Pontificia Universidad Católica de Chile
Facultad de Ingenieria
Departamento de Ciencias de la Computación
IIC2283 - Diseño y Análisis de Algoritmos

Ayudantía 12

30 de Noviembre, 2018

Problema 4

Dado un polinomio $A(x)$ con grado acotado por n, se define su i-ésima derivada como

$$A^{(t)}(x) = \begin{cases} A(x) & t = 0 \\ \frac{d}{dx}A^{(t-1)}(x) & 1 \leq t \leq n-1 \\ 0 & t \geq n \end{cases}$$

Desde la representación de coeficientes $(a_0, a_1, ..., a_{n-1})$ de $A(x)$ y un punto x_0, se desea determinar $A^{(t)}(x_0)$ para $t = 0, 1, ..., n-1$.

1. Dado coeficientes b_0, b_1, b_{n-1} tales que

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j$$

muestre como calcular $A^{(t)}(x_0)$ para $t = 0, 1, ..., n-1$ en tiempo $O(n)$.

Solución:

Tenemos que

$$\begin{aligned} A^{(0)}(x) &= \sum_{j=0}^{n-1} b_j (x - x_0)^j \\ A^{(0)}(x_0) &= 0! \cdot b_0 \\ A^{(1)}(x) &= \sum_{j=1}^{n-1} j \cdot b_j (x - x_0)^{j-1} \\ A^{(1)}(x_0) &= 1! \cdot b_1 \\ A^{(2)}(x) &= \sum_{j=2}^{n-1} j(j-1) \cdot b_j (x - x_0)^{j-2} \\ A^{(2)}(x_0) &= 2! \cdot b_2 \end{aligned}$$

Se puede demostrar fácilmente por inducción que este patrón continua, y que $A^{(t)}(x_0) = t! \cdot b_t$. Luego se pueden calcular todos los valores de $A^{(t)}(x_0)$ en tiempo lineal con un simple `for` que vaya calculando $t!$ para cada t y lo multiplique con b_t.

2. Explique como encontrar $b_0, b_1, ..., b_{n-1}$ en $O(n \log(n))$, dado $A(x_0 + \omega_n^k)$ para $k = 0, ..., n-1$.

Solución:

Tenemos los valores de $v_k = A(x_0 + \omega_n^k)$ para $k = 0, ..., n-1$. Tenemos que

$$v_k = A(x_0 + \omega_n^k) = \sum_{j=0}^{n-1} b_j (x_0 + \omega_n^k - x_0)^j = \sum_{j=0}^{n-1} b_j (\omega_n^k)^j$$

$$v_k = \sum_{j=0}^{n-1} b_j (\omega_n^k)^j$$

Así, dado los valores de v_k podemos calcular los valores de b_j en tiempo $O(n\log(n))$ utilizando la inversa de la transformada rápida de Fourier.

3. Demuestre que

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right)$$

donde $f(j) = a_j \cdot j!$ y

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & -(n-1) \le l \le 0 \\ 0 & 1 \le l \le n-1 \end{cases}$$

Solución:

Sea $\chi(a) = \begin{cases} 1 & x \ge 0 \\ 0 & x < 0 \end{cases}$.

Tenemos que

$$\begin{aligned}
A(x_0 + \omega_n^k) &= \sum_{j=0}^{n-1} a_j (x_0 + \omega_n^k)^j \\
&= \sum_{j=0}^{n-1} a_j \sum_{r=0}^{j} \binom{j}{r} (\omega_n^k)^r x_0^{j-r} \\
&= \sum_{j=0}^{n-1} a_j \sum_{r=0}^{n-1} \binom{j}{r} (\omega_n^k)^r x_0^{j-r} \cdot \chi(j-r) \\
&= \sum_{j=0}^{n-1} a_j \sum_{r=0}^{n-1} \frac{j!}{r!(j-r!)} (\omega_n^k)^r x_0^{j-r} \cdot \chi(j-r) \\
&= \sum_{j=0}^{n-1} \sum_{r=0}^{n-1} \frac{a_j j!}{r!(j-r!)} (\omega_n^k)^r x_0^{j-r} \cdot \chi(j-r) \\
&= \sum_{r=0}^{n-1} \sum_{j=0}^{n-1} \frac{a_j j!}{r!(j-r!)} (\omega_n^k)^r x_0^{j-r} \cdot \chi(j-r) \\
&= \sum_{r=0}^{n-1} \frac{w_n^{kr}}{r!} \sum_{j=0}^{n-1} a_j j! \frac{x_0^{j-r} \cdot \chi(j-r)}{(j-r!)} \\
&= \sum_{r=0}^{n-1} \frac{w_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j)
\end{aligned}$$

