La tranformada discreta de Fourier

La transformada discreta de Fourier (clásica) es la aplicación

$$\begin{aligned} DFT_n : \mathbb{C}[x]_{<n} &\longrightarrow \mathbb{C}^n \\ p(x) &\mapsto \big(p(1), p(\xi_n), p(\xi_n^2), \dots, p(\xi_n^{n-1})\big) \end{aligned}$$

donde $\mathbb{C}[x]_{<n}$ es el conjunto de polinomios en $\mathbb{C}[x]$ de grado menor que n y $\xi_n \in \mathbb{C}$ es una raíz n-ésima primitiva de la unidad, es decir, $\xi_n^n = 1$ pero $\xi_n^i \neq 1$ para $i = 1, \dots, n-1$. Es inmediato que DFT_n es un morfismo de $\mathbb{C}$-espacios vectoriales. Si representamos a los elementos de $\mathbb{C}[x]_{<n}$ en la base $\{1, x, x^2, \dots, x^{n-1}\}$ y a los de $\mathbb{C}^n$ en la base canónica, la matriz de DFT_n es

$$D(\xi_n) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \xi_n & \xi_n^2 & \cdots & \xi_n^{n-1} \\ 1 & \xi_n^2 & \xi_n^4 & \cdots & \xi_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_n^{n-1} & \xi_n^{2(n-1)} & \cdots & \xi_n^{(n-1)^2} \end{bmatrix} \in \mathbb{C}^{n\times n}.$$

Esta matriz es simétrica y de tipo Vandermonde.

La transformada discreta de Fourier es un monomorfismo, ya que un polinomio de grado menor que n está determinado por su evaluación en n puntos distintos. Como además $\mathbb{C}[x]_{<n}$ y $\mathbb{C}^n$ son $\mathbb{C}$-espacios vectoriales de la misma dimensión, DFT_n es también un isomorfismo. La inversa de DFT_n, que se denota $IDFT_n : \mathbb{C}^n \longrightarrow \mathbb{C}[x]_{<n}$, corresponde con la interpolación de polinomios.

Teorema 7.1. *La matriz inversa de $D(\xi_n)$ es $\frac{1}{n}D(\xi_n^{-1})$.*

Proof. En lo que sigue utilizaremos índices $0, 1, \dots, n-1$ para indicar las filas y columnas de matrices de $n \times n$.

$$(D(\xi_n)D(\xi_n^{-1}))_{i,j} = \sum_{k=0}^{n-1} D(\xi_n)_{i,k} D(\xi_n^{-1})_{k,j} = \sum_{k=0}^{n-1} \xi_n^{ik}\xi_n^{-kj} = 1 + \xi_n^{i-j} + \xi_n^{2(i-j)} + \cdots + \xi_n^{(n-1)(i-j)}$$

Es claro que si $i = j$, los n sumandos son todos 1 y por lo tanto $(D(\xi_n)D(\xi_n^{-1}))_{i,i} = n$. Si $i \neq j$, tenemos la suma de una progresión geométrica de razón ξ_n^{i-j}, y entonces $(D(\xi_n)D(\xi_n^{-1}))_{i,j} = \frac{1-\xi_n^{n(i-j)}}{1-\xi_n^{i-j}} = 0$, ya que $\xi_n^n = 1$ y $\xi_n^{i-j} \neq 1$. Combinando ambos resultados, obtenemos que $D(\xi_n)D(\xi_n^{-1}) = n \cdot \mathrm{I}_{n\times n}$ y por lo tanto $D(\xi_n)$ es invertible y su inversa es $\frac{1}{n}D(\xi_n^{-1})$. □

Es claro que los $\mathbb{C}$-espacios vectoriales $\mathbb{C}[x]_{<n}$ y $\mathbb{C}[x]/\langle x^n - 1\rangle$ son isomorfos, ya que en toda clase de equivalencia módulo $x^n - 1$ hay un único representante de grado menor que n. Es habitual considerar la transformada discreta de Fourier como la aplicación

$$\begin{aligned} \overline{DFT}_n : \mathbb{C}[x]/\langle x^n - 1\rangle &\longrightarrow \mathbb{C}^n \\ [p(x)] &\mapsto \big(p(1), p(\xi_n), \dots, p(\xi_n^{n-1})\big) \end{aligned}$$

que está bien definida, ya que el polinomio $x^n - 1$ se anula en todos los ξ_n^i con $i = 0, \dots, n-1$. La ventaja de esta formulación es que $\overline{DFT}_n$ es un morfismo de anillos.

