciónHuffman(w)
    if w = ε then return ∅
    else
        Σ := conjunto de símbolos mencionados en w
        if Σ = {a} then return {(a, 0)}
        else return CalcularCodificaciónHuffman(fr_w)
```

¿Cuál es la complejidad de **CalcularCodificaciónHuffman**(f)?

R: $\mathcal{O}(n \cdot \log n)$ (considerando que $|f| \in \Theta(|\Sigma|)$.

Transformada rápida de Fourier

Representación de un polinomio

Sea $p(x)$ un **polinomio no nulo de coeficientes racionales**.

La representación canónica de $p(x)$ es:

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

donde $n \geq 1$, $a_{n-1} \neq 0$ y el grado de $p(x)$ es $n - 1$.

- Utilizamos el grado $n - 1$ para dar énfasis a que estos polinomios poseen n coeficientes.
- Si bien trabajaremos con polinomios de coeficientes racionales, vamos a evaluarlos usando números reales y complejos.

Representamos $p(x)$ a través de una **tupla** $(a_0, \ldots, a_{n-1})$ de largo n.

- También podemos representar $p(x)$ como una tupla $(a_0, \ldots, a_{n-1}, 0, \ldots, 0)$ de largo $m > n$ donde cada término x^i tiene coeficiente 0 si $i \geq n$.

Suma de polinomios:

La **suma** de dos polinomios $(a_0, \ldots, a_{n-1})$ y $(b_0, \ldots, b_{n-1})$ es un polinomio $(c_0, \ldots, c_{n-1})$ tal que:

$$c_i = a_i + b_i \qquad \forall i \in \{0, \ldots, n_1\}$$

Consideramos a la **suma y multiplicación de números en** $\mathbb{C}$ como las operaciones básicas a contar.

¿Cuál es la **complejidad** de este algoritmo? **R**: $\mathcal{O}(n)$

Multiplicación de polinomios:

La multiplicación de dos polinomios $(a_0, \dots, a_{n-1})$ y $(b_0, \dots, b_{n-1})$ es un polinomio $(c_0, \dots, c_{n-1})$ tal que:

$$c_i = \sum_{k,l \in \{0,\dots,n-1\} \,:\, k+l=i} a_k \cdot b_l \qquad \forall i \in \{0, \dots, 2n-2\}$$

¿Cuál es la **complejidad** de realizar esta operación? **R**: $\mathcal{O}(n^2)$

Una representación alternativa de un polinomio

Un polinomio $p(x)$ de grado $n-1$ se puede representar de manera única a través de **un conjunto de n pares de puntos-valores** (así como una parábola puede representarse con 3 puntos en $\mathbb{R}^2$):

$$p(x) \mapsto \{(v_0, p(v_0)), (v_1, p(v_1)), \dots, (v_{n-1}, p(v_{n-1}))\},$$

suponiendo que $v_i \neq v_j$ para todo $i \neq j$.

Un polinomio $p(x)$ de grado $n-1$ también se puede representar de manera única a través de un conjunto de n pares de puntos-valores com $m > n$ elementos:

$$p(x) \mapsto \{(v_0, p(v_0)), \dots, (v_{n-1}, p(v_{n-1})), (v_n, p(v_n)), \dots, (v_{m-1}, p(v_{m-1}))\},$$

suponiendo que $v_i \neq v_j$ para todo $i \neq j$.

¿Por qué es útil la represntación basada en puntos-valores?

Sean $p(x)$ y $q(x)$ dos polinomios de grado $n-1$ representados por:

$$p(x) \mapsto \{(v_0, p(v_0)), \dots, (v_{n-1}, p(v_{n-1}))\}$$
$$q(x) \mapsto \{(v_0, q(v_0)), \dots, (v_{n-1}, q(v_{n-1}))\}$$

¿Cuál es la representación de $r(x) = p(x) + q(x)$?

R: $r(x) \mapsto \{(v_0, p(v_0) + q(v_0)), \dots, (v_{n-1}, p(v_{n-1}) + q(v_{n-1}))\}$

¿Cómo lo hacemos para $s(x) = p(x) \cdot q(x)$?

