Algoritmos y Complejidad

Algoritmos “dividir y conquistar”

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Primer Cuatrimestre 2017

Algoritmos “dividir y conquistar”

Generalidades

- “dividir y conquistar” (DYC) es un técnica de diseño de algoritmos que consiste en
  1. descomponer la instancia del problema a resolver en un conjunto de instancias más pequeñas del mismo problema
  2. resolver independientemente cada una de estas subinstancias. No se guardan resultados de instancias previamente calculadas, como en PD.
  3. combinar estas soluciones en una solución a la instancia original.
- es probable que esta técnica resulte en un algoritmo más eficiente que el original.

- es importante entonces determinar para cada problema:
  1. cuáles son las subinstancias, y cómo se encuentran
  2. cómo solucionar el problema en las subinstancias
  3. cómo combinar las soluciones parciales
- para el punto 2. se puede aplicar nuevamente la técnica DYC, hasta que se llegue a subinstancias de tamaño suficientemente pequeño para ser resueltas inmediatamente.

Esquema General

```
function DYC(x)
 IF x es suficientemente simple
    RETURN algoritmoBasico(x)
 ELSE
    descomponer x en x[1],x[2],...,x[s]
    FOR i ::= 1 TO s
       y[i] ::= DYC(x[i])
    ENDFOR
    combinar y[i] en una solución y a x
    RETURN y
 ENDINF
```


- se debe especificar cuáles son el algoritmo básico, el de descomposición y el de combinación
- también se necesita determinar cuándo una instancia es suficientemente simple como para dejar de aplicar la división
- si se trata de tamaño, este valor se denomina umbral

Análisis general del tiempo de ejecución

- el tiempo de ejecución está determinado por la recurrencia:

$$T_{DYC}(n) = \begin{cases} f(n) & \text{si } n \text{ es simple} \\ sT_{DYC}(n \div b) + g(n) & \text{sino} \end{cases}$$

donde

- $n = |x|$
- b es una constante tal que $n \div b$ aproxime el tamaño de las subinstancias
- $f(n)$ es el tiempo de `algoritmoBásico()`
- $g(n)$ es el tiempo de la partición y combinación.

- los métodos vistos para resolver recurrencias brindan soluciones a la mayoría de las recurrencias generadas por algoritmos DYC.
- en especial el método del teorema maestro.
- para que DYC sea eficiente las subinstancias deben ser todas de aproximadamente el mismo tamaño.
- además, se debe estudiar cuidadosamente cuál o cuáles son los mejores umbrales.

Determinación del umbral

- la determinación del umbral no afecta en general el orden del tiempo de ejecución de los algoritmos DYC
- pero sí afecta considerablemente las constantes ocultas
- ejemplo

$$T_{DyC} = \begin{cases} n^2 \ \mu\text{seg} & \text{si } n \leq n_0 \\ 3T_{DyC}(\lfloor n/2 \rfloor) + 16n \ \mu\text{seg} & \text{sino} \end{cases}$$

  suponiendo el algoritmo directo de $\Theta(n^2)$
- entonces resulta $T_{DYC}(n) \in \Theta(n^{\log 3})$

- cambiando el umbral se obtienen los siguientes tiempos absolutos:

n	T_{DyC} con $n_0 = 1$	T_{DyC} con $n_0 = 64$	Algoritmo básico
5000	41 seg	6 seg	25 seg
32000	15 min	2min	15 min

- es muy difícil, o a veces imposible, encontrar teóricamente un umbral optimal (ya sea para cada instancia o incluso para cada n).
- puede pasar que el umbral óptimo cambia de instancia en instancia, y que dependa de cada implementación en particular.
- también es poco práctico encontrar empíricamente una aproximación a un buen umbral: sería necesario ejecutar el algoritmo muchas veces en una gran cantidad de instancias
- la solución generalmente tomada es un camino híbrido:
  1. se encuentra la función exacta del tiempo de ejecución de los algoritmos (no alcanza con conocer sólo el orden!) dando valores a las constantes de acuerdo pruebas empíricas.
  2. se toma como n_0 un valor en el cual tome aproximadamente el mismo tiempo el algoritmo directo que el DyC

Correctitud

- a diferencia de los algoritmos *greedy*, es fácil probar la correctitud de los algoritmo DYC
- se supone la correctitud del algoritmo básico, y se prueba por inducción sobre el tamaño de la instancia que la solución obtenida es correcta suponiendo la correctitud de las instancias más chicas
- no vamos a ver en detalle ninguna prueba de correctitud para DYC, pero no son difíciles de hacer

MULTIPLICACION DE ENTEROS GRANDES

- Problema: supongamos que tenemos que multiplicar dos enteros **a** y **b**, de n y m dígitos cada uno, cantidades que no son posibles de representar directamente por el HW de la máquina
- es fácil implementar una estructura de datos para estos enteros grandes, que soporte
  1. suma de $\Theta(n+m)$.
  2. resta de de $\Theta(n+m)$.
  3. productos y divisiones por la base de $\Theta(n+m)$.

- si se implementa cualquiera de los algoritmos tradicionales para el producto entre dos números cualesquiera, el resultado es de $\Theta(nm)$
- aplicaremos DYC para tratar de mejorar este tiempo. Suponiendo por el momento que $n = m$

- aplicando DYC una sola vez, y usando base 10 se tiene:

a	x	y	**b**	w	z
	$\lceil n/2 \rceil$	$\lfloor n/2 \rfloor$		$\lceil n/2 \rceil$	$\lfloor n/2 \rfloor$

- esto es:

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= (x10^{\lfloor n/2 \rfloor} + y) \times (w10^{\lfloor n/2 \rfloor} + z) = \\ &= xw10^{2\lfloor n/2 \rfloor} + (xz + wy)10^{\lfloor n/2 \rfloor} + yz \end{aligned}$$

- si se extiende el método aplicando DYC recursivamente se obtiene la recurrencia:

$$t_{DyC}(n) = \begin{cases} \Theta(n^2) & \text{si } n \leq n_0 \\ 4t_{DyC}(\lceil n/2 \rceil) + \Theta(n) & \text{sino} \end{cases}$$

- el resultado no es bueno (aplicar el teorema maestro!)
- el principal problema es que se necesitan cuatro productos más pequeños.

- se puede reducir esta cantidad de productos, observando

$$\begin{aligned} \mathbf{r} &= (\mathbf{x}+\mathbf{y})(\mathbf{w}+\mathbf{z}) = \\ &= \mathbf{xw} + (\mathbf{xz}+\mathbf{yw}) + \mathbf{yz} \end{aligned}$$

- con lo que resulta

$$(\mathbf{xz}+\mathbf{yw}) = \mathbf{r} - \mathbf{xw} - \mathbf{yz}$$

- y también

$$\mathbf{a} \times \mathbf{b} = \mathbf{xw}10^{2\lfloor n/2 \rfloor} + (\mathbf{r} - \mathbf{xw} - \mathbf{yz})10^{\lfloor n/2 \rfloor} + \mathbf{yz}$$

- se tiene entonces dos productos de $\lfloor n/2 \rfloor$ dígitos, un producto de a lo sumo $\lfloor n/2 \rfloor + 1$ dígitos, más sumas, restas y productos de potencias de la base.