4. Explique como evaluar $A(x_0 + \omega_n^k)$ para $k = 0, 1, ..., n-1$ en $O(n\log(n))$. Concluya que es posible evaluar todas las derivadas no triviales de $A(x)$ en x_0 en tiempo $O(n\log(n))$.

Solución:

Sea $c_r = \sum_{j=0}^{n-1} f(j)g(r-j)$. Notemos que esto es una convolución, por lo que podemos utilizar FFT para calcular $c_0, c_1, ..., c_{n-1}$ en tiempo $O(n\log(n))$. Luego tenemos que

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \frac{c_r}{r!} w_n^{kr}$$

Tenemos los valores de $\frac{c_r}{r!}$ para $r = 0, ..., n-1$, por lo que nuevamente podemos utilizar FFT para calcular los valores de $A(x_0 + \omega_n^k)$ para $k = 0, ..., n-1$ en tiempo $O(n\log(n))$.

Una vez hecho esto podemos utilizar las partes 2 y 1 de esta pregunta para evaluar todas las derivadas no triviales de $A(x)$ en x_0. Dado que solo se realiza una cantidad constante de operaciones $O(n\log(n))$, el proceso completo también es $O(n\log(n))$.

Problema 2

Se puede generalizar la transformada discreta de Fourier a d dimensiones. El input es un arreglo d-dimensional $A = (a_{j_1,j_2,...,j_d})$ cuyas dimensiones son $n_1, n_2, ..., n_d$, donde $n_1 \cdot n_2 \cdot ... \cdot n_d = n$. Se define la transformada discreta de Fourier d-dimensional como

$$y_{k_1,k_2,...,k_d} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1} \omega_{n_2}^{j_2\cdot k_2} \cdots \omega_{n_d}^{j_d\cdot k_d}$$

para $0 \le k_1 < n_1$,$0 \le k_2 < n_2$,...,$0 \le k_d < n_d$.

1. Muestra que se puede calcular DFT d-dimensional computando solo DFTs en una dimensión. Esto es, primero se calculan n/n_1 DFTs 1-dimensionales sobre la dimensión 1. Luego, utilizando este resultado se computan n/n_2 DFTs sobre la dimensión 2. Luego se computan n/n_3 DFTs sobre la dimensión 3, y así sucesivamente.

   Solución:

   Podemos reordenar las sumatorias de tal forma que

$$\begin{aligned} y_{k_1,k_2,...,k_d} &= \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_d=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1} \omega_{n_2}^{j_2\cdot k_2} \cdots \omega_{n_d}^{j_d\cdot k_d} \\ &= \sum_{j_d=0}^{n_1-1} \sum_{j_{d-1}=0}^{n_2-1} \cdots \sum_{j_1=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1} \omega_{n_2}^{j_2\cdot k_2} \cdots \omega_{n_d}^{j_d\cdot k_d} \\ &= \sum_{j_d=0}^{n_1-1} \sum_{j_{d-1}=0}^{n_2-1} \cdots \left(\sum_{j_1=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1} \right) \omega_{n_2}^{j_2\cdot k_2} \cdots \omega_{n_d}^{j_d\cdot k_d} \end{aligned}$$

   Fijando los valores para $j_2, ..., j_d$, podemos calcular el valor de

$$b_{k_1,j_2,...,j_d} = \sum_{j_1=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1}$$

   para $k_1 = 1, ..., n_1 - 1$ computando la DFT en una dimensión. Tenemos que repetir esto para cada posible valor de $j_2, ..., j_d$, por lo que hay que realizar $n_2 \cdot n_3 \cdot ... \cdot n_d = \frac{n}{n_1}$ DFTs. Una vez hecho esto tenemos que

