Esercizi sugli aspetti computazionali della Trasformata Discreta di Fourier

Exercise 1.1 Descrivere e analizzare un algoritmo per moltiplicare due polinomi di grado limitato da m e n (con $m \ll n$) in tempo $O(n \log m)$. (*Suggerimento:* Si determini una scomposizione del polinomio di limite di grado maggiore in polinomi di grado limitato da m moltiplicati per una potenza dell'indeterminata.)

Answer: Let $A(x)$ and $B(x)$ be two polynomials of degree-bound m and n, with $m \ll n$, represented by the two coefficient vectors $\boldsymbol{a} \in \boldsymbol{C}^m$ and $\boldsymbol{b} \in \boldsymbol{C}^n$, respectively. Without loss of generality, let us assume that n is a multiple of m, since otherwise we can pad $\boldsymbol{b}$ with $m - (n \bmod m) \leq m \ll n$ zeroes in its most significant components to take its size to the next multiple of m, which yields a new vector of (much) less than twice the original size of $\boldsymbol{b}$. For $0 \leq k \leq n/m - 1$, let $\boldsymbol{b}_k = (b_{km}, b_{km+1}, \ldots, b_{(k+1)m-1}) \in \boldsymbol{C}^m$ be the k-th subvector of m consecutive components of $\boldsymbol{b}$, and let $B_k(x)$ be the polynomial of degree-bound m represented by $\boldsymbol{b}_k$. We have that

$$\boldsymbol{b} = (\boldsymbol{b}_{n/m-1} | \boldsymbol{b}_{n/m-2} | \ldots | \boldsymbol{b}_0)$$

and

$$\begin{aligned} B(x) &= \sum_{i=0}^{n-1} b_i x^i \\ &= \sum_{k=0}^{n/m-1} \sum_{j=0}^{m-1} b_{km+j} x^{km+j} \\ &= \sum_{k=0}^{n/m-1} x^{km} \sum_{j=0}^{m-1} b_{km+j} x^j \\ &= \sum_{k=0}^{n/m-1} x^{km} B_k(x), \end{aligned}$$

Using the above relation, we obtain:

$$\begin{aligned} A(x)B(x) &= A(x)\sum_{k=0}^{n/m-1} x^{km}B_k(x) \\ &= \sum_{k=0}^{n/m-1} x^{km}A(x)B_k(x) \end{aligned}$$

Thus, we have expressed the product $A(x)B(x)$ as a function of the products $A(x)B_k(x)$, for $0 \leq k \leq n/m-1$. The k-th such product is multiplied by the km-th power of the indeterminate and then summed to the other products. Observe that multiplying a polynomial by a power x^r of the indeterminate simply adds a term r to the degree of each of its monomials. Special care must then be exercised when summing the coefficients of the m/n polynomials of the above summation with respect to the right degree of their monomials. In particular, for $0 \leq k \leq n/m-1$ and $0 \leq i \leq 2m-2$, the degree of the i-th (least significant) monomial of $x^{km}A(x)B_k(x)$ will be $km+i$.

Let us assume that we have available the subroutine LIN_CONV($\boldsymbol{x}, \boldsymbol{y}$) (seen in class) that computes the linear convolution $\boldsymbol{x} \star \boldsymbol{y}$ of two vectors of the same size m in time $\Theta(m \log m)$ thorugh the linear convolution theorem. Putting it all together, we obtain the desired multiplication algorithm, working under the static constraint on the degree bounds m and n discussed above.

```
UNEVEN_LIN_CONV(a, b)
m ← a.len; n ← b.len
⋆ Let b = (b_{n/m-1}|b_{n/m-2}|...|b_0) ⋆
c ← 0_{m+n-1}
for k ← 0 to n/m − 1 do
    z ← LIN_CONV(a, b_k)
    for i ← 0 to 2m − 2 do
        c_{km+i} ← c_{km+i} + z_i
return c
```

□

The correctness of UNEVEN_LIN_CONV immediately follows from the above discussion. As for its running time, we have that the n/m calls to LIN_CONV account for a total of $\Theta((n/m)m \log m) = \Theta(n \log m)$ arithmetic operations between complex numbers, while the additions needed to obtain the final coefficients add a total of $(n/m)(2m-1) = \Theta(n)$ scalar sums, for a total time $T(m,n) = \Theta(n \log m)$.