- si el tiempo del algoritmo directo es $an^2 + bn + c$ y el de la sobrecarga es $g(n)$, entonces aplicando DYC una sóla vez se obtiene

$$\begin{aligned} T(n) &= 3a(\lfloor n/2 \rfloor)^2 + 3b\lfloor n/2 \rfloor + 3c + g(n) \\ &\leq (3/4)an^2 + (3/2)bn + 3c + g(n) \end{aligned}$$

- comparado con $an^2 + bn + c$ es sólo una mejora del 25 % en la constante del término principal, pero igualmente es de $\Theta(n^2)$

- para obtener una mejora asintótica es preciso aplicar DYC recursivamente a los productos más pequeños
- el tiempo de este algoritmo genera la siguiente recurrencia:

$$T_{DyC}(n) = \begin{cases} \Theta(n^2) & \text{si } n \text{ es pequeño} \\ T_{DyC}(\lfloor n/2 \rfloor) + T_{DyC}(\lceil n/2 \rceil) + \\ \quad + T_{DyC}(\lfloor n/2 \rfloor + 1) + \Theta(n) & \text{sino} \end{cases}$$

- se puede deducir de esta recurrencia que $T_{DyC}(n) \in O(n^{\log 3} | n = 2^k)$, usando otra vez el teorema maestro.
- como $T_{DyC}(n)$ es eventualmente no decreciente y $n^{\log 3}$ es de crecimiento suave, entonces $T_{DyC}(n) \in O(n^{\log 3})$, aplicando la regla de las funciones de crecimiento suave.
- análogamente se puede mostrar que $T_{DyC}(n) \in \Omega(n^{\log 3})$ (ejercicio).

- restaría determinar cuál es el tamaño suficientemente pequeño para que convenga aplicar el algoritmo directo. ¿Cómo podría hacerse?
- usando el método híbrido, encontrando la intersección entre el tiempo DYC y el tiempo del algoritmo básico

- si $m \neq n$ pero ambos son de la misma magnitud, entonces es posible completar el número más chico con ceros hasta llegar al tamaño del más grande
- pero esta solución no siempre es buena.
- ¿cómo aprocechar mejor la misma técnica si $m << n$? (ejercicio Ayuda: partir los números en pedazos de a m)
- en este caso se puede obtener un resultado de $\Theta(nm^{log(3/2)})$, que es asintóticamente mejor que $\Theta(nm)$, o que $\Theta(n^{\log 3})$ si $m << n$

BÚSQUEDA BINARIA

- Problema: dado un arreglo de enteros $T[1..n]$, ordenado en forma creciente, y un entero x, se quiere encontrar el índice i tal que $T[i-1] < x \leq T[i]$
- por simplicidad se supone la convención de que $T[0] = -\infty$ y $T[n+1] = +\infty$.
- el tradicional algoritmo de búsqueda binaria puede verse como una degeneración de algoritmos DyC, en donde la cantidad de subinstancia es 1
- en estos casos la técnica DYC se denomina simplificación

- un algoritmo *naïve* para resolver BÚSQUEDA BINARIA es:

```
function BúsquedaSecuencial(T[1..n],x)
 FOR i ::= 1 TO n
    IF T[i]>=x
       RETURN i
    ENDIF
 ENDFOR
 RETURN n+1
```

- este algoritmo *naïve* tiene tiempo $\Theta(n)$ en el peor caso, y $\Theta(1)$ en el mejor caso
- si todas las instancias del arreglo tienen igual probabilidad de ser llamadas, entonces el tiempo promedio también es de $\Theta(n)$
- para aplicar DYC se determina en cuál mitad del arreglo debería estar x, comparándolo con el elemento del medio
- luego se busca recursivamente en esa mitad


```
function BúsqBinaria(T[i..j],x)
 IF i=j
    RETURN i
 ELSE
    k ::= (i+j) div 2
    IF x<=T[k]
       RETURN BúsqBinaria(T[i..k],x)
    ELSE
       RETURN BúsqBinaria(T[k+1..n],x)
    ENDIF
 ENDIF
```


Análisis del tiempo de ejecución

- sea $m = j - i + 1$. El tiempo de ejecución genera la recurrencia:

$$T(m) \leq \begin{cases} a & \text{si } m = 1 \\ b + t(\lceil m/2 \rceil) & \text{sino} \end{cases}$$

- resolviendo se obtiene $T(m) \in \Theta(\log m)$ en el peor caso
- el mismo resultado se obtiene aún en el mejor caso. ¿cómo se puede modificar el algoritmo para mejorar este punto?

Mergesort

- mergesort es el algoritmo “obvio” para el ordenamiento de un arreglo usando DYC.
- consiste en partir el arreglo en dos mitades, ordenar cada una de las mitades por separado y hacer una mezcla de estas mitades ya ordenadas.

```
Mergesort(A[1..n])
 IF n es pequeño
    RETURN Inserción(A)
 ELSE
    crear A1 y A2 subarreglos de A
    B1 ::= Mergesort(A1)
    B2 ::= Mergesort(A2)
    Mezcla(B1, B2, B)
    RETURN B
 ENDIF
```


- la partición consiste en la creación de dos mitades del arreglo original
- la combinación es la mezcla de las mitades ordenadas
- hay que tener cuidado con el manejo de los parámetros, para evitar duplicar los arreglos. En este caso se pasarán los índices

- informalmente, el tiempo de ejecución está determinado por la recurrencia:

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \text{ es pequeño} \\ 2T(n \div 2) + \Theta(n) & \text{sino} \end{cases}$$

- donde:
  - $\Theta(1)$ es el tiempo de `Inserción()`, que es ser acotado por una constante suficientemente grande porque vale cuando n es pequeño
  - $\Theta(n)$ es el tiempo de la partición y de la mezcla

- según el método del teorema maestro, el resultado es de $O(n \log n)$, en el mismo orden que *heapsort*
- para una demostración formal, habría que resolver la recurrencia $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$, que también da de $\Theta(n \log n)$.

- es fundamental para que el tiempo sea de $\Theta(n \log n)$ que las dos subinstancias en las que se parte el problema sean de tamaño semejante
- en el caso extremo de partir en subinstancias de tamaño desparejo, la recurrencia sería

$$T(n) = \begin{cases} \Theta(n^2) & \text{si } n \le n_0 \\ T(n-1) + T(1) + \Theta(n) = & \\ = T(n-1) + \Theta(n) & \text{sino} \end{cases}$$

que resulta $T(n) \in \Theta(n^2)$.

Ejercicios

- ¿qué pasa si se divide el problema original en subinstancias de tamaño k y $n-k$, con k constante?
- ¿qué pasa si se divide el problema original en tres subinstancias de tamaño semejante?
- ¿qué pasa si la partición o la combinación toman tiempo de $\Theta(n^2)$ en lugar de $\Theta(n)$?

Quicksort

- a diferencia de *mergesort*, que hace una descomposición trivial pero con una recombinación costosa, el algoritmo quicksort (Hoare) pone énfasis en la descomposición
- la partición del arreglo a ordenar se realiza eligiendo un elemento (el pivote), y luego partiendo en dos subarreglos con los elementos menores o iguales al pivote, y con lo elementos mayores que el pivote.
- estos nuevos arreglos son ordenados en forma recursiva, y directamente concatenados para obtener la solución al problema original.
- es posible obtener una implementación del pivoteo en tiempo de $\Theta(n)$, incluso realizando una sola recorrida al arreglo

	costo
`Quicksort(A[i..j])`	
`IF j-i es pequeño`	c
`Inserción(A[i..j])`	$\Theta(1)$
`ELSE`	
`piv ::= A[i]`	c
`Pivotear(A[i..j],piv,l)`	$\Theta(n)$
`Quicksort(A[i..l-1])`	
`Quicksort(A[l+1..j])`	
`ENDIF`	$T(n-1)$ peor caso

- el tiempo de ejecución es

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \text{ es pequeño} \\ T(n-1) + \Theta(n) & \text{sino} \end{cases}$$

- usando la ecuación característica, $T(n) \in \Theta(n^2)$.