Suponga que se agrega n puntos a las representaciones de $p(x)$ y $q(x)$:

$$\{(v_0, p(v_0)), \dots, (v_{n-1}, p(v_{n-1})), (v_n, p(v_n)), \dots, (v_{2n-1}, p(v_{2n-1}))\}$$
$$\{(v_0, q(v_0)), \dots, (v_{n-1}, q(v_{n-1})), (v_n, q(v_n)), \dots, (v_{2n-1}, q(v_{2n-1}))\}$$

El polinomio $s(x) = p(x) \cdot q(x)$ es representado por:

$$\{(v_0, p(v_0) \cdot q(v_0)), \dots, (v_{2n-1}, p(v_{2n-1}) \cdot q(v_{2n-1}))\}$$

Podemos sumar y multiplicar polinomios en tiempo $\mathcal{O}(n)$ si están representados por partes de puntos-valores (y por los mismos puntos).

De la representación puntos-valores a la canónica

Sea $p(x)$ un polinomio de grado $n-1$ dado por una representación punto-valores:

$$\{(v_0, p(v_0)), \dots, (v_{n-1}, p(v_{n-1})), (v_n, p(v_n)), \dots, (v_{m-1}, p(v_{m-1}))\}$$

donde $m \geq n$.

Podemos pasar a la representación canónica de $p(x)$ utilizando la **fórmula de Lagrange**:

$$p(x) = \sum_{i=0}^{m-1} p(v_i) \cdot \left(\prod_{j \in \{0,\dots,m-1\} \,:\, j \neq i} \frac{x - v_j}{v_i - v_j} \right)$$

La solución: la transformada rápida de Fourier

La transformada rápida de Fourier nos va a permitir entonces calcular la multiplicación de dos polinomios de grado $n-1$ en tiempo $\mathcal{O}(n \cdot \log_2(n))$.

- La idea clave es cómo elegir los puntos $v_0, \ldots, v_{2n-1}$ cuando se calcula la representación como punto-valores de un polinomio de grado $n-1$.

Los **números complejos** y las **raíces de la unidad** juegan un papel fundamental en la definición de la transformada rápida de Fourier.

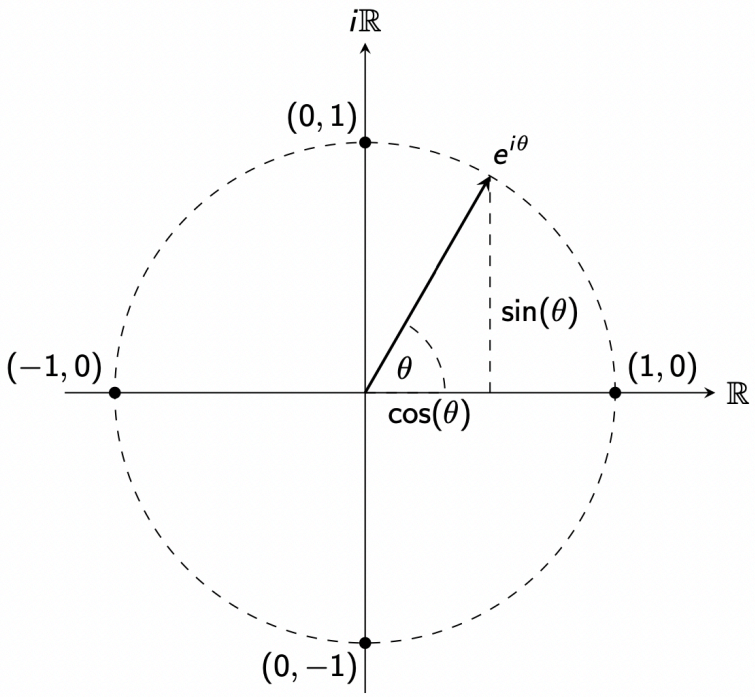
Teorema

Para todo número real x:

$$e^{ix} = \cos(x) + i \cdot \sin(x)$$

Podemos representar entonces a $e^{i\theta}$ como un vectos $(\cos(\theta), \sin(\theta))$ en el plano complejo.