El resultado más importante desde el punto de vista computacional es el método de Cooley-Tuckey para calcular $DFT_n(p(x))$ haciendo $O(n\log(n))$ operaciones en $\mathbb{C}$ en el caso en que n es una potencia de 2. A este algoritmo se lo llama la *transformada rápida de Fourier*.

Digamos que $n = 2^k$ para un cierto $k \geq 0$. Nuestro objetivo es implementar la función `FFT(p,xi)` que toma una lista de números complejos `p=[p0,p1,p2,...]` de longitud `n=len(p)` y una raíz primitiva n-ésima `xi` y calcula la transformada de Fourier del polinomio $p_0 + p_1 x + \cdots + p_{n-1}x^{n-1} \in \mathbb{C}[x]_{<n}$. La función devuelve una lista `[a0,a1,a2,...]` de números complejos de la misma longitud que `p`.

El truco de Cooley-Tuckey consiste en reescribir al polinomio $p(x)$ separando los exponentes pares de los impares. Más precisamente,

$$p(x) = p_{even}(x^2) + x \cdot p_{odd}(x^2) \tag{1}$$

con $p_{even}, p_{odd} \in \mathbb{C}[x]_{<n/2}$. En el código, bastará con crear dos listas `p_even` y `p_odd` con las entradas de `p` de índice par e impar, respectivamente. Además, notamos que si $\xi \in \mathbb{C}$ es una raíz n-ésima primitiva de la unidad, entonces ξ^2 es una raíz $(n/2)$-ésima primitiva de la unidad. Si calculamos recursivamente `a_even = FFT(p_even, xi**2)` y `a_odd = FFT(p_odd, xi**2)`, tendremos los valores de $p_{even}(\xi^{2i})$ y $p_{odd}(\xi^{2i})$ para $i = 0, 1, \ldots, \frac{n}{2} - 1$. Para cualquier $i \geq \frac{n}{2}$, los valores de $p_{even}(\xi^{2i})$ y $p_{odd}(\xi^{2i})$ son los mismos que ya calculamos, es decir, $p_{even}(\xi^{2(i-n/2)})$ y $p_{odd}(\xi^{2(i-n/2)})$. Lo único que resta hacer es utilizar la ecuación (1) para obtener $p(\xi^i)$ para $i = 0, 1, \ldots, n-1$. Concretamente, para cada $i = 0, 1, \ldots, n/2 - 1$, hay que hacer `a[i] = a_even[i] + xi**i * a_odd[i]` y `a[i+n/2] = a_even[i] - xi**i * a_odd[i]` .

```python
# algoritmo de Cooley-Tuckey para calcular DFT_n(p)
# n = len(p) debe ser una potencia de 2 y xi debe ser
# una raíz n-ésima primitiva de la unidad
def fft(p, xi):
    n = len(p)
    if n == 1:
        return p
    p_even = [0j] * (n//2)
    p_odd  = [0j] * (n//2)
    for i in range(n//2):
        p_even[i] = p[2*i]
        p_odd[i]  = p[2*i+1]
    a_even = fft(p_even, xi**2)
    a_odd  = fft(p_odd,  xi**2)
    a = [0j] * n
    for i in range(n//2):
        a[i]      = a_even[i] + xi**i * a_odd[i]
        a[i+n//2] = a_even[i] - xi**i * a_odd[i]
    return a

# esta función calcula la transformada inversa
# utilizando que D(xi)^(-1) = 1/n * D(1/xi)
def ifft(a, xi):
    n = len(a)
    p = fft(a, 1.0/xi)
    for i in range(n):
        p[i] /= n
    return p
```