Algoritmo de pivoteo

Primera parte: encontrar los dos primeros elementos para intercambiar

```
Pivotear(A[i..j],piv,var l)
 k ::= i; l ::= j+1
 REPEAT
    k ::= k+1
 UNTIL A[k]>piv or k>j
 REPEAT
    l ::= l-1
 UNTIL A[l]<=piv
 ...
```


Segunda parte: si existen esos elementos, intercambiarlos y encontrar los siguientes elementos para intercambiar

```
...
WHILE k<l
   intercambiar A[k] y A[l]
   REPEAT
      k ::= k+1
   UNTIL A[k]>piv
   REPEAT
      l ::= l-1
   UNTIL A[l]<=piv
ENDWHILE
```


- el total de iteraciones es $l - i + 1$ para k y $j + 1 - l$ para j
- luego en total de iteraciones es de $\Theta(j - i)$
- como el tiempo del cuerpo de los ciclos es constante, el tiempo total es entonces de $\Theta(j - i)$

Análisis probabilístico

- se puede probar que el tiempo promedio del quicksort es de $\Theta(n \log n)$, asignando igual probabilidad a todos los arreglos
- supongamos que todas las $n!$ instancias tienen igual probabilidad de ser llamadas, y que todos los elementos de T son distintos
- esto implica que el pivote cae con igual probabilidad en cada una de las posiciones del arreglo a ordenar; y también que cada uno de las subinstancias generadas heredan una distribución uniforme.

- el tiempo promedio está definido entonces por la recurrencia:

$$\begin{aligned}
T_{prom}(n) &= \frac{1}{n}\left(\sum_{k=0}^{n-1} \Theta(n) + T_{prom}(k) + T_{prom}(n-1-k)\right) = \\
&= \Theta(n) + \frac{2}{n}\sum_{k=0}^{n-1} T_{prom}(k) = \\
&= \Theta(n) + \frac{2}{n}[T_{prom}(0) + T_{prom}(1) + \sum_{k=2}^{n-1} T_{prom}(k)] = \\
&= \Theta(n) + \frac{2}{n}\sum_{k=2}^{n-1} T_{prom}(k)
\end{aligned}$$

Teorema 1
Quicksort *tiene tiempo promedio en* $O(n \log n)$.

Prueba.
Por inducción constructiva se encuentran los valores para c tal que $T_{prom}(n) \leq cn \log n$. □

- con el objetivo de mejorar el tiempo de ejecución en el peor caso de quicksort para llevarlo a $\Theta(n \log n)$ se podría implementar una mejor elección del pivote
- se podría elegir cómo pivote el elemento mediano (aquel que estaría en la posición del medio una vez ordenado el arreglo), que parte al arreglo en dos mitades semejantes.
- pero esto no siempre es así si el arreglo tiene elementos repetidos.
- luego se necesita además partir el arreglo en tres partes (menores, iguales y mayores), no sólo en dos

- en resumen, para que *quicksort* sea de tiempo $\Theta(n \log n)$ en el peor caso se requiere:
  - una mejor elección del pivote, que asegure subarreglos de tamaño semejante, pero siempre en tiempo $\Theta(n)$.
  - modificar el pivotear para partir el arreglo en tres subarreglos: los elementos menores, los elementos iguales y los elementos mayores que el pivote.
- estas "mejoras" involucran constantes ocultas que hacen del algoritmo resultante practicamente inviable comparado con la versión *naïve*.
- se verá a continuación primero el nuevo algoritmo de pivoteo, y luego cómo mejorar la elección del pivote.

Algoritmo de pivoteo de la bandera holandesa

```
function PivoteoBH(p,var T[i..j],k,l)
 k ::= i-1; m ::= i; l ::= j+1
 WHILE m<l
    CASE
     T[m]=p: m ::= m+1
     T[m]>p: l ::= l-1; swap(T[m],T[l])
     T[m]<p: k ::= k+1; swap(T[m],T[k])
             m ::= m+1
    ENDCASE
 ENDWHILE
```

- el tiempo es de $\Theta(n)$; es más, sólo recorre al arreglo una vez (ejercicio).

- también se puede elegir el pivote al azar, resultando en una versión probabilística de Quicksort con tiempo esperado de $\Theta(n \log n)$
- la ventaja de esta versión es que no existe una instancia en la que tarde más, sino que ciertas ejecuciones sobre cualquier instancia son las que llevan el peor caso

SELECCCIÓN ELEMENTO MEDIANO

- Problema: se tiene un arreglo $A[1..n]$ de enteros, no necesariamente ordenado, y se quiere encontrar el elemento mediano.
- éste es el elemento que estaría en la posición $\lceil n/2 \rceil$ si el arreglo estuviera ordenado.
- si el elemento mediano es elegido como pivote cuando no hay elementos repetidos, se asegura la partición del arreglo en subarreglos de tamaño semejante

- la solución obvia a este problema es ordenar el arreglo y devolver $A(\lceil n/2 \rceil)$
- pero su tiempo de ejecución es de $\Theta(n \log n)$, y eso no mejoraría *quicksort* (ejercicio)
- para mejorar asintóticamente este tiempo, se solucionará un problema más general, denominado SELECCIÓN.

SELECCIÓN

- Problema: se tiene un arreglo $A[1..n]$ de enteros, no necesariamente ordenado, y un entero $s, 1 \leq s \leq n$. Se quiere encontrar el elemento s-ésimo.
- el s-esimo elemento es el elemento que estaría en la posición s de estar el arreglo ordenado
- SELECCIÓN y MEDIANO son equivalentes (existen reducciones en ambas direcciones).

Reducciones

- la reducción MEDIANO $\longrightarrow$ SELECCIÓN es trivial, basta con llamar a SELECCIÓN con $s = \lceil n/2 \rceil$.
- la reducción SELECCIÓN $\longrightarrow$ MEDIANO tampoco es difícil, ya que es muy parecida al algoritmo de búsqueda binaria

Reducción SELECCIÓN $\longrightarrow$ MEDIANO

```
function Selección(A[i..j],s)
 p ::= Mediano(A[i..j])
 PivotearBH(p,A[i..j],k,l)
 CASE
     s<=k: j ::= k
           r ::= Selección(T[i..j],s)
     k<s<l: r ::= p
     s>=l: i ::= l
           r ::= Selección(T[i..j],s-l+1)
 ENDCASE
 RETURN r
```

- la cantidad de llamadas recursivas es de $\Theta(\log n)$, similar a la búsqueda binaria
- se puede transformar en un algoritmo iterativo sin problemas.

- a partir de este algoritmo, cambiando la elección del pivote, se puede asegurar un peor tiempo lineal para SELECCIÓN
- el pivote no necesariamente es exactamente el mediano.
- esta solución también servirá para nuestro problema original: MEDIANO

- se necesita entonces un algoritmo eficiente para seleccionar un elemento que divida al arreglo ordenado en partes "suficientemente" parejas.
- Ejercicio: Sea $T_S(n)$ el tiempo del algoritmo de SELECCIÓN. Supongamos que seleccionar el pivote lleva tiempo $T_S(n/b)$, para algún b natural, $b \leq 9$. ¿cuán "suficientemente" buena, en función de b, tendría que ser la división del arreglo? (Ayuda: usar las propiedades de las recurrencias.)
- el siguiente algoritmo encuentra una aproximación al mediano con tiempo de ejecución en $\Theta(T_S(\lfloor n/5 \rfloor) + \lfloor n/5 \rfloor)$.