- $e^{i\theta}$ en un vector unitario: $\left\|e^{i\theta}\right\| = \cos^2(\theta) + \sin^2(\theta) = 1$.



Dado $n \geq 1$, queremos encontrar las n raíces del polinomio $p(x) = x^n - 1$.

- Sabemos que este polinomio tiene n raíces en los números complejos.
- Llamamos a estos elementos las **n-raíces de la unidad**.

El componente básico para definir las n-raíces de la unidad:

$$\omega_n = e^{\frac{2\pi i}{n}}$$

Las n-raíces de la unidad son $\omega_n^0,\ \omega_n^1,\ \omega_n^2, \dots,\ \omega_n^{n-1}$.

Si $k \in \{0, \dots, n-1\}$ tenemos que:

$$\begin{aligned}
(\omega_n^k)^n &= \left(\left(e^{\frac{2\pi i}{n}}\right)^k\right)^n \\
&= \left(\left(e^{\frac{2\pi i}{n}}\right)^n\right)^k \\
&= \left(e^{2\pi i}\right)^k \\
&= (\cos(2\pi) + i \cdot \sin(2\pi))^k \\
&= 1^k \\
&= 1
\end{aligned}$$

Además, si $0 \leq k \leq l \leq n-1$, entonces:

$$\begin{aligned}
\omega_n^k = \omega_n^l \quad &\Rightarrow \quad \left(e^{\frac{2\pi i}{n}}\right)^k = \left(e^{\frac{2\pi i}{n}}\right)^l \\
&\Rightarrow \quad \left(e^{\frac{2\pi i}{n}}\right)^{l-k} = 1 \\
&\Rightarrow \quad \left(e^{\frac{2\pi(l-k)i}{n}}\right) = 1 \\
&\Rightarrow \quad \cos\left(\frac{2\pi(l-k)}{n}\right) + i \cdot \sin\left(\frac{2\pi(l-k)}{n}\right) = 1 \\
&\Rightarrow \quad \cos\left(\frac{2\pi(l-k)}{n}\right) = 1 \\
&\Rightarrow \quad \frac{l-k}{n} = 0 \\
&\Rightarrow \quad l = k
\end{aligned}$$

Puesto que:
$0 \leq \frac{l-k}{n} \leq \frac{n-1}{n}$

Por lo tanto, $\omega_n^0, \dots, \omega_n^{n-1}$ son elementos distintos.

Ejemplo:

¿Cuáles son las raíces del polinomio $x^4 - 1$?

- Considerando $\omega_4 = e^{\frac{2\pi i}{4}} = e^{\frac{\pi}{2}i}$, tenemos que las 4-raíces de la unidad son:

$$\omega_4^0 = 1$$
$$\omega_4^1 = e^{\frac{\pi}{2}i} = \cos\left(\frac{\pi}{2}\right) + i \cdot \sin\left(\frac{\pi}{2}\right) = i$$
$$\omega_4^2 = \left(e^{\frac{\pi}{2}i}\right)^2 = e^{\pi i} = \cos(\pi) + i \cdot \sin(\pi) = -1$$
$$\omega_4^3 = \left(e^{\frac{\pi}{2}i}\right)^3 = e^{\frac{3\pi}{2}i} = \cos\left(\frac{3\pi}{2}\right) + i \cdot \sin\left(\frac{3\pi}{2}\right) = -i$$

La transformada discreta de Fourier

Definición

Sea $n \geq 2$ y un polinomio $p(x) = \sum_{k=0}^{n-1} a_k x^k$.

La **transformada discreta de Fourier (DFT)** de $p(x)$ se define como:

$$[p(\omega_n^0), p(\omega_n^1), \dots, p(\omega_n^{n-1})]$$

¿Cómo podemos calcular **DFT** de manera eficiente?

Calcular DFT

Recordemos que vamos a representar $p(x)$ a través del vector $\bar{a} = (a_0, \dots, a_{n-1})$.