Programa 1: fft1.py

La aplicación principal de DFT_n en álgebra computacional es para calcular el producto de dos polinomios en $\mathbb{C}[x]$. Supongamos que $f, g \in \mathbb{C}[x]$ satisfacen $\deg(f) + \deg(g) < n$. Entonces,

$$fg = IDFT_n\left(DFT_n(f) \cdot DFT_n(g)\right),$$

donde el producto en $\mathbb{C}^n$ se hace coordenada a coordenada. Esto funciona bien, ya que $\overline{DFT}_n$ y $\overline{IDFT}_n$ son morfismos de anillos y $\deg(fg) < n$. Por supuesto, el cálculo se hace eligiendo n una potencia de 2 mayor que $\deg(f) + \deg(g)$ y utilizando la transformada rápida de Fourier. Esto muestra que es posible multiplicar dos polinomios en $\mathbb{C}[x]$ en $O(n \log n)$ operaciones en $\mathbb{C}$, donde $n = \deg(f) + \deg(g)$.

Problema 7.1. Definimos $\widehat{DFT}_n : \mathbb{C}^n \to \mathbb{C}^n$ interpretando a los vectores de $\mathbb{C}^n$ como polinomios en $\mathbb{C}[x]_{<n}$, es decir, $\widehat{DFT}_n(v) = DFT_n(v_0 + v_1 x + \cdots + v_{n-1}x^{n-1})$. Sean $v, w \in \mathbb{C}^n$. Definimos la convolución $v * w$ como el vector de longitud n dado por $(v * w)_i = \sum_{j=0}^{n-1} v_j w_{i-j}$, donde los índices se entienden módulo n. Demostrar que

$$v * w = \widehat{IDFT}_n(\widehat{DFT}_n(v) \cdot \widehat{DFT}_n(w)),$$

donde el $\cdot$ representa el producto coordenada a coordenada.

Problema 7.2. Sean $v, w \in \mathbb{C}^n$. Definimos la negaconvolución $v \mathbin{\bar{*}} w \in \mathbb{C}^n$ mediante

$$(v \mathbin{\bar{*}} w)_i = \sum_{j+k=i} v_j w_k - \sum_{j+k=i+n} v_j w_k.$$

En otras palabras, si se interpretan a los vectores de $\mathbb{C}^n$ como elementos $[v], [w] \in \mathbb{C}[x]/\langle x^n + 1\rangle$, la negaconvolución corresponde con el producto $[v][w]$ en A. Sea $\xi_{2n} \in \mathbb{C}$ una raíz $2n$-ésima de la unidad y sea $a = (1, \xi_{2n}, \xi_{2n}^2, \ldots, \xi_{2n}^{n-1}) \in \mathbb{C}^n$. Demostrar que

$$v \mathbin{\bar{*}} w = a^{-1} \cdot \widehat{IDFT}_n(\widehat{DFT}_n(a \cdot v) \cdot \widehat{DFT}_n(a \cdot w)),$$

donde el $\cdot$ representa el producto coordenada a coordenada.

Problema 7.3. Desarrollar un método recursivo (similar al de Cooley-Tuckey) para calcular DFT_n para $n = 3^k$, separando al polinomio en tres bloques en lugar de dos.

Sea A un anillo conmutativo y sea $\xi_n \in A$ una raíz n-ésima de la unidad tal que $\sum_{k=0}^{n-1} \xi_n^{ik} = 0$ para $i = 1, \ldots, n-1$. A un ξ_n que cumple esa condición se lo llama raíz n-ésima primitiva de la unidad. En un dominio íntegro, basta con comprobar que $\xi_n^i \neq 1$ para $i = 1, \ldots, n-1$, es decir, que el orden multiplicativo de ξ_n es n. Lamentablemente, eso no es suficiente para anillos conmutativos en general.

Cuidado: la noción de raíz primitiva de $\mathbb{Z}/p^k\mathbb{Z}$ que vimos durante en el tema #4 no coincide exactamente con la de raíz n-ésima primitiva de la unidad en $\mathbb{Z}/p^k\mathbb{Z}$, aunque están muy relacionadas.