Código	Costo	Veces
`function PseudoMediano(A[1..n])`		
`IF n<=5`		
`p ::= MedAdhoc5(A[1..n])`	a	1
`ELSE`		
`z ::= piso(n/5)`	b	1
`array Medianos[1..z]`		
`FOR i ::= 1 TO z`		
`Medianos[i] ::=`		
`MedAdhoc5(A[5i-4..5i])`	c	$\lfloor n/5 \rfloor$
`ENDFOR`		
`p ::= Selección(Medianos[1..z],`		
`techo(z/2))`	$T_S(\lfloor n/5 \rfloor)$	1
`ENDIF; RETURN p`		

- donde `MedAdhoc5()` es un algoritmo para encontrar el mediano de a los sumo 5 elementos
- y por lo tanto es de tiempo $\Theta(1)$.
- el tiempo de ejecución de `Pseudomediano()` es $\Theta(n) + T_S(\lfloor n/5 \rfloor)$, donde $T_S()$ es el tiempo de ejecución del algoritmo para resolver el problema SELECCIÓN

	Costo
`function Selección(A[i..j],s)`	
`p ::= PseudoMediano(A[i..j])`	$\Theta(n) + t_S(\lfloor n/5 \rfloor)$
`PivotearBH(p,A[i..j],k,l)`	$\Theta(n)$
`CASE`	
`s<=k: r::=Selección(A[i..k],s)`	$T_S(k-i+1)$
`k<s<l: r::=p`	$\Theta(1)$
`s>=l: s::=s-l+1`	$\Theta(1)$
`r::=Selección(A[l..j],s)`	$T_S(j-l+1)$
`ENDCASE`	
`RETURN r`	$\Theta(1)$

Análisis del tiempo de ejecución

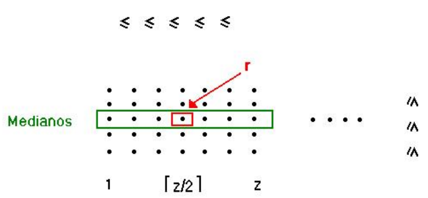
- el algoritmo anterior genera la recurrencia:

$$T_S(n) = \begin{cases} a & \text{si los elementos son iguales, o } n \leq 5 \\ \Theta(n) + T_S(\lfloor n/5 \rfloor) + \\ + T_S(\max_{k,l}(k-i+1, j-l+1)) & \text{sino} \end{cases}$$

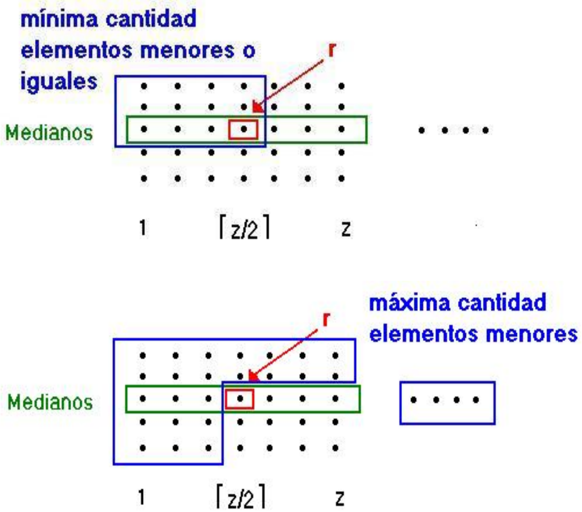
- ¿porqué? para saberlo es necesario conocer cómo se aproxima el pseudomediano al mediano verdadero (ie la relación entre $k-i+1$ y $j-l+1$ con $n/2$.

Lema 2
La posición r del resultado del algoritmo `Pseudomediano(A)` cuando `A` tiene n elementos es tal que $\frac{3}{10}n - \frac{6}{5} \leq r \leq \frac{7}{10}n + \frac{6}{5}$.

Prueba.



$\square$



Algoritmos y Complejidad
Elemento mediano
Selección

Lema 3

La posición r del resultado del algoritmo `Pseudomediano()` *cuando* `A` *tiene n elementos es tal que* $\frac{3}{10}n-4 \leq r \leq \frac{7}{10}n+\frac{6}{5}$.

Prueba.

La mínima cantidad de elementos menores o iguales es $3\lceil\frac{z}{2}\rceil \geq \frac{3}{10}n-\frac{6}{5}$. Luego la máxima cantidad de elementos mayores es menor que $n-(\frac{3}{10}n-\frac{6}{5}) = \frac{7}{10}n+\frac{6}{5}$. Por otro lado, la máxima candidad de elementos menores es a lo sumo $2z+3\lfloor\frac{z}{2}\rfloor+4 \leq \frac{7}{10}n+4$. □

Algoritmos y Complejidad
Elemento mediano
Selección

- la recurrencia para SELECCIÓN queda entonces

$$T_S(n) \leq \begin{cases} a & \text{si los elementos son iguales, o } n \leq 5 \\ \Theta(n) + T_S(\lfloor n/5 \rfloor) + \max_{m \leq 7n/10+4}(T_S(m)) & \text{sino} \end{cases}$$

- no se puede analizar ni con el teorema maestro ni con la ecuación característica.

Algoritmos y Complejidad
Elemento mediano
Selección

Teorema 4

$T_S(n) \in \Theta(n)$

Prueba.

Es fácil ver que $T_S(n) \in \Omega(n)$. Para el orden, por inducción constructiva sobre n, se encuentran los valores para c y n_0 tales que $T_S(n) \leq cn$ para todo $n \geq n_0$. □

Quicksort revisado

- usando entonces este algoritmo para calcular el mediano, y el pivoteo de la bandera holandesa se puede obtener la siguiente versión de *quicksort*

	costo
`Quicksort(A[i..j])`	
`IF j-i es pequeño`	$\Theta(1)$
`Inserción(A[i..j])`	$\Theta(1)$
`ELSE`	
`piv ::= Mediano(A[i..j])`	$\Theta(n)$
`PivotearBH(A[i..j],piv,k,l)`	$\Theta(n)$
`Quicksort(A[i..k])`	$T(n/2)$ peor caso
`Quicksort(A[l..j])`	$T(n/2)$ peor caso
`ENDIF`	

- el tiempo de ejecución es

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \text{ es pequeño} \\ 2T(n/2) + \Theta(n) & \text{sino} \end{cases}$$

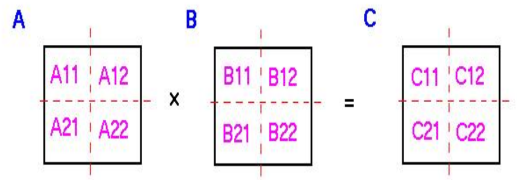
- usando el teorema maestro, $T(n) \in \Theta(n \log n)$.
- ¿qué pasa si esta versión de *quicksort* en lugar de llamar a `Mediano()`, busca el pivote con `PseudoMediano()`? (ejercicio)

MULTIPLICACIÓN DE MATRICES

- se tiene el problema de calcular el producto C de dos matrices A, B cuadradas de dimensión n
- el algoritmo directo tiene tiempo de ejecución en $\Theta(n^3)$, ya que cada uno de los n^2 elementos de C lleva tiempo $\Theta(n)$ en computarse
- si existiera un algoritmo dividir y conquistar que partiera las matrices originales en matrices de $n/2 \times n/2$, 8 o más multiplicaciones de estas matrices igualaría o empeoraría el tiempo del algoritmo directo (ejercicio).