Exercise 1.2 Si consideri l'operazione di convoluzione lineare $\boldsymbol{u} \star \boldsymbol{x}$, con entrambe le sequenze di lunghezza n e con $\boldsymbol{u} = (1, 1, \ldots, 1)$. Si sviluppi un algoritmo iterativo per eseguire tale operazione in tempo $O(n)$. (*Suggerimento:* Si determini una ricorrenza sulle componenti della convoluzione.)

Answer: Let $\boldsymbol{c} = \boldsymbol{u} \star \boldsymbol{x} = \boldsymbol{x} \star \boldsymbol{u}$. By the definition of linear convolution we have that for $0 \leq i \leq 2n-2$,

$$c_i = \sum_{j=\max\{0,i-n+1\}}^{\min\{i,n-1\}} x_j = \begin{cases} \sum_{j=0}^{i} x_j & 0 \leq i \leq n-1 \\ \sum_{j=i-n+1}^{n-1} x_j & n \leq i \leq 2n-2. \end{cases}$$

The above expression immediateley yields the following two recurrences:

$$c_i = \begin{cases} x_0 & i = 0 \\ c_{i-1} + x_i & 0 < i \leq n-1 \end{cases} \quad \text{and } c_i = \begin{cases} x_{n-1} & i = 2n-2 \\ c_{i+1} + x_{i-n+1} & 2n-2 > i \geq n \end{cases}.$$

The code immediately follows.

```
UNIT_CONV(x)
n ← x.len
c_0 ← x_0
for i ← 1 to n − 1 do
    c_i ← c_{i−1} + x_i
c_{2n−2} ← x_{n−1}
for i ← 2n − 3 downto n do
    c_i ← c_{i+1} + x_{i−n+1}
return c
```

The above algorithm computes the desired convolution by executing exactly one scalar sum for each but two of the $2n-1$ components of the linear convolution $\boldsymbol{u} \star \boldsymbol{x}$. Its running time is therefore $T_{\text{UC}}(n) = 2n - 3$. □

Exercise 1.3 Siano k e n potenze di due, con $1 \leq k \leq n$. Un vettore $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$ si dice (n,k)-*padded* se $x_i = 0$ per $k \leq i \leq n-1$.

Punto 1. Si fornisca lo pseudocodice e si provi la correttezza di un algoritmo PADDED_FFT$(\boldsymbol{x}, k)$, ottenuto modificando l'algoritmo ricorsivo FFT visto in classe, per il caso di vettori (n,k)-padded. (*Suggerimento*: la ricorsione deve essere in funzione del parametro k, con caso di base $k = 1$.)

Punto 2 Si imposti e si risolva la ricorrenza sul numero di operazioni aritmetiche tra scalari complessi $T_{\mathrm{PF}}(n,k)$ effettuate dall'algoritmo sviluppato al Punto 1.

Answer: Point 1 We first observe that when $k = 1$, an (n,k)-padded vector $\boldsymbol{x}$ has only its first component (possibly) different from zero. In this case, we can compute $DFT_n(\boldsymbol{x})$ directly as $F_n \cdot \boldsymbol{x} = (x_0, x_0, \ldots, x_0)$. When $k > 1$, let $\boldsymbol{x}^{[0]} = (x_0, x_2, \ldots, x_{n-2})$ and $\boldsymbol{x}^{[1]} = (x_1, x_3, \ldots, x_{n-1})$ be the two $n/2$-vectors containing, respectively, the even-indexed and odd-indexed components of $\boldsymbol{x}$. The straightforward but crucial property upon which we will base our recursive strategy is the following: if $\boldsymbol{x}$ is (n,k)-padded, then both $\boldsymbol{x}^{[0]}$ and $\boldsymbol{x}^{[1]}$ are $(n/2, k/2)$-padded. Recall that the FFT algorithm computes $DFT_n(\boldsymbol{x})$ by recursively computing $DFT_{n/2}(\boldsymbol{x}^{[0]})$ and $DFT_{n/2}(\boldsymbol{x}^{[1]})$ and then performing another $\Theta(n)$ additional operations. When invoked on an (n,k)-padded vector, such recursive calls are needed only if $k > 1$. Otherwise, $DFT_n(\boldsymbol{x})$ is obtained directly. The algorithm follows.