Supongamos además que el entero n, considerado como el elemento $n \cdot 1_A$ de A, es invertible. Entonces todos los resultados de arriba, incluyendo el algoritmo de Cooley-Tuckey funcionan en A.

Problema 7.4. Sea $N = p_1^{n_1} p_2^{n_2} \cdots p_k^{n_k} \geq 2$ impar, con los p_i primos distintos y $n_i \geq 1$.

- Demostrar que $a \in \mathbb{Z}/N\mathbb{Z}$ es una raíz n-ésima primitiva de la unidad si y solo si $a^n \equiv 1 \pmod{N}$ y $\gcd(a^m - 1, N) = 1$ para $m = 1, \ldots, n-1$.
- Demostrar que una condición necesaria y suficiente para la existencia de raíces n-ésimas primitivas de la unidad en $\mathbb{Z}/N\mathbb{Z}$ es que $n \mid \gcd(p_1 - 1, p_2 - 1, \ldots, p_k - 1)$.

Problema 7.5. Sea $A = \mathbb{Z}/p\mathbb{Z}$ con $p = 3 \cdot 2^{189} + 1$ el primo del problema 4.12. Encontrar un elemento $\xi \in A$ de orden multiplicativo 2^{20} y usarlo para implementar una función para multiplicar polinomios $f, g \in A[x]$ de grados menores que 500000. Discutir como utilizar esa función para multiplicar polinomios en $\mathbb{Z}[x]$ si se sabe que sus coeficientes están en $[-2^{84}, 2^{84}]$.

Problema 7.6. Sea R un dominio íntegro (en el que $2 \cdot 1_R \neq 0_R$). Sean $n, m \geq 1$ enteros y sea u una variable.

- Demostrar que
$$\gcd(u^n - 1, u^m - 1) = u^{\gcd(n,m)} - 1$$
$$\gcd(u^n + 1, u^m - 1) = \frac{u^{\gcd(2n,m)} - 1}{u^{\gcd(n,m)} - 1}$$
en el anillo de polinomios $R[u]$.

- Demostrar que $[u] \in R[u]/\langle u^n + 1\rangle$ es una raíz $2n$-ésima primitiva de la unidad si y sólo si n es una potencia de 2.

- Demostrar que si n es primo, entonces $[2] \in \mathbb{Z}/(2^n - 1)\mathbb{Z}$ es una raíz n-ésima primitiva. ¿Sigue siendo esto válido si se cambian los 2 por 3?

- Demostrar que si n es una potencia de 2, entonces $[2] \in \mathbb{Z}/(2^n + 1)\mathbb{Z}$ es una raíz $2n$-ésima primitiva de la unidad. ¿Sigue siendo esto válido si se cambian los 2 por 3?

Ahora veremos el método de Schönhage-Strassen para multiplicar polinomios en $R[x]$. Este método toma un entero $n = 2^k \geq 1$ y dos polinomios $f, g \in R[x]$ de grado menor que n y y calcula $[f] \cdot [g]$ en $R[x]/\langle x^n + 1\rangle$, es decir, obtiene un polinomio $h \in R[x]$ tal que $h \equiv fg \pmod{x^n + 1}$ y $\deg(h) < n$. En el caso de que $\deg(f)+\deg(g) < n$, entonces $h = fg$ en $R[x]$, es decir, el método de Schönhage-Strassen puede utilizarse para multiplicar polinomios como caso particular.

Sean $n_1 = 2^{k_1} \geq 1$ y $n_2 = 2^{k_2} \geq 1$ tales que $n = n_1 n_2$ y $2n_1 \geq n_2$, es decir, $k_1 + k_2 = k$ y $1 + k_1 \geq k_2$. Tomamos a los polinomios f y g y los reescribimos del siguiente modo

$$\begin{aligned} f &= f_0(x) + f_1(x)x^{n_1} + f_2(x)x^{2n_1} + \cdots + f_{n_2-1}(x)x^{(n_2-1)n_1}, \\ g &= g_0(x) + g_1(x)x^{n_1} + g_2(x)x^{2n_1} + \cdots + g_{n_2-1}(x)x^{(n_2-1)n_1}, \end{aligned}$$