Algoritmo de Strassen

- Strassen, a fines de los '60s, descubrió que 7 productos son suficientes
- la idea del algoritmo de Strassen es dividir las matrices en cuatro partes iguales, y resolver el producto original en base a operaciones sobre estas partes
- usando sumas y restas entre los componentes de $\Theta(n^2)$, en forma análoga al problema de multiplicar enteros grandes



- cada una de $A_{11}, \ldots, C_{22}$ tiene dimensión $\frac{n}{2} \times \frac{n}{2}$. Sumas y restas de estas matrices se puede computar en tiempo $\Theta(n^2)$

- si se definen las siguientes matrices auxiliares, también de $\frac{n}{2} \times \frac{n}{2}$

$$
\begin{aligned}
M_1 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12} + B_{11}) \\
M_2 &= A_{11} \times B_{11} \\
M_3 &= A_{12} \times B_{21} \\
M_4 &= (A_{11} - A_{21}) \times (B_{22} - B_{12}) \\
M_5 &= (A_{21} + A_{22}) \times (B_{12} - B_{11}) \\
M_6 &= (A_{12} - A_{21} + A_{11} - A_{22}) \times B_{22} \\
M_7 &= A_{22} \times (B_{11} + B_{22} - B_{12} - B_{21})
\end{aligned}
$$

- resulta que

$$
C = \begin{pmatrix} C_{11} = M_2 + M_3 & C_{12} = M_1 + M_2 + M_5 + M_6 \\ C_{21} = M_1 + M_2 + M_4 - M_7 & C_{22} = M_1 + M_2 + M_4 + M_5 \end{pmatrix}
$$

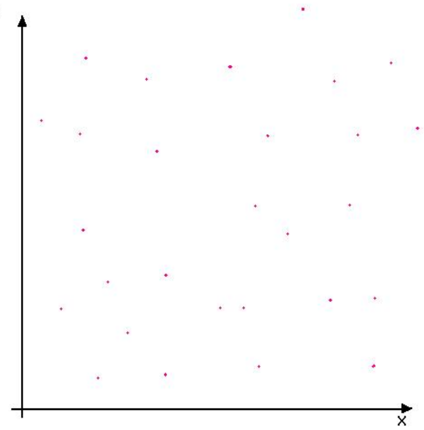
- el algoritmo de Strassen tiene tiempo de ejecución

$$
T(n) = \begin{cases} \Theta(n^3) & \text{si } n \text{ es pequeño} \\ 7T(n/2) + \Theta(n^2) & \text{sino} \end{cases}
$$

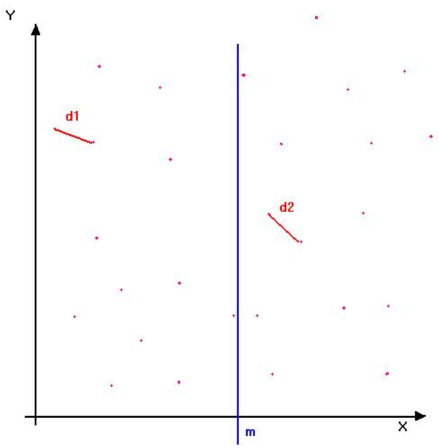
- resolviendo esta recurrencia con el teorema maestro resulta $T(n) \in \Theta(n^{\log 7})$ (ejercicio) si el tamaño n de la matriz es potencia de 2
- dado que $\log_2 7 = 2{,}81 < 3$ este algoritmo DYC es asintóticamente mejor que el algoritmo directo

- para las matrices cuyos n no son potencia de 2, es necesario completarlas con 0's hasta llegar a una potencia de 2
- otros algoritmos similares se han definido siguiendo esta estrategia; se divide a la matriz en $b \times b$ partes, y se busca una forma de calcular el producto con k componentes generados a partir de operaciones escalares en las partes tal que $\log_b k < \log_2 7$.

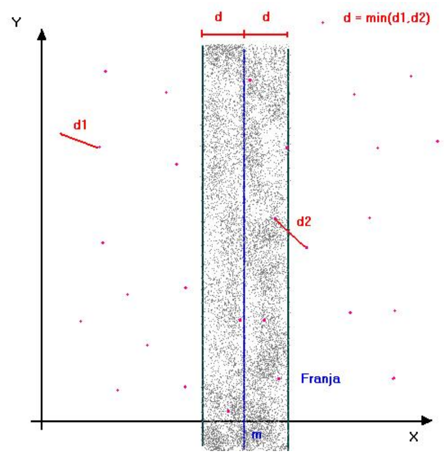
Definición del problema



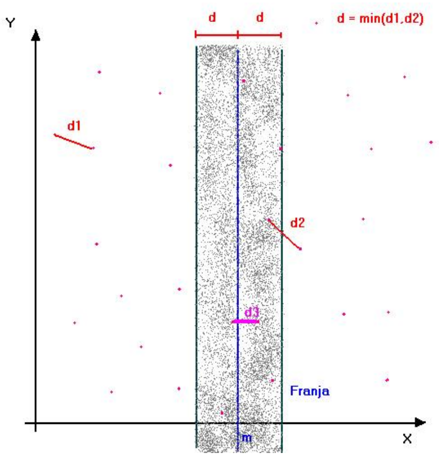
- sea un conjunto de n puntos en el plano (que supondremos en el primer cuadrante). Problema: se quiere encontrar el par de puntos más cercanos.
- el algoritmo directo debe comparar $\binom{n}{2}$ pares de puntos, por lo que su tiempo está en $\Theta(n^2)$.



- para aplicar DYC se divide el conjunto de puntos en partes iguales, por medio de una recta m, según el eje x.
- se encuentran entonces d_1 y d_2 las distancias mínimas en cada parte
- pero la solución al problema original no se encuentra tan fácil a partir de estos dos valores



- el par de puntos buscado puede estar repartido entre las dos mitades, por lo que ni d_1 ni d_2 necesariamente son solución
- entonces se crea *Franja* de ancho $2d$ alrededor de la recta m, siendo $d = \min(d_1, d_2)$, y se busca si existe una distancia menor que d en esa zona



- si existe en *Franja* una distancia $d_3 < d$ entonces ésa es la solución. En caso contrario la solución es d
- para que el algoritmo DYC sea más eficiente que el algoritmo directo, debe tenerse cuidado que la sobrecarga de la partición y combinación sean de tiempo menor que $\Theta(n^2)$

	costo
`function MasCercanos(P[1..n])`	
`IF  n es pequeño`	$\Theta(1)$
`RETURN algoritmoBasico(P)`	$\Theta(1)$
`ELSE`	
`m ::= punto medio de coord. x`	$\Theta(1)$
`crear P1 y P2`	??
`d1 ::= MásCercanos(P1)`	$T(\lfloor n/2 \rfloor)$
`d2 ::= MásCercanos(P2)`	$T(\lceil n/2 \rceil)$
`d ::= min(d1, d2)`	$\Theta(1)$
`crear Franja con ancho 2d de m`	??
`d3 ::= recorrido(Franja)`	??
`RETURN min(d,d3)`	
`ENDIF`	

Implementación

- para la creación eficiente de las subinstancias es posible ordenar el arreglo P por coordenadas x una sola vez al comienzo de la ejecución, agregando tiempo en $\Theta(n \log n)$ por única vez
- para la creación de `Franja` y la búsqueda eficiente en ese arreglo se puede:
  - ordenar el arreglo P también por coordenada y
  - partirlo de acuerdo a si cada punto pertenece o no a `Franja`
  - por cada punto considerar la distancia con solo los 7 siguientes dentro de `Franja` (esto es suficiente no puede haber más de 8 puntos en un rectángulo de $2d \times d$ cuya distancia sea menor o igual a d)