PADDED_FFT($\boldsymbol{x}, k$)
$n \leftarrow \mathrm{length}(\boldsymbol{x})$
if $k = 1$
  then for $j \leftarrow 0$ **to** $n-1$ **do** $y_j \leftarrow x_0$
    return $\boldsymbol{y}$
$\boldsymbol{x}^{[0]} \leftarrow (x_0, x_2, \ldots, x_{n-2})$
$\boldsymbol{x}^{[1]} \leftarrow (x_1, x_3, \ldots, x_{n-1})$
$\boldsymbol{y}^{[0]} \leftarrow$ PADDED_FFT($\boldsymbol{x}^{[0]}, k/2$)
$\boldsymbol{y}^{[1]} \leftarrow$ PADDED_FFT($\boldsymbol{x}^{[1]}, k/2$)
$ON \leftarrow e^{2\pi i/n}$; $ONJ \leftarrow 1$
for $j \leftarrow 0$ **to** $n/2 - 1$ **do**
  $y_j \leftarrow y_j^{[0]} + ONJ \cdot y_j^{[1]}$
  $y_{j+n/2} \leftarrow y_j^{[0]} - ONJ \cdot y_j^{[1]}$
  $ONJ \leftarrow ONJ * ON$
return $\boldsymbol{y}$

The correctness of the above algorithm follows from the correctness of the FFT algorithm and the observations made above.

Point 2 We can write the following recurrence in n and k:

$$T_{\mathrm{P}}(n,k) = \begin{cases} 0, & k = 1, \\ 2 \cdot T_{\mathrm{P}}\left(\frac{n}{2}, \frac{k}{2}\right) + c \cdot n, & 1 < k \leq n \end{cases}$$

where c is a fixed positive constant. By unfolding, we obtain:

$$\begin{aligned}
T_{\mathrm{P}}(n,k) &= 2T_{\mathrm{P}}\left(\frac{n}{2},\frac{k}{2}\right)+cn \\
&= 2^2T_{\mathrm{P}}\left(\frac{n}{2^2},\frac{k}{2^2}\right)+cn+cn \\
&\vdots \\
&= 2^iT_{\mathrm{P}}\left(\frac{n}{2^i},\frac{k}{2^i}\right)+c\cdot i\cdot n \\
&= 2^{\log k}T_{\mathrm{P}}\left(\frac{n}{2^{\log k}},\frac{k}{2^{\log k}}\right)+cn\log k \\
&= kT_{\mathrm{P}}\left(\frac{n}{k},1\right)+cn\log k \\
&= cn\log k.
\end{aligned}$$

Therefore, $T_{\mathrm{P}}(n)=\Theta(n\log k)$. □

Exercise 1.4 Siano k,n potenze di due, con $1\le k<n$. Dato un vettore $\boldsymbol{A}\in\boldsymbol{C}^k$, si consideri il vettore a n componenti $\boldsymbol{B}\in\boldsymbol{C}^n$ ottenuto concatenando $\boldsymbol{A}$ con se stesso per n/k volte:

$$\boldsymbol{B}=(\underbrace{\boldsymbol{A}|\boldsymbol{A}|\ldots|\boldsymbol{A}}_{n/k\ \text{volte}}).$$

Si discuta di come ottenere $DFT_n(\boldsymbol{B})$ in tempo $\Theta((n/k)\log(n/k))$. Nell'analisi di complessità, si considerino di costo unitario e non nullo solo le operazioni aritmetiche tra numeri complessi. (*Suggerimento:* Si ricordi che $\boldsymbol{B}$ ricalca la struttura della trasformata di un vettore (n,k)-sparso (vista a lezione) e si utilizzi la relazione tra DFT_n^{-1} e DFT_n^R.)