$$\begin{aligned} y_{k_1,k_2,...,k_d} &= \sum_{j_d=0}^{n_1-1} \sum_{j_{d-1}=0}^{n_2-1} \cdots \sum_{j_2=0}^{n_d-1} \left(\sum_{j_1=0}^{n_d-1} a_{j_1,j_2,...,j_d} \omega_{n_1}^{j_1\cdot k_1} \right) \omega_{n_2}^{j_2\cdot k_2} \omega_{n_3}^{j_3\cdot k_3} \cdots \omega_{n_d}^{j_d\cdot k_d} \\ &= \sum_{j_d=0}^{n_1-1} \sum_{j_{d-1}=0}^{n_2-1} \cdots \left(\sum_{j_2=0}^{n_d-1} b_{k_1,j_2,...,j_d} \omega_{n_2}^{j_2\cdot k_2} \right) \omega_{n_3}^{j_3\cdot k_3} \cdots \omega_{n_d}^{j_d\cdot k_d} \end{aligned}$$

Ahora podemos aplicar el mismo procedimiento anterior: si fijamos los valores de $k_1, j_3, ..., j_d$ podemos calcular

$$c_{k_1,k_2,j_3,...,j_d} = \sum_{j_2=0}^{n_d-1} b_{k_1,j_2,...,j_d} \omega_{n_2}^{j_2 \cdot k_2}$$

para $k_2 = 0, ..., n_2 - 1$ computando la DFT sobre la segunda dimensión. Esto se tiene que repetir para cada valor de $k_1, j_3, ..., j_n$, por lo que se tienen que realizar $n_1 \cdot n_3 \cdot ... \cdot n_d = \frac{n}{n_2}$ DFTs.

Este procediemiento se puede continuar, calculando $\frac{n}{n_3}$ DFTs en la dimensión 3, $\frac{n}{n_4}$ DFTs en la dimensión 4, etc. hasta finalmente calcular $y_{k_1,k_2,...,k_d}$.

2. Muestra que el orden de las dimensiones no importa, por lo que podemos calcular la DFT d-dimensional calculando las DFTs en una dimensión en cualquier orden.

   Solución:

   Así como reordenamos las sumatorias, también podemos reordenar los factores

   $$\omega_{n_1}^{j_1 \cdot k_1} \omega_{n_2}^{j_2 \cdot k_2} \cdots \omega_{n_d}^{j_d \cdot k_d}$$

   Así, dado un orden cualquiera de las sumatorias solo tenemos que ordenar los $\omega_{n_i}^{j_i \cdot k_i}$ en el orden inverso para poder aplicar el procedimiento anterior. Esto nos permite calcular las DFTs en cualquier orden.

3. Muestra que si computamos cada una de las DFTs 1-dimensionales utilizando la transformada rápida de Fourier, entonces el tiempo total para calcular la DFT d-dimensional es $O(n \log(n))$, independiente de d.

   Solución:

   Hay que realizar $\frac{n}{n_k}$ FFTs para la dimensión k, y cada una de ellas toma tiempo $O(n_k \log(n_k))$. Por lo tanto, el tiempo total es

   $$\begin{aligned} T &= \sum_{k=1}^{d} \frac{n}{n_k} n_k \log(n_k) \\ &= \sum_{k=1}^{d} n \log(n_k) \\ &= n \cdot \sum_{k=1}^{d} \log(n_k) \\ &= n \cdot \log(\prod_{k=1}^{d} n_k) \\ &= n \cdot \log(n) \end{aligned}$$

Problema 3

Consideremos el problema de representar n centavos por la menor cantidad de monedas. Asuma que el valor de cada moneda es un entero.

1. Entregue un algoritmo codicioso que represente n centavos con la menor cantidad de monedas de 1, 5, 10 y 25 centavos. Demuestra que el algoritmo es óptimo.

   Solución:

   El algoritmo es: Tomar la moneda más grande que sea menor o igual a n, y luego aplicar recursivamente el procedimiento sobre el número restante de centavos.

Para demostrar que este algoritmo codicioso es óptimo, mostremos primero que el problema tiene subestructura óptima:

Tomemos una solución óptima para el problema de representar n centavos, y digamos que esta solución utiliza k monedas. Supongamos que sabemos que esta solución utiliza una moneda de c centavos. Entonces la solución debe incluir la solución óptima para $n - c$ centavos. Claramente nuestra solución utiliza $k - 1$ monedas para representar los $n - c$ centavos. Si este no fuera el óptimo y se pudieran representar con $q < k - 1$ monedas, entonces podríamos tomar estas q monedas, agregar la moneda de c centavos y obtener una solución que utiliza $q + 1 < k$ monedas para n centavos, lo que contradice el hecho que nuestra solución óptima utilizaba k monedas.