Análisis del tiempo de ejecución

- luego se genera la recurrencia:

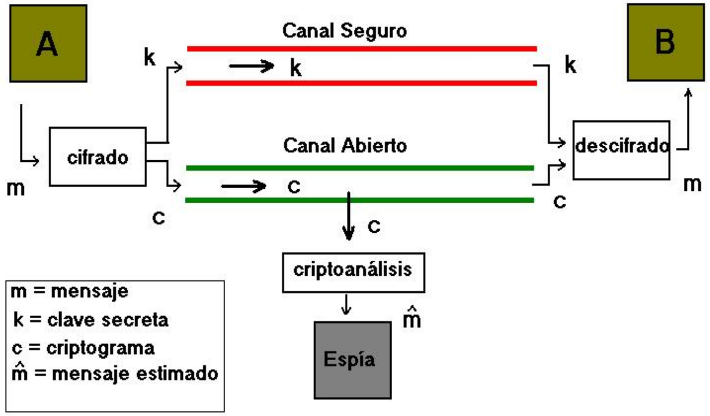
$$T(n) = \begin{cases} \Theta(1) & \text{para } n \text{ pequeño} \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n \log n) & \text{sino} \end{cases}$$

- resolviendo cambio de variables y la regla de funciones de crecimiento suave, resulta $T(n) \in \Theta(n \log^2 n)$.

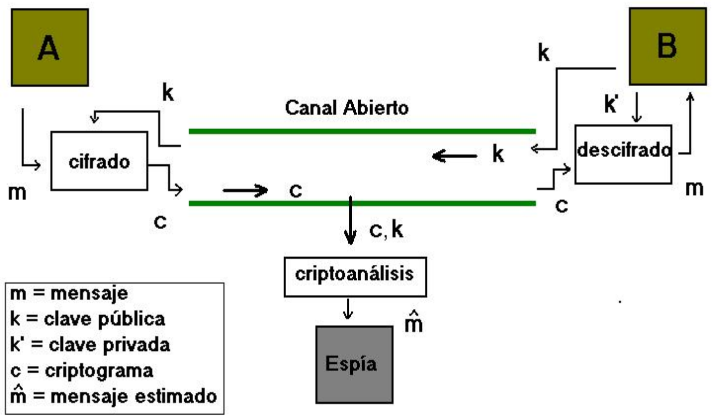
- se puede mejorar el tiempo de $\Theta(n \log^2 n)$ a costa de incorporar como entrada al algoritmo no solo P ordenado por coordenadas x, sino también ordenado por coordenas y.
- de esta forma se elimina el ordenamiento en cada llamada recursiva, realizándose solamente un única vez al comienzo del algoritmo, y la creación de `Franja` se puede hacer entonces en tiempo lineal
- el problema ahora es crear los nuevos arreglos ordenados por y para cada llamada recursiva; pero esto puede hacerse en tiempo lineal a partir del arreglo completo ordenado por y
- la recurrencia queda entonces $T(n) = 2T(n/2) + \Theta(n)$ que sabemos que está en $\Theta(n \log n)$

Criptografía

Esquema para protocolos de Clave Secreta



Esquema de un protocolo de Clave Pública



- los protocolos de Clave Pública solo fueron posibles a partir del estudio sistemático de la Algoritmia, a mediados de los '70s
- a continuación se presentará una solución simple propuesta por Rivest, Shamir y Adleman conocida como el sistema criptográfico RSA (1978)

Protocolo RSA

- B (el receptor del mensaje) elige dos números primos p y q (cuanto más grandes, más difícil de quebrar el cifrado) y calcula $z = pq$
- existen algoritmos eficientes para testear si un número es primo (se verá más adelante un algoritmo probabilístico) y para multiplicar enteros grandes (ya visto)
- sin embargo, no se conocen algoritmos eficientes para factorear z.

- B también debe elegir un número n aleatorio, tal que $1 < n < z - 1$, que no tenga factores comunes con $(p-1)(q-1)$
- existe un algoritmo eficiente (basado en el algoritmo de Euclides) que dado cualquier n, no sólo comprueba si cumple con la propiedad sino que al mismo tiemo calcula el único s tal que $1 \leq s \leq z - 1$ y $ns \bmod (p-1)(q-1) = 1$

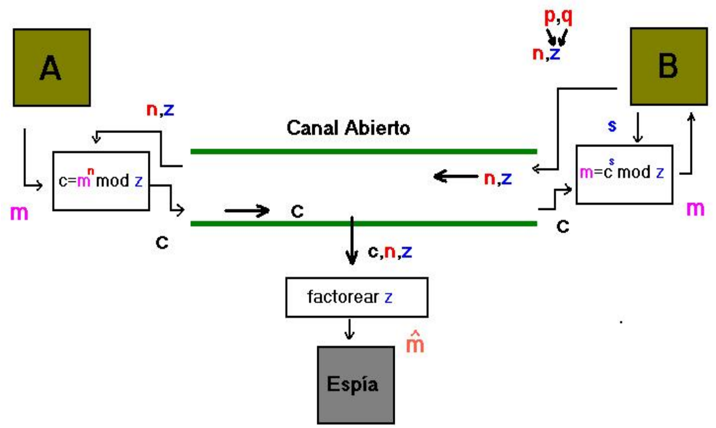
- lo interesante de estos números es que se puede probar que si $1 \leq a < z$ entonces $a^x \bmod z = a$, para todo x tal que $x \bmod (p-1)(q-1) = 1$
- la clave pública está formada por z y n, y la clave secreta (sólo conocida por B) por s
- el remitente A del mensaje m, $1 \leq m \leq z - 1$ (si no cumple con esta restricción, se parte el mensaje en pedazos de ese tamaño) debe calcular $c = m^n \bmod z$

- cuando B recibe c, a partir de la clave secreta s se calcula:

$$c^s \bmod z = (m^n \bmod z)^s \bmod z = (m^n)^s \bmod z = m^{ns} \bmod z = m$$

  según la propiedad anterior
- es necesario una implementación eficiente de la exponenciación modular $x^y \bmod z$
- el espía, con conocimiento de c, n y z, sólo puede factorear z en pq para hallar s y calcular m. Se supone que el factoreo de números grandes no se puede realizar en tiempo razonable

Esquema de un protocolo de Clave Pública



Exponenciación modular

- <u>Problema:</u> dados enteros grandes m, n, z se quiere calcular $m^n \bmod z$.
- la solución se obtiene a partir de un algoritmo DYC para calcular exponentes de enteros grandes
- usando además las siguientes propiedades:

$$x^y \bmod z = (x \bmod z)^y \bmod z$$

$$xy \bmod z = [(x \bmod z)(y \bmod z)] \bmod z$$

- el algoritmo DYC resulta
  - si y es par, $y = 2y'$ luego

$$x^y \bmod z \quad = \quad x^{2y'} \bmod z = (x^2 \bmod z)^{y'} \bmod z$$

  - si $y > 1$ es impar, $y = 2y' + 1$ luego

$$\begin{aligned} x^y \bmod z &= x^{2y'+1} \bmod z = x^{2y'} x \bmod z = \\ &= [(x^{2y'} \bmod z)(x \bmod z)] \bmod z \end{aligned}$$