Answer: Let $\boldsymbol{a}=(a_0,a_1,\ldots a_{k-1})=DFT_n^{-1}(\boldsymbol{A})$, and let $\boldsymbol{x}=(a_0,\underbrace{0,\ldots,0}_{n/k-1},a_1,\underbrace{0,\ldots,0}_{n/k-1},\ldots a_{k-1},\underbrace{0,\ldots 0}_{n/k-1})$. Recall that in class we showed that

$$DFT_n(\boldsymbol{x})=(\underbrace{\boldsymbol{A}|\boldsymbol{A}|\ldots|\boldsymbol{A}}_{n/k})=\boldsymbol{B}.$$

Therefore, $DFT_n^{-1}(\boldsymbol{B})=(1/n)DFT_n^R(\boldsymbol{B})=\boldsymbol{x}$, whence

$$DFT_n(\boldsymbol{B})=n\boldsymbol{x}^R=.(na_0,\underbrace{0,\ldots,0}_{n/k-1},na_{k-1},\underbrace{0,\ldots,0}_{n/k-1},\ldots na_1,\underbrace{0,\ldots 0}_{n/k-1})$$

The algorithm immediately follows.

$$\begin{array}{l}
\text{REP_FFT}(\boldsymbol{A}, n)\ \{\text{computes } DFT_n(\overbrace{\boldsymbol{A}|\boldsymbol{A}|\ldots|\boldsymbol{A}}^{n/k})\} \\
k \leftarrow \boldsymbol{A}.len \\
\boldsymbol{a} \leftarrow \text{INV_FFT}(\boldsymbol{A}) \\
\textbf{return } (na_0, \underbrace{0, \ldots, 0,}_{n/k-1} na_{k-1}, \underbrace{0, \ldots, 0,}_{n/k-1} \ldots na_1, \underbrace{0, \ldots 0}_{n/k-1})
\end{array}$$

The running time of the above algorithm (in terms of arithmetic operations between complex scalars) is clearly $\Theta(k \log k)$.

It is interesting to observe that if $\boldsymbol{a} = DFT_n^{-1}(\boldsymbol{A})$, then $(na_0, na_{k-1}, \ldots, na_1) = DFT_n(\boldsymbol{A})$. Therefore we can simplify the above algorithm as follows.

$$\begin{array}{l}
\text{REP_FFT}(\boldsymbol{A}, n) \\
k \leftarrow \boldsymbol{A}.len \\
\boldsymbol{b} \leftarrow \text{FFT}(\boldsymbol{A}) \\
\textbf{return } (b_0, \underbrace{0, \ldots, 0,}_{n/k-1} b_1, \underbrace{0, \ldots, 0,}_{n/k-1} \ldots b_{k-1}, \underbrace{0, \ldots 0}_{n/k-1})
\end{array}$$

□

Exercise 1.5

Punto 1. Si fornisca un algoritmo divide-and-conquer per ottenere la potenza k-sima z^k di un dato numero complesso z eseguendo $\Theta(\log k)$ moltiplicazioni complesse (*Suggerimento:* Si ottenga z^k in funzione di $z^{\lfloor k/2 \rfloor}$.)

Punto 2. Siano ora $k, n > 0$. Dato il vettore dei coefficienti $\boldsymbol{a} \in \boldsymbol{C}^n$ di un polinomio $A(x)$ di grado limitato da n , si fornisca un algoritmo che restituisca il vettore dei coefficienti del polinomio $p(x) = (A(x))^k$.