Para demostrar que el algoritmo codicioso entrega una solución óptima falta demostrar que la elección codiciosa es correcta, es decir, existe una solución óptima que utiliza la moneda escogida por nuestro algoritmo. Si hacemos esto estamos listos, puesto que recursivamente el algoritmo encontrará la solución óptima al subproblema y lo combinará con la elección codiciosa. Dado que el problema tiene subestructura óptima, esto entrega el óptimo al problema completo.

Demostraremos que la elección codiciosa es correcta viendo los distintos casos sobre n:

- $1 \leq n < 5$: En este caso nuestro algoritmo elige una moneda de 1 centavo. Esto claramente es correcto: no hay más opción que utilizar n monedas de 1 centavo.
- $5 \leq n < 10$: Nuestro algoritmo elige una moneda de 5 centavos. Si una solución óptima no incluyera una moneda de 5 centavos, entonces necesariamente está utilizando más de 5 monedas de 1 centavo. Podemos tomar esas 5 monedas y cambiarlas por una moneda de 5 centavos, mejorando la solución. Esto es una contradicción, por lo que la solución óptima debe incluir una moneda de 5 centavos y nuestra elección es correcta.
- $10 \leq n < 25$: Nuestro algoritmo elige una moneda de 10 centavos. Si una solución óptima no incluyera esta moneda, entonces tenemos 3 casos: utiliza 2 o más monedas de 5 centavos, utiliza 1 moneda de 5 centavos y 5 o más monedas de 1 centavo, o utiliza solamente monedas de 1 centavo. En todos los casos hay un subconjunto de las monedas que suma 10 centavos. Al igual que antes, si reemplazamos estas monedas por una de 10 centavos mejora la solución, lo que es una contradicción.
- $n \leq 25$: En este caso nuestro algoritmo escoge una moneda de 25 centavos. Si una solución óptima no utiliza una moneda de 25 centavos, entonces tenemos 2 casos: utiliza 3 o más monedas de 10 centavos (en este caso las reemplazamos por una de 25 y otra de 5 centavos, mejorando la solución), o utiliza 2 o menos monedas de 10 centavos. En el segundo caso siempre hay un subconjunto de las monedas que suma 25 centavos (el analisis sería similar al del caso anterior), por lo que reemplazandolas se mejora la solución. Así, en cualquier caso se llega a una contradicción si la solución no incluye una moneda de 25 centavos.

Esto demuestra que la elección codiciosa siempre es correcta y nuestro algoritmo es óptimo para estos valores de las monedas.

2. Suponga que hay $k + 1$ monedas con valores $c^0, c^1, ..., c^k$ para algún entero $c > 1$. Muestra que el algoritmo codicioso siempre entrega una solución óptima.

   Solución:

   Al igual que antes, demostremos que la elección codiciosa de nuestro algoritmo siempre es correcta. Pongámonos en el caso genérico que $c^i \leq n < c^i + 1$. Nuestro algoritmo elige una moneda de valor c^i.

   Supongamos por contradicción que una solución óptima no incluye ninguna moneda de valor c^i. Sea a_k la cantidad de monedas de valor c^k en la solución óptima. Tenemos que para todo j, $a_j < c$ (si no fuera este el caso y $a_j \geq c$ para algún j, podríamos tomar $a'_j = a_j - c$, $a'_{j+1} = a_{j+1} + 1$ y mejorar la solución, lo que sería una contradicción pues la solución es óptima).

Tenemos que

$$\begin{aligned} n = \sum_{j=0}^{i-1} a_j c^j &\leq \sum_{j=0}^{i-1} (c-1)c^j \\ &= (c-1)\sum_{j=0}^{i-1} c^j \\ &= (c-1)\frac{c^i - 1}{c-1} \\ &= c^i - 1 \\ &< c^i \\ &\leq n \end{aligned}$$

Esto es una contradicción pues llegamos a que $n < n$. Por lo tanto, la solución óptima debe incluir una moneda de valor c^i, lo que implica que nuestra elección codiciosa es correcta.