donde $\deg(f_i), \deg(g_i) < n_1$ para todo $i = 0, 1, \ldots, n_2 - 1$. Sea $A = R[u]/\langle u^{2n_1} + 1\rangle$. Definimos los polinomios $\tilde{f}, \tilde{g} \in A[y]$ del siguiente modo

$$\begin{aligned} \tilde{f} &= [f_0(u)] + [f_1(u)]y + [f_2(u)]y^2 + \cdots + [f_{n_2-1}(u)]y^{n_2-1}, \\ \tilde{g} &= [g_0(u)] + [g_1(u)]y + [g_2(u)]y^2 + \cdots + [g_{n_2-1}(u)]y^{n_2-1}. \end{aligned}$$

Calculamos el $\tilde{h} = \tilde{f}\tilde{g} \bmod y^{n_2} + 1$, haciendo una negaconvolución con $\xi_{2n_2} = [u^{2n_1/n_2}]$. Esto es posible, ya que $[u]$ es una raíz $4n_1$-ésima primitiva de la unidad de A y por lo tanto $[u^{2n_1/n_2}]$ es una raíz $2n_2$-ésima primitiva de la unidad. Esto requiere $O(n_2 \log(n_2))$ operaciones de suma, resta y multiplicación por potencias de $[u]$ en A, cada una de las cuales tiene una complejidad aritmética $O(n_1)$ en R. Además se necesitan hacer n_2 productos en A, para lo que habrá que invocar al método recursivamente. Finalmente, hay que reconstruir el resultado final en $R[x]$ teniendo

$$\tilde{h} = \tilde{f}\tilde{g} \bmod y^{n_2} + 1 = [h_0(u)] + [h_1(u)]y + [h_2(u)]y^2 + \cdots + [h_{n_2-1}(u)]y^{n_2-1} \tag{2}$$

con los $\deg(h_i) < 2n_1$ para $i = 0, 1, \ldots, n_2 - 1$. Afirmamos que será

$$h = fg \bmod x^n + 1 = h_0(x) + h_1(x)x^{n_1} + h_2(x)x^{2n_1} + \cdots + h_{n_2-1}(x)x^{(n_2-1)n_1} \bmod x^n + 1 \tag{3}$$

y que para hacer esto sólo se necesitan hacer n sumas y restas en R. En total, el método requiere

$$\mathrm{MultSS}(n) \leq Cn\log(n_2) + n_2\,\mathrm{MultSS}(2n_1) \tag{4}$$

operaciones aritméticas en R, para una cierta constante $C > 0$.

Cuando k es par, podemos tomar $k_1 = k_2 = \frac{k}{2}$, y en caso contrario, es decir, cuando k es impar, $k_1 = \frac{k-1}{2}$ y $k_2 = \frac{k+1}{2}$. Haciendo esto, y resolviendo la recursión (4) nos queda que

$$\mathrm{MultSS}(n) \leq Dn\log(n)\log(\log(n)) \tag{5}$$

para una cierta constante $D > 0$. En efecto, suponiendo que ya tuvieramos la desigualdad (5) demostrada para los valores menores que n, podemos usar (4) y nos queda

$$\begin{aligned} \mathrm{MultSS}(n) &\leq Cn\log(n_2) + n_2 D 2n_1 \log(2n_1)\log(\log(2n_1)) \\ &\leq Cn\log(\sqrt{2n}) + 2Dn\log(2\sqrt{n})\log(\log(2\sqrt{n})) \\ &= \frac{C}{2}n\log(n) + \frac{C}{2}n + Dn\log(n)\log\left(\frac{1}{2}\log(n)\right) + Dn\log\left(\frac{1}{2}\log(n)\right) \\ &= Dn\log(n)\log(\log(n)) - \left(D - \frac{C}{2}\right)n - \left(D - \frac{C}{2}\right)n\log(n) + Dn\log(\log(n)) \\ &\leq Dn\log(n)\log(\log(n)) \end{aligned}$$

para n suficientemente grande (suponiendo que $D > C/2$).