Answer:

Point 1. We have that $z^0 = 1$ and $z^1 = z$. Consider now the case $k > 1$. If k is even, then $z^k = (z^{k/2})^2 = (z^{\lfloor k/2 \rfloor})^2$. If k is odd, then $z^k = (z^{(k-1)/2})^2 \cdot z = (z^{\lfloor k/2 \rfloor})^2 \cdot z$. The above substructure property indicates that we can obtain z^k from $z^{\lfloor k/2 \rfloor}$ by performing at most two extra multiplications (in case k is odd). The algorithm immediately follows.

```
POWER(z, k)
if k = 0 then return 1
if k = 1 then return z
x ← POWER(z, ⌊k/2⌋)
y ← x · x
if EVEN(k) then return y
           else  return y · z
```

The correctness of the above algorithm follows immediately from the substructure property. The worst-case running time in terms of number of multiplications obeys the recurrence $T(k) = T(k/2) + 2$ which can be studied using the Master Theorem observing that the threshold function $k^{\log_2 1} = k^0 = 1$ is of the same order as $w(k) = 2$. Therefore, $T(k) = \Theta(\log k)$.

Point 2. Assume that we have available the algorithm LIN_CONV($\boldsymbol{a}, \boldsymbol{b}$) which returns the linear convolution $\boldsymbol{a} \star \boldsymbol{b}$ of two vectors of respective lengths m and n in time $\Theta(\max\{m, n\} \log(\max\{m, n\}))$ (the algorithm follows immediately from the convolution theorem with the right amount of padding added to the two vectors). We can straightforwardly modify the algorithm POWER developed in Point 1 to yield polynomial exponentiation $(A(x))^k$, with $A(x) \equiv \boldsymbol{a}$, by substituting the scalar multiplications with calls to LIN_CONV as follows.

```
POLY_POWER(a, k)
if k = 0 then return (1) { constant polynomial }
if k = 1 then return a
b ← POLY_POWER(a, ⌊k/2⌋)
c ← LIN_CONV(b, b)
if EVEN(k) then return c
           else  return LIN_CONV(c, a)
```

To analyze the running time of the above algorithm, we have to take into account that the complexity of LIN_CONV depends on the maximum length of its two operands. Let $T(n, k)$ be the worst-case running time (in terms of arithmetic operations between scalars) of POLY_POWER($\boldsymbol{a}, k$), where n is the length of $\boldsymbol{a}$ (i.e., the degree-bound of the polynomial that has to be exponentiated). We have that $T(n, 0) = T(n, 1) = 0$, and $T(n, k) = T(n, \lfloor k/2 \rfloor) + cnk \log_2(nk)$, for some constant c, since the length of operand $\boldsymbol{b}$ is $\Theta(nk)$. Thus, $T(n, k) \geq cnk \log_2(nk)$. Recalling that for $f(k) = \lfloor k/2 \rfloor$ we have that $f^{(i)}(k) =$

$\lfloor k/2^i \rfloor$ and $f^\star(n,2) \leq \lfloor \log_2 k \rfloor$, we obtain that

$$\begin{aligned} T(n,k) &\leq \sum_{i=0}^{\lfloor \log_2 k \rfloor} (cnk/2^i)\log(nk/2^i) \\ &\leq \sum_{i=0}^{\lfloor \log_2 k \rfloor} (cnk/2^i)\log(nk) \\ &\leq 2cnk\log_2(nk). \end{aligned}$$

Thus, $T(n,k) = \Theta(nk\log(nk))$.

An alternative approach to polynomial exponentiation is to evaluate $A(x)$ on a number of points which is as large as the degree-bound of $(A(x))^k$, that is, $(n-1)k+1$. Once this extended point-representation of $A(x)$ is available, we can obtain a point representation of $(A(x))^k$ by simply exponentiating the single evaluations using Algorithm POWER developed in Point 1, and then interpolating to obtain the coefficient vector of $(A(x))^k$. Extended evaluation and interpolation can be obtained using FFT and INV_FFT on a sufficiently large number of points that is also a power of two, e.g., the first power of two larger or equal to $(n-1)k+1$. The algorithm follows immediately.