El problema a resolver es calcular de manera eficiente la DFT de $p(x)$, la cual denotamos como **DFT**$(\bar{a})$.

- Definimos $y_k = p(\omega_n^k)$ para cada $k \in \{0, \dots, n-1\}$, de manera tal que queremos calcular **DFT**$(\bar{a})$.

Calcular DFT de manera eficiente

Suponemos que $n = 2^t$ para $t \in \mathbb{N}$, calculamos **DFT**$(\bar{a})$ realizando los siguientes pasos:

1. Calcular **DFT**$(\bar{a}_0)$ y **DFT**$(\bar{a}_1)$ para dos vectores $\bar{a}_0$ y $\bar{a}_1$ de largo $\frac{n}{2}$.
2. Combinar **DFT**$(\bar{a}_0)$ y **DFT**$(\bar{a}_1)$ para obtener **DFT**$(\bar{a})$.

Es fundamental que el paso 2 sea realizado en tiempo $\mathcal{O}(n)$.

Tenemos que:

$$\begin{aligned} p(x) &= \sum_{k=0}^{n-1} a_k x^k \\ &= \sum_{k=0}^{\frac{n}{2}-1} a_{2k} x^{2k} + \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1} x^{2k+1} \\ &= \sum_{k=0}^{\frac{n}{2}-1} a_{2k} x^{2k} + x \cdot \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1} x^{2k} \end{aligned}$$

Definimos los polinomios:

$$\begin{aligned} q(z) &= \sum_{k=0}^{\frac{n}{2}-1} a_{2k} z^k \\ r(z) &= \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1} z^k \end{aligned}$$

Tenemos que:

$$p(x) = q(x^2) + x \cdot r(x^2)$$

Para calcular $[p(\omega_n^0), p(\omega_n^1), \dots, p(\omega_n^{n-1})]$, tenemos entonces que calcular:

$$[q((\omega_n^0)^2), q((\omega_n^1)^2), \dots, q((\omega_n^{n-1})^2)]$$
$$[r((\omega_n^0)^2), r((\omega_n^1)^2), \dots, r((\omega_n^{n-1})^2)]$$

Pero si $k \in \left\{0, \ldots, \frac{n}{2} - 1\right\}$, entonces tenemos que:

$$\begin{aligned}\left(\omega_n^{\frac{n}{2}+k}\right)^2 &= \omega_n^{n+2k} \\ &= \omega_n^n \cdot \omega_n^{2k} \\ &= \left(e^{\frac{2\pi i}{n}}\right)^n \cdot (\omega_n^k)^2 \\ &= e^{2\pi i} \cdot (\omega_n^k)^2 \\ &= 1 \cdot (\omega_n^k)^2 \\ &= (\omega_n^k)^2\end{aligned}$$

¿ Si $\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$ son las raíces de la unidad, quiénes son $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{\frac{n}{2}-1}\right)^2$?

Lema

Si $n \geq 2$ es par, entonces $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{\frac{n}{2}-1}\right)^2$ son las $\frac{n}{2}$-raíces de la unidad (vale decir, son las raíces del polinomio $x^{\frac{n}{2}} - 1$).

Demostración

Primero tenemos que demostrar la regla de simplificación $\omega_{m \cdot l}^{k \cdot l} = \omega_m^k$ para $l > 0$:

$$\omega_{m \cdot l}^{k \cdot l} = \left(e^{\frac{2\pi i}{m \cdot l}}\right)^{k \cdot l} = \left(e^{\frac{2\pi i \cdot l}{m \cdot l}}\right)^k = \left(e^{\frac{2\pi i}{m}}\right)^k = \omega_m^k$$

Dado que $k \in \left\{0, \ldots, \frac{n}{2} - 1\right\}$, se tiene que:

$$(\omega_n^k) = \omega_n^{k \cdot 2} = \omega_{\frac{n}{2} \cdot 2}^{k \cdot 2} = \omega_{\frac{n}{2}}^k \qquad \text{(por la regla de simplificación)}$$

Por lo tanto, $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{\frac{n}{2}-1}\right)^2$ son las $\frac{n}{2}$ – raíces de la unidad.