El mismo razonamiento aplica cuando $n \leq c^k$.

3. Muestra un conjunto de monedas tal que el algoritmo codicioso no entrega un valor óptimo. El conjunto debe incluir la moneda de 1 centavo para que haya solución para cada n.

   Solución:

   Un conjunto es $\{1, 30, 31\}$. Si tomamos $n = 60$, el algoritmo entrega que hay que usar 1 moneda de 31 centavos y 29 de 1 centavo, pero el óptimo es usar 2 monedas de 30 centavos.

4. Entrega un algoritmo que funcione en tiempo $O(nk)$ que entregue la representación de n centavos con el mínimo número de monedas para cualquier conjunto de k monedas, asumiendo que una de las monedas vale 1 centavo.

   Solución:

   Sea f una función que indica el número de monedas en la solución óptima al problema de representar n centavos, donde las monedas disponibles valen $c_0, ..., c_{k-1}$. Ya demostramos que si la solución utiliza una moneda de c centavos, entonces el óptimo es combinar la solución al problema de $n - c$ centavos con la elección de la moneda de c centavos. El problema es que no sabemos qué monedas utiliza la solución.

   Lo que podemos hacer es tomar el mínimo sobre todas las posibles elecciones de monedas. Haciendo esto se obtiene la siguiente recurrencia:

$$f(n) = \begin{cases} 0 & n \leq 0 \\ 1 + \min\limits_i \{f(n - c_i)\} & n > 0 \end{cases}$$

   Por lo tanto, el óptimo para n centavos consiste en tomar la moneda que produce la solución óptima más pequeña para el subproblema. Utilizando esta recurrencia podemos construir el siguiente algoritmo de programación dinámica:

```
function f(n, V[0...k-1])                    ▷ V[i] es el valor de la i-ésima moneda
    m = [∞; n + 1]
    p = [−1; n + 1]
    m[0] = 0
    for i = 1 to n do
        for j = 0 to k − 1 do
            if i − j ≥ 0 and m[i − V[j]] + 1 < m[i] then
                m[i] = m[i − V[j]] + 1
                p[i] = j
    return m[n], p
```

El algoritmo retorna el mínimo número de monedas más la información necesaria (en el arreglo p) para reconstruir la elección de monedas en tiempo lineal. Para reconstruir la elección de monedas podemos aplicar el siguiente algoritmo:

```
function r(n, V, p)
    c = [0; k]                          ▷ c[i] es el número de monedas de valor V[i] utilizadas
    i = n
    while i > 0 do
        c[p[i]] = c[p[i]] + 1
        i = i − V[p[i]]
    return c
```

La función f toma tiempo $O(nk)$ puesto que simplemente hace un `for` sobre n y luego otro `for` anidado sobre k, donde la operación al interior del `for` anidado toma tiempo constante.

La función r toma tiempo lineal puesto que i disminuye desde n hasta 0 de manera monótona, donde cada iteración del `while` es $O(1)$.

Por lo tanto, el algoritmo completo es $O(nk)$.

Problema 4

Este problema compara la eficiencia de 3 métodos para calcular el enésimo número de Fibonacci F_n, dado n. Asuma que el costo de sumar, restar y multiplicar números es $O(1)$, independiente del tamaño de los números.

1. Muestra que el tiempo de ejecución del método recursivo ingenuo toma tiempo exponencial.

   Solución:

   El método recursivo ingenuo es el siguiente algoritmo:

   ```
   function F1(n)
       if n ≤ 1 then
           return n
       return F1(n − 1) + F1(n − 2)
   ```

   Si analizamos el árbol de recursión que se produce al evaluar $F_1(n)$ podemos ver que es un árbol binario donde la rama más pequeña tiene altura $\frac{n}{2}$. Por lo tanto, la cantidad de nodos es mayor a $2^{\frac{n}{2}}$, lo que es exponencial.

2. Muestra como calcular F_n en $O(n)$.

   Solución:

   El algoritmo es simplemente

   ```
   function F2(n)
       if n ≤ 1 then
           return n
       a = 0
       b = 1
       for i = 1 to n − 1 do
           c = a + b
           a = b
           b = c
       return b
   ```

3. Muestra como calcular F_n en $O(\log(n))$ usando solo adición y multiplicación. Hint: Considera la matriz $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ y sus potencias.