POLY_POWER2($\boldsymbol{a}, k$)
$n \leftarrow \boldsymbol{a}.len$
$h \leftarrow 2^{\lceil \log_2((n-1)k+1) \rceil}$
$\boldsymbol{y} \leftarrow \text{FFT}(\boldsymbol{a}|\boldsymbol{0}_{h-n})$
for $i \leftarrow 0$ **to** $h-1$ **do**
    $y_i \leftarrow \text{POWER}(y_i, k)$
$\boldsymbol{z} \leftarrow \text{INV\_FFT}(\boldsymbol{y})$
return $(z_0, \ldots z_{(n-1)k})$

The running time of POLY_POWER2 is determined by the two calls to FFT and INV_FFT on vectors of size $h = \Theta(nk)$, and the h calls to POWER to compute h k-th powers of complex numbers, for a total running time of $\Theta(h\log h + h\log k) = \Theta(nk\log(nk) + nk\log k) = \Theta(nk\log(nk))$. □

Exercise 1.6 Sia n una potenza di due. Usando la FFT, progettare e analizzare un algoritmo che, dato in ingresso n, produca in uscita il vettore $C[k] = \binom{n-1}{k}$, con $0 \leq k \leq n-1$ eseguendo $O(n\log n)$ operazioni tra scalari complessi. (*Suggerimento:* Si consideri l'espansione di Newton del polinomio $p(x) = (x+1)^{n-1}$ ....)

Answer: By Newton's binomial expansion we have that

$$(x+1)^{n-1} = \sum_{k=0}^{n-1} \binom{n-1}{k} x^k,$$

therefore we can obtain the desired vector as the coefficient vector of the polynomial $(x+1)^{n-1}$. Using the subroutine POLY_POWER developed in the previous exercise, the algorithm follows immediately.

BIN_COEFFICIENTS(n)
$\boldsymbol{a} \leftarrow (1,1)$ { coefficient vector of (x+1) }
return POLY_POWER($\boldsymbol{a}, n-1$)

The running time of the above algorithm is clearly $\Theta(n \log n)$. □

Exercise 1.7 Sia $n > 0$. Per $0 \leq j \leq n-1$, sia $\boldsymbol{F}_n^j$ la j-sima colonna della matrice di Fourier di ordine n. Si determinino le componenti di $DFT_n(\boldsymbol{F}_n^j)$ e si fornisca lo pseudocodice di un algoritmo per calcolarle che non esegue alcuna operazione aritmetica tra scalari complessi.

Answer: Let us determine the components of $DFT_n(\boldsymbol{F}_n^j)$ analytically, for $0 \leq j \leq n-1$. We have:

$$\begin{aligned}
[DFT_n(\boldsymbol{F}_n^j)]_i &= \sum_{k=0}^{n-1} [F_n]_{ik} [\boldsymbol{F}_n^j]_k \\
&= \sum_{k=0}^{n-1} \omega_n^{ik} \omega_n^{kj} \\
&= \sum_{k=0}^{n-1} \omega_n^{(i+j)k}.
\end{aligned}$$

By the summation lemma, we have that $[DFT_n(\boldsymbol{F}_n^j)]_i = n$ when $(i+j) \bmod n = 0$, that is, $i = (n-j) \bmod n$, and 0 otherwise. As a consequence, $[DFT_n(\boldsymbol{F}_n^j)]$ is a vector with all null components but the $(n-j) \bmod n$-th component, equal to n. The required algorithm immediately follows.