- como la instancia se resuelve en base a la solución de un sólo subproblema, se trata de un caso de simplificación


```
function Expomod(x,y,z)
 a ::= x mod z
 IF y=1
    RETURN a
 ELSEIF y es par
    aux ::= a^2 mod z
    RETURN Expomod(aux,y/2,z)
 ELSEIF y es impar
    RETURN (a * Expomod(a,y-1,z)) mod z
 ENDIF
```

Análisis del tiempo de ejecución

- La recurrencia de su tiempo de ejecución es:

$$T_{EXP}(y) = \begin{cases} \Theta(|x| + |z|) & \text{si } y = 1 \\ T_{EXP}(y/2) + \Theta(|z|^2) & \text{si } y \text{ es par} \\ T_{EXP}(y-1) + \Theta(|z|^2) & \text{si } y \text{ es impar} \end{cases}$$

- ni siquiera es eventualmente no decreciente

- para solucionar la recurrencia se expande una vez el caso impar:

$$\begin{aligned} T_{EXP}(y) &= T_{EXP}(y-1) + \Theta(|z|^2) = \\ &= T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2) + \Theta(|z|^2) = \\ &= T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2) \end{aligned}$$

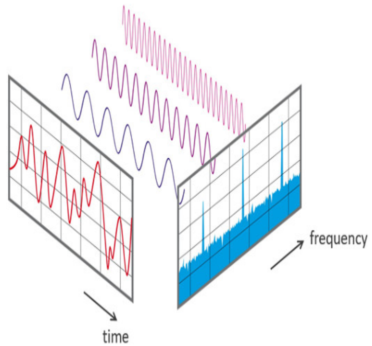
- con lo que:

$$T_{EXP}(y) \leq \begin{cases} \Theta(|x| + |z|) & \text{si } y = 1 \\ T_{EXP}(\lfloor y/2 \rfloor) + \Theta(|z|^2) & \text{si } y > 1 \end{cases}$$

- esta nueva recurrencia, que sí es eventualmente no decreciente, tiene como resultado $T_{EXP}(y) \in O(\log y)$, considerando constante el costo de las multiplicaciones

- análogamente se muestra que $T_{EXP}(y) \in \Omega(\log y)$
- tener en cuenta para ambos casos que y es el valor de uno de los datos de entrada, y no su longitud
- este tiempo asintótico se puede mejorar usando el algoritmo DYC para multiplicación de enteros grandes
- esta algoritmo es el algoritmo de encriptación y decriptación del protocolo RSA.

Transformada de Fourier



- la Transformada de Fourier toma una función representando un patrón basado en tiempo y la descompone en función de varios ciclos

Transformada de Fourier

- tiene muchas aplicaciones:
  - técnicas de codificación de video y audio digital(MP3, JPEG, MPEG)
  - ecualización de audio
  - filtros de imágenes (Gaussian blur)
  - procesamiento de señales de sonar para clasificar objetivos
  - vibraciones de terremotos
  - etc
- la señal original se representa como un polinomio de señales más sencillas
- es necesario representar polinomios y computar sus operaciones

Polinomios: representación por coeficientes

- sea $A(x)$ un polinomio, se puede representar por coeficientes como $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ de grado $n-1$, siendo los coeficientes $a_i \in \mathbb{F}$ y $\mathbb{F}$ un campo cualquiera (por ejemplo $\mathbb{Q}, \mathbb{R}, \mathbb{C}, \ldots$)
- alternativamente $A(x) = \sum_{k=0}^{n} a_k x^k$, o $A(x) = \langle a_0, a_1, \ldots, a_k \rangle$
- se dice que $A(x)$ está grado-acotado por n si su grado es $n-1, n-2, \ldots, 0$

Polinomios: representación por raíces

- $A(x)$ se puede representar también por sus raíces $r_i, 1 \leq i \leq n-1$, entonces $A(x) = c(x-r_1)\ldots(x-r_{n-1})$
- se garantiza que está representación es única

Teorema 5 (Teorema fundamental del algebra)

Todo polinomio de grado $n-1$ con $a_i \in \mathbb{C}$ tiene exactamente $n-1$ raíces complejas, contando multiplicidades.

Polinomios: representación por muestras

- también se puede representar un polinomio $A(x)$ grado-acotado por n tomando n muestras

$$A(x) = \{((x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$$

siempre que los x_k sean todos distintos y que $A(x_k) = y_k$

Teorema 6 (Unicidad de la representación por muestras)

Para cualquier conjunto $\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ de n muestras tal que todos los x_k son distintos, existe un único polinomio $A(x)$ grado-acotado por n tal que $A(x_k) = y_k$ para todo $i, 1 \leq i \leq n$.

Operaciones con polinomios

- evaluación dados $A(x)$ y x_0, encontrar $y_0 = A(x_0)$
- suma dados $A(x)$ y $B(x)$, encontrar $C(x) = A(x) + B(x)$
- multiplicación dados $A(x)$ y $B(x)$, encontrar $C(x) = A(x) \times B(x)$
- convolución dados $A(x)$ y $B(x)$, encontrar $C(x) = A(x) \otimes B(x)$

Cálculo con coeficientes

- evaluación: $A(x_0) = \sum_{k=0}^{n-1} a_k x_0^k$ con tiempo en $O(n^2)$
- pero con la regla de Horner $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots x_0(a_{n-2} + x_0(a_{n-1}))\ldots))$ con tiempo en $O(n)$ (usando simplificación=DYC con una sola subinstancia)
- suma: dados $A(x) = \sum_{k=0}^{n} a_k x_k$ y $B(x) = \sum_{k=0}^{n} b_k x_k$ entonces $C(x) = \sum_{k=0}^{n} (a_k + b_k) x_k$ con tiempo $O(n)$
- multiplicación: dados $A(x) = \sum_{k=0}^{n-1} a_k x_k$ y $B(x) = \sum_{k=0}^{n-1} b_k x_k$ entonces $C(x) = \sum_{k=0}^{2n-2} (\sum_{j=0}^{k} a_j b_{k-j}) x^k$ con tiempo $O(n^2)$
- convolución: coincide con la multiplicación, interpretando los polinomios como vectores

Cálculo con raíces

- evaluación: $A(x_0) = c \sum_{k=0}^{n-1} (x_0 - r_k)$ con tiempo en $O(n)$
- suma: no es posible
- multiplicación: dados $A(x) = c^A \sum_{k=0}^{n} (x - r_k^A)$ y $B(x) = c^B \sum_{k=0}^{n} (x - r_k^B)$ entonces $C(x) = c^A c^B \prod_{k=0}^{n-1} (x - r_k^A)(x - r_k^B)$ con tiempo $O(n)$

Cálculo con muestras

- evaluación: es necesario calcular la intrepolación de polinomios para obtener $A(x_0)$ cuyo tiempo es $O(n^2)$
- suma: dados $A(x) = \{(x_i, y_i^A), 0 \leq i \leq n-1\}$ y $B(x) = \{(x_i, y_i^B), 0 \leq i \leq n-1\}$ representados por muestras sobre los mismos puntos x_i, entonces $C(x) = A(x) + B(x) = \{(x_i, y_i^A + y_i^B), 0 \leq i \leq n-1\}$ con tiempo $O(n)$
- multiplicación: dados $A(x) = \{(x_i, y_i^A), 0 \leq i \leq n-1\}$ y $B(x) = \{(x_i, y_i^B), 0 \leq i \leq n-1\}$ representados por muestras sobre los mismos puntos x_i, entonces $C(x) = A(x) \times B(x) = \{(x_i, y_i^A + y_i^B), 0 \leq i \leq n-1\}$ con tiempo $O(n)$