Solo resta probar que la fórmula (3) es correcta. Para eso, partimos de la ecuación (2) que nos dice que

$$[h_i(u)] = \sum_{j+k=i} [f_j(u)][g_k(u)] - \sum_{j+k=i+n_2} [f_j(u)][g_k(u)]$$

para todo $i = 0, 1, \dots, n_2 - 1$. Como $\deg(f_j), \deg(g_k) < n_2$ y estamos tomando clases módulo $u^{2n_2} + 1$, resulta que

$$h_i(u) = \sum_{j+k=i} f_j(u)g_k(u) - \sum_{j+k=i+n_2} f_j(u)g_k(u)$$

para todo $i = 0, 1, \dots, n_2 - 1$. Por lo tanto,

$$\begin{aligned}\sum_{i=0}^{n_2-1} h_i(x)x^{in_1} &= \sum_{0\le j+k<n_2} f_j(x)g_k(x)x^{(j+k)n_1} - \sum_{n_2\le j+k<2n_2} f_j(x)g_k(x)x^{(j+k-n_2)n_1} \\ &\equiv \sum_{0\le j+k<2n_2} f_j(x)g_k(x)x^{(j+k)n_1} \equiv fg \equiv h \pmod{x^n+1}.\end{aligned}$$

Eso concluye la demostración de la correctitud del método.

Una de las principales ventajas del método de Schönhage-Strassen para multiplicar polinomios en $R[x]$ es que no se necesita que R tenga raíces de la unidad. El precio que se paga, comparado con la multiplicación en $\mathbb{C}[x]$ usando transformadas de Fourier, es un factor $\log(\log(n))$ en la complejidad. Por otra parte, el cálculo que hicimos sobre la complejidad sólo tiene en cuenta la cantidad de operaciones aritméticas en R y no el tamaño de sus operandos. Esto lo hace ideal para trabajar en anillos R en los que el tamaño de sus elementos sea constante (los cuerpos finitos).

Problema 7.7. Implementar la función `mult_pol_mod(f,g,p)` que tome un primo $p \neq 2$ y dos polinomios $f, g \in (\mathbb{Z}/p\mathbb{Z})[x]$, representados por la lista de sus coeficientes, y que calcule su producto. Para esto, implementar una función recursiva `mult_ss_mod(f,g,k,p)` que tome $[f], [g] \in (\mathbb{Z}/p\mathbb{Z})[x]/\langle x^{2^k}+1\rangle$ y calcule su producto aplicando el método de Schönhage-Strassen.

La misma idea que vimos para multiplicar polinomios en $R[x]$ puede usarse para multiplicar enteros. Se puede probar que es posible calcular productos en $\mathbb{Z}/\langle 2^n + 1\rangle$ con $n = 2^k$ en $O(n\log(n)\log(\log(n)))$ operaciones binarias. Sólo hay que tener cuidado de tomar $A = \mathbb{Z}/\langle 2^{2n_1+\log(n_2)} + 1\rangle$ para que la demostración (2)$\Rightarrow$(3) funcione bien. Haciendo esto queda una recursión similar a (4),

$$\mathrm{MultSS}_{\mathbb{Z}}(n) \le Cn\log(n_2) + n_2\,\mathrm{MultSS}_{\mathbb{Z}}(2n_1 + \log(n_2))$$

pero su solución sigue siendo de tipo $O(n\log(n)\log(\log(n)))$.

Problema 7.8. Sean $a, b \in [0, 2^{2^{20}}]$ enteros. Mostrar como el método de Schönhage-Strassen reduce el problema de calcular ab a hacer 2^{10} productos de enteros en $[0, 2^{2^{10}+10}]$. Implementar la función `mult_enteros_ss_20(a,b)` que aplique ese procedimiento. Esta función NO debe ser recursiva e internamente sólo puede usar el operador de multiplicación 2^{10} veces y con enteros en el rango $[0, 2^{1034}]$. Todas las demás operaciones deben ser `+`, `-`, `<<` y `>>`.