FDFT(n, j)
$\boldsymbol{y} \leftarrow \boldsymbol{0}_n$
$y_{(n-j) \bmod n} \leftarrow n$
return $\boldsymbol{y}$

□

Exercise 1.8 Sia n una potenza di due e siano dati due insiemi $A, B \subseteq \{0, 1, 2, \ldots n-1\}$ rappresentati in ingresso per mezzo dei loro vettori caratteristici (cioè vettori binari $\boldsymbol{a}$ e $\boldsymbol{b} \in \{0,1\}^n$ con $a_i = 1 \Leftrightarrow i \in A$ e $b_j = 1 \Leftrightarrow j \in B$). Si voglia calcolare, per ogni intero i, $0 \leq i \leq 2n-2$, il numero z_i di coppie distinte $(a, b) \in A \times B$ tali che $a + b = i$.

Punto 1. Si riconduca il problema al calcolo di una opportuna convoluzione lineare.

Punto 2. Si fornisca lo pseudocodice dell'algoritmo CARTESIAN_SUM($\boldsymbol{a}, \boldsymbol{b}$) che risolve il problema in tempo $O(n \log n)$.

Answer:

Part (a) It is sufficient to compute

$$\boldsymbol{z} = (z_0, z_1, \ldots, z_{2n-2}) = \boldsymbol{a} \star \boldsymbol{b},$$

where $\boldsymbol{a} \star \boldsymbol{b}$ denotes the linear convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$. Indeed, by definition of linear convolution we have:

$$\begin{aligned} z_i &= \sum_{\substack{(j,k):j+k=i \\ 0 \leq j,k \leq n-1}} a_j b_k \\ &= |\{(j,k) \in A \times B : j + k = i\}|, \end{aligned}$$

since the only nonzero terms in the summation are all and only those with indices in $A \times B$.

Part (b) The algorithm is as follows:

CARTESIAN_SUM($\boldsymbol{a}, \boldsymbol{b}$)
$n \leftarrow$ length($\boldsymbol{a}$)
$\boldsymbol{fa} \leftarrow$ FFT($\boldsymbol{a}|\boldsymbol{0}_n$)
$\boldsymbol{fb} \leftarrow$ FFT($\boldsymbol{b}|\boldsymbol{0}_n$))
for $i \leftarrow 0$ **to** $2n-1$ **do**
    $zf_i \leftarrow fa_i \cdot fb_i$
$\boldsymbol{z} \leftarrow$ INV_FFT($\boldsymbol{zf}$)
return $(z_0, z_1, \ldots, z_{2n-2})$

The above algorithm executes in $\Theta(n \log n)$ time. □

Exercise 1.9

Punto 1. Sia $n > 0$. Si dimostri rigorosamente che per ogni vettore $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1}) \in \boldsymbol{C}^n$ si ha:

$$(F_n)^2\boldsymbol{x} = n \cdot \boldsymbol{x}^R,$$

dove $(F_n)^2 = F_n \times F_n$ denota il quadrato della matrice di Fourier di ordine n e $\boldsymbol{x}^R = (x_0, x_{n-1}, x_{n-2} \ldots, x_1)$ denota il *reverse* del vettore $\boldsymbol{x}$.

Punto 2. Utilizzando la relazione provata al Punto 1, si fornisca lo pseudocodice e si analizzi un algoritmo *divide-and-conquer* $\mathrm{KFT}(\boldsymbol{x}, k)$ che, dati in ingresso un vettore complesso $\boldsymbol{x} \in \boldsymbol{C}^n$, con n potenza di due e k un generico intero positivo o nullo, restituisca $\boldsymbol{y} = (F_n)^k\boldsymbol{x}$ eseguendo $T(n, k) = O(n(k + \log n))$ operazioni aritmetiche tra scalari complessi.
(*Suggerimento:* si osservi che per $k \geq 2$ vale $(F_n)^k\boldsymbol{x} = (F_n)^2((F_n)^{k-2}\boldsymbol{x}) \ldots)$