- Puesto que $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, \left(\omega_n^{\frac{n}{2}-1}\right)^2 = \omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \ldots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}$.

Recuerde que $\bar{a} = (a_0, \dots, a_{n-1})$, y defina:

$$\bar{a}_0 = (a_0, a_2, \dots, a_{n-2})$$
$$\bar{a}_1 = (a_1, a_3, \dots, a_{n-1})$$

De los resultados anteriores concluimos que para calcular **DFT**$(\bar{a})$, primero tenemos que calcular **DFT**$(\bar{a})_0$ y **DFT**$(\bar{a}_1)$.

¿Cómo se construye **DFT**$(\bar{a})$ a partir de **DFT**$(\bar{a}_0)$ y **DFT**$(\bar{a}_1)$?

Sea:

$$\mathbf{DFT}(\bar{a}_0) = \left[u_0, u_1, \dots, u_{\frac{n}{2}-1}\right]$$
$$\mathbf{DFT}(\bar{a}_1) = \left[v_0, v_1, \dots, v_{\frac{n}{2}-1}\right]$$

Para $k \in \left\{0, \dots, \frac{n}{2} - 1\right\}$ tenemos que:

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

Adenás, para $k \in \{0, \ldots, \frac{n}{2} - 1\}$ tenemos que:

$$\begin{aligned}
y_{\frac{n}{2}+k} &= p\left(\omega_n^{\frac{n}{2}+k}\right) \\
&= q\left(\left(\omega_n^{\frac{n}{2}+k}\right)^2\right) + \omega_n^{\frac{n}{2}+k} \cdot r\left(\left(\omega_n^{\frac{n}{2}+k}\right)^2\right) \\
&= q(\omega_n^{n+2k}) + \omega_n^{\frac{n}{2}+k} \cdot r(\omega_n^{n+2k}) \\
&= q(\omega_n^n \cdot \omega_n^{2k}) + \left(e^{\frac{2\pi i}{n}}\right)^{\frac{n}{2}} \cdot \omega_n^k \cdot r(\omega_n^n \cdot \omega_n^{2k}) \\
&= q(1 \cdot \omega_n^{2k}) + e^{\pi i} \cdot \omega_n^k \cdot r(1 \cdot \omega_n^{2k}) \\
&= q(\omega_n^{2k}) - \omega_n^k \cdot r(\omega_n^{2k}) \\
&= q\left(\omega_{\frac{n}{2}}^k\right) - \omega_n^k \cdot r\left(\omega_{\frac{n}{2}}^k\right) \\
&= u_k - \omega_n^k \cdot v_k
\end{aligned}$$

Resumiendo, para $k \in \left\{0, \ldots, \frac{n}{2} - 1\right\}$ tenemos que:

$$\begin{aligned}
y_k &= u_k + \omega_n^k \cdot v_k \\
y_{\frac{n}{2}+k} &= u_k - \omega_n^k \cdot v_k
\end{aligned}$$

Para tener un algoritmo recursivo para calcular **DFT** sólo nos falta el **caso base**.

- Consideramos $n = 2$ y un polinomio $p(x) = a_0 + a_1 x$.

Tenemos que:

$$\begin{aligned}
p(\omega_2^0) &= a_0 + a_1 \cdot \omega_2^0 = a_0 + a_1 \\
p(\omega_2^1) &= a_0 + a_1 \cdot \omega_2^1 = a_0 - a_1
\end{aligned}$$

Un algoritmo recursivo eficiente para **DFT**

- La entrada del algoritmo es un polinomio $p(x) = \sum_{k=0}^{n-1} a_k x^k$.
  - Este polinomio es representado por el vector $\bar{a} = (a_0, \ldots, a_{n-1})$.
- Supongamos además que $n \geq 2$ y n es una potencia de 2.
- El algoritmo se llama la **transformada rápida de Fourier (FFT)**.
  - Fue propuesto por *Cooley & Tukey* (1965).

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