El método de Cooley-Tuckey para calcular DFT_n con $n = 2^k$ puede interpretarse como el producto de matrices

$$D(\xi_n) = \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & E(\xi_n) \end{bmatrix} \cdot \begin{bmatrix} D(\xi_{n/2}) & 0 \\ 0 & D(\xi_{n/2}) \end{bmatrix} \cdot P_n$$

donde P_n es la matriz de permutación que manda los índices pares $0, 2, 4, \dots$ en $0, 1, 2, \dots$ y los impares $1, 3, 5, \dots$ en $\frac{n}{2}, \frac{n}{2} + 1, \dots$ y $E(\xi_n)$ es la matriz diagonal de las primeras $\frac{n}{2}$ potencias de ξ_n. Dado que la matriz $D(\xi_n)$ es simétrica, también tenemos que

$$D(\xi_n) = P_n^{-1} \cdot \begin{bmatrix} D(\xi_{n/2}) & 0 \\ 0 & D(\xi_{n/2}) \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & E(\xi_n) \end{bmatrix} \cdot \begin{bmatrix} I & I \\ I & -I \end{bmatrix}.$$

Aplicando esta fórmula recursivamente, nos queda

$$D(\xi_n) = P_n^{-1} \cdot \begin{bmatrix} P_{n/2}^{-1} & 0 \\ 0 & P_{n/2}^{-1} \end{bmatrix} \cdot \begin{bmatrix} D(\xi_{n/4}) & 0 & 0 & 0 \\ 0 & D(\xi_{n/4}) & 0 & 0 \\ 0 & 0 & D(\xi_{n/4}) & 0 \\ 0 & 0 & 0 & D(\xi_{n/4}) \end{bmatrix} \cdot$$
$$\cdot \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & E(\xi_{n/2}) & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & E(\xi_{n/2}) \end{bmatrix} \cdot \begin{bmatrix} I & I & 0 & 0 \\ I & -I & 0 & 0 \\ 0 & 0 & I & I \\ 0 & 0 & I & -I \end{bmatrix} \cdot \begin{bmatrix} I & 0 \\ 0 & E(\xi_n) \end{bmatrix} \cdot \begin{bmatrix} I & I \\ I & -I \end{bmatrix} \tag{6}$$

y así podríamos continuar k veces hasta que quede una diagonal de $D(\xi_1)$, es decir, la matriz identidad.

Esa fórmula para $D(\xi_n)$ permite implementar una versión iterativa de la transformada rápida de Fourier que opera "in-place".

```python
def fft_in_place(p, xi):
    n = len(p)//2
    d = 1
    while n >= 1:
        for i in range(d):
            for j in range(n):
                x = p[2*n*i+j]
                y = p[2*n*i+j+n]
                p[2*n*i+j]  = x+y
                p[2*n*i+j+n] = x-y
        for i in range(d):
            for j in range(n):
                p[2*n*i+j+n] *= xi**(j*d)
        n //= 2
        d *= 2

def ifft_in_place(p, xi):
    d = len(p)//2
    n = 1
    while d >= 1:
        for i in range(d):
            for j in range(n):
                p[2*n*i+j+n] /= xi**(j*d)
        for i in range(d):
            for j in range(n):
                x = p[2*n*i+j]
                y = p[2*n*i+j+n]
                p[2*n*i+j]  = 0.5*(x+y)
                p[2*n*i+j+n] = 0.5*(x-y)
        n *= 2
        d //= 2
```

Programa 2: fft2.py

Por supuesto, las funciones `fft_in_place(p, xi)` y `ifft_in_place(p, xi)` no devuelven ningún resultado, ya que modifican a las propias entradas de la lista `p`. Es importante notar que estas funciones producen la transforma de Fourier *desordenada*. El código no aplica ninguna de las permutaciones que aparecen en la fórmula (6).

Problema 7.9. Demostrar que la entrada i-ésima del resultado de aplicar `fft_in_place(p, xi)` es la coordenada $\sigma(i)$-ésima de $DFT_n(p)$, donde $\sigma(i)$ es el número cuya representación binaria es la de i escrita al revés.