   Solución:

Sea $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Se demostrará por inducción que $A^i = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix}$.

- Caso base: $i = 0$

  Este caso es trivial, tenemos que $A^1 = A = \begin{pmatrix} F_0 & F_1 \\ F_1 & F_2 \end{pmatrix}$

- Hipotesis de Inducción: $A^i = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix}$

- Tesis de Inducción: $A^{i+1} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i+1} & F_{i+2} \end{pmatrix}$

  Tenemos que

$$A^{i+1} = A^i \cdot A = \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_i & F_{i-1} + F_i \\ F_{i+1} & F_i + F_{i+1} \end{pmatrix} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i+1} & F_{i+2} \end{pmatrix} \blacksquare$$

Utilizando esto podemos aplicar el siguiente algoritmo para obtener el resultado:

function $F_3(n)$
  $B = A^{n+1}$
  return $B[0, 1]$

Si se utiliza exponenciación rápida para calcular A^{n+1}, el algoritmo toma $O(\log(n))$.

4. Asuma que sumar dos números de β bits toma tiempo $\Theta(\beta)$ y que la multiplicación de dos números de β bits toma $\Theta(\beta^2)$. ¿Cual es el tiempo de ejecución de cada uno de los algoritmos anteriores bajo estos supuestos?

   Solución:

   Se puede demostrar que F_n tiene $\Theta(n)$ bits. Por lo tanto, tenemos que

   - El método ingenuo tiene un árbol de tamaño exponencial, donde cada operación es $O(n)$. Por lo tanto, podemos acotar el tiempo de ejecución por $O(n \cdot 2^n)$.
   - El tiempo de ejecución del segundo algoritmo es

$$T_2(n) = T_2(n-1) + \Theta(n)$$

     La solución a esta recurrencia es $T_2(n) \in \Theta(n^2)$.

   - El tiempo de ejecución del tercer algoritmo es

$$T_3(n) = 2T_3(n/2) + \Theta(n^2)$$

     La solución a esta recurrencia también es $\Theta(n^2)$.

5. Muestra cómo multiplicar dos números de β bits en tiempo $O(\beta \log(\beta))$.

   Solución:

   Dados dos números a y b de β bits, los podemos representar como

$$a = a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + ... + a_{\beta-1} 2^{\beta-1}$$

$$b = b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + ... + b_{\beta-1} 2^{\beta-1}$$

   Notemos que si multiplicamos a y b y agrupamos los términos según sus potencias de 2, obtenemos

$$\begin{aligned} c = a \cdot b &= (a_0 b_0) 2^0 + (a_0 b_1 + a_1 b_0) 2^1 + (a_0 b_2 + a_1 b_1 + a_2 b_0) 2^2 + ... \\ &= \sum_{i=0}^{\beta} \left(\sum_{j=0}^{i} a_j b_{i-j} \right) 2^i \end{aligned}$$

Sea $c_i = \sum_{j=0}^{i} a_j b_{i-j}$. Notemos que esto es una convolución, y que por lo tanto podemos calcular los c_i para $i = 0, ..., \beta - 1$ en tiempo $O(\beta \log(\beta))$[1] utilizando la transformada rápida de Fourier. Una vez obtenido el valor de todos los c_i, podemos hacer una pasada lineal[2] sobre ellos arrastrando el *carry* de las potencias menores a las mayores. Al terminar, la concatenación de los c_i entrega la multiplicación de a y b.

6. ¿Cual es el tiempo de ejecución del tercer algoritmo utilizando el método de multiplicación anterior?

   Solución:

   Asumiendo que el resultado anterior es cierto, la recurrencia para el tercer método queda

   $$T_3(n) = 2T_3(n/2) + \Theta(n \log(n))$$

   Podemos utilizar el teorema maestro para concluir que $T_3(n) \in \Theta(n \log(n))$.

[1] Hay un error en esta solución, pero solo me dí cuenta después de haber hecho la ayudantía: para demostrar que FFT es $O(n \log(n))$ se debe asumir que las sumas y multiplicaciones toman tiempo unitario, que es precisamente lo que no estamos asumiendo en esta pregunta.

[2] Nuevamente, arrastrar los carries toma $O(n)$ asumiendo que las sumas y restas son $O(1)$, lo que es un error.