Answer: Point 1 In order to prove the stated relation, let $\boldsymbol{y} = F_n\boldsymbol{x}$. Then we have:

$$(F_n)^2\boldsymbol{x} = n\boldsymbol{x}^R \Leftrightarrow F_n\boldsymbol{y} = n\boldsymbol{x}^R \Leftrightarrow \boldsymbol{x}^R = \frac{1}{n}F_n\boldsymbol{y} \Leftrightarrow \boldsymbol{x} = \frac{1}{n}(F_n\boldsymbol{y})^R \Leftrightarrow \boldsymbol{x} = (F_n)^{-1}\boldsymbol{y},$$

where the (trivially true) last relation follows from the reduction of DFT_n^{-1} to DFT_n^R discussed in class.
Point 2 Let $\boldsymbol{x} \in \boldsymbol{C}^n$ and $k \geq 0$. In order to derive a divide-and-conquer algorithm, let us first discuss the base cases. If $k = 0$, we have $(F_n)^0 = I_n$, hence we return $\boldsymbol{x}$. If $k = 1$, we have $(F_n)^1 = F_n$, hence we return $F_n\boldsymbol{x}$ by calling $\mathrm{FFT}(\boldsymbol{x})$. When $k \geq 2$, we observe that $(F_n)^k\boldsymbol{x} = (F_n)^2\boldsymbol{z}$, where $\boldsymbol{z} = (F_n)^{k-2}\boldsymbol{x}$. We can then obtain $\boldsymbol{z}$ recursively and then obtain $(F_n)^k\boldsymbol{x}$ by applying the relation proved in Point 1, by first reversing $\boldsymbol{z}$ and then multiplying each of its components by n. The pseudocode follows.

$\mathrm{KFT}(\boldsymbol{x}, k)$
$n \leftarrow \mathrm{length}(\boldsymbol{x})$
if $(k = 0)$ **then return** $\boldsymbol{x}$
if $(k = 1)$ **then return** $\mathrm{FFT}(\boldsymbol{x})$
$\boldsymbol{z} \leftarrow \mathrm{KFT}(\boldsymbol{x}, k - 2)$
$y_0 \leftarrow n \cdot z_0$
for $i \leftarrow 1$ **to** $n - 1$ **do** $y_i \leftarrow n \cdot z_{n-i}$
return $\boldsymbol{y}$

The above algorithm makes use of the routine FFT developed in class, and its correctness follows from the previous discussion. The recurrence relation on the number of arithmetic operations between complex scalars performed by $\mathrm{KFT}(\boldsymbol{x}, k)$ is:

$$T(n,k) = \begin{cases} 0 & k = 0, \\ cn \log n & k = 1 \\ T(n, k-2) + n & k > 1, \end{cases}$$

for some constant c (determined by the call to FFT). The above relation unfolds to $T(n,k) = T(n, k-2i) + ni$, for $1 \leq i \leq \lfloor k/2 \rfloor$. For even k, we then obtain $T(n,k) = nk/2 + T(n,0) = nk/2$, while for odd k (the worst case) we get $T(n,k) = n\lfloor k/2 \rfloor + T(n,1) = n\lfloor k/2 \rfloor + cn \log n$. Therefore $T(n,k) = O(n(k + \log n))$.

Indeed, this is not the best algorithm for computing $(F_n)^k \boldsymbol{x}$, since it is not difficult to prove (reasoning along similar lines) that the following closed formula holds. Let $\boldsymbol{y} = F_n \boldsymbol{x}$. Then:

$$(F_n)^k \boldsymbol{x} = \begin{cases} n^{k/2} \cdot \boldsymbol{x} & \text{if } k \bmod 4 = 0, \\ n^{(k-1)/2} \cdot \boldsymbol{y} & \text{if } k \bmod 4 = 1, \\ n^{k/2} \cdot \boldsymbol{x}^R & \text{if } k \bmod 4 = 2, \\ n^{(k-1)/2} \cdot \boldsymbol{y}^R & \text{if } k \bmod 4 = 3. \end{cases}$$

Implementing the above formula, in the worst case ($k \bmod 4$ an odd number), we would only need to compute $n^{(k-1)/2}$ in $O(\log k)$ time, transform $\boldsymbol{x}$ in $O(n \log n)$ time and perform n additional scalar multiplications, for a total of $O(n \log n + \log k)$ time. □