Transformaciones: coeficientes a muestras

- sean $x_i, 0 \leq i \leq n-1$ los puntos donde se quiere muestrear el polinomio, entonces los valores $y_i, 0 \leq i \leq n-1$ se obtienen resolviendo $VA = Y$ donde

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

- usando la regla de Horner se puede resolver en tiempo de $O(n^2)$
- V_n tal que $[V_n]_{jk} = x_j^k$ se denomina la matriz de Vandermonde

Transformaciones: muestras a coeficientes

- dada la representación por muestras $A(x) = \{(x_i, y_i^A), 0 \leq i \leq n-1\}$, se puede resolver $VA = Y$ para hallar A usando eliminación de Gauss, en tiempo de $O(n^3)$
- alternativamente, se puede hallar V_n^{-1} la matriz inversa de Vandermonde, y calcular $A = V^{-1}Y$ en tiempo $O(n^2)$
- V^{-1} se puede hallar en tiempo de $O(n^2)$ usando la fórmula de Lagrange (ejercicio)

Transformaciones a y desde representación por raíces

- si tenemos los polinomios representados por coeficientes o muestras, sólo se pueden obtener algebraicamente la raíces de polinomios grado-acotados por 5
- se pueden obtener las muestras a partir de las raíces hacien n evaluaciones (de tiempo $O(n)$)

Resumen

	Coeficientes	**Raíces**	**Muestras**
evaluación	$O(n)$	$O(n)$	$O(n^2)$
suma	$O(n)$	–	$O(n)$
producto	$O(n^2)$	$O(n)$	$O(n)$

- FFT permitirá la transformación **Coeficientes**$\leftrightarrow$**Muestras** en tiempo $O(n \log n)$, y acotar así también los tiempos de las operaciones no importa su representación

Algoritmo DYC para la transformación

```
                                                     costo
Recursiv-Transf(A,X)
 IF (A.length=1) RETURN A                            Θ(1)
 X^2 ::= {X[i]*X[i], 0<= i<= m-1}                    Θ(m)
 Apar ::=  {A[0], A[2],..., A[n-2]}                  Θ(n)
 Aimpar ::= {A[1],A[2],...,A[n-1]}                   Θ(n)
 Ypar ::= Recursiv_Transf( Apar,X^2)
 Yimpar ::= Recursiv_Transf(Aimpar,X^2)
 Y ::= {Ypar[i]+
        X[i]*Yimpar[i], 0<= i <= m-1}                Θ(m)
RETURN Y
```


Algoritmo DYC para la transformación

- el tiempo de ejecución es, si $|A| = n$ y $|X| = m$,

$$T_{RT}(n,m) = \begin{cases} \Theta(1) & \text{si } n = 1 \\ 2T_{RT}(n/2,m) + \Theta(n+m) & \text{si } n > 1 \end{cases}$$

- por inducción constructiva $T_{RT}(n,m) \in O(nm)$
- como $n = m$ al inicio, entonces es $O(n^2)$, no es suficiente

Tiempo de ejecución

- sería necesario que el tamaño de los puntos de muestras X disminuya al mismo tiempo que los coeficientes A
- se tiene la ventaja de que los puntos de muestras son arbitrarios

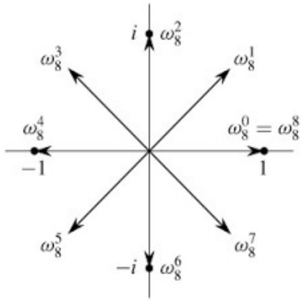
Conjuntos colapsantes

- X es colapsante si $|X^2| = |X|/2$ y X^2 también es colapsante
- como caso base sirve cualquier singleton cuyo elemento no sea 0. Podría ser $X = \{1\}$

$$\begin{array}{ll} |X| = 1 & X = \{1\} \\ |X| = 2 & X = \{1, -1\} \\ |X| = 4 & X = \{1, -1, i, -i\} \\ |X| = 8 & X = \{\pm 1, \pm i, \sqrt{2}/2(\pm 1 \pm i)\} \\ \quad \ldots & \end{array}$$

Raíces enésimas de la unidad

- el conjunto $X_n = \{\omega_n^j : \omega \in \mathbb{C} \wedge \omega_n^{jn} = 1\}$ de raíces enésimas de la unidad para $n = 2^k$ es un conjunto colapsante
- $|X| = n$ con $\omega_n^j = e^{2\pi j/n}, 0 \leq j \leq n-1$ o lo que equivale $\omega_n^j = \cos(2\pi j/n) + i \sin(2\pi j/n)$
- los ω también se llaman números de De Moivre



Algoritmos y Complejidad
Transformada Rápida de Fourier (FFT)

Raíces enésimas de la unidad: propiedades

Lema 7 (Lema de Cancelación)

$\omega_{dn}^{dk} = \omega_n^k$ *para* $n \geq 0, k \geq 0, d > 0$.

Lema 8

Si $n > 0$ *es par, entonces los cuadrados de las raíces n-esimas de la unidad son las* $n/2$ *raíces* $n/2$*-esimas de la unidad.*

Lema 9

$\sum_{j=0}^{n-1} (\omega_k^n)^j = 0$ *para* $n \geq 1, k \neq 0$ *y* k *no divisible por* n

- las demostraciones quedan como ejercicios
- las raíces enésimas de la unidad forman un grupo (asociativa, elemento neutro y elemento simétrico) con la multiplicación, con propiedades similares a $(\mathbb{Z}, +_{mod\ n})$

Algoritmos y Complejidad
Transformada Rápida de Fourier (FFT)

Transformada Rápida de Fourier

- $FFT(A) = Recursiv\_Transf(A, X_n)$ donde X_n son las raíces enésimas de la unidad
- entonces, $n = m$ y vale $T_{FFT}(n) = 2T_{FFT}(n/2) + \Theta(n) \in \Theta(n \log n)$
- de esta manera podemo realizar cualquier operación sobre polinomios en tiempo $\Theta(n \log n)$, usando las implementaciones más eficientes posiblemente combinadas con transformaciones
- para que esto sea cierto necesitamos la transformada inversa a la FFT, de muestras a coeficientes
- la transformada discreta de Fourier (DFT) es la evaluación de un polinomio A en los puntos X_n determinados por las raíces enésimas de la unidad
- en $DFT(A) = \{y_k = \omega_n^k\} = \{\sum_{j=0}^{n-1} a_j \omega_n^{kj}\}$

Algoritmos y Complejidad
Transformada Rápida de Fourier (FFT)

Algoritmo eficiente para FFT

```
FFT(A)
  n::= A.length
 IF (n=1) RETURN A
 wn ::= e^2*PI*i/n; w ::= 1
 Apar ::= {A[0],A[2],...,A[n-2]}
 Aimpar ::= {A[1],A[3],...,A[n-1]}
 Ypar ::= Recursiv_Transf(Apar)
 Yimpar ::= Recursiv_Transf(Aimpar)
 FOR k::=0 TO n/2-1
   Y[k] ::= Ypar[k]+w*Yimpar[k]
   Y[k+n/2] ::= Ypar[k]-w*Y[Yimpar[k]
   w ::= w*wn
RETURN Y
```

Transformada inversa

- permite ir de la representación por muestras a la de coeficientes

Lema 10
$V_n^{-1} = \bar{V}_n/n$

Demostración.
Se calcula $[V_n^{-1}V_n]_{jk} = \sum_{m=0}^{n-1}(\omega_n^{-mj}/n)(\omega_n^{mk}) = \sum_{m=0}^{n-1}\omega_n^{k(j-k)}/n$. □

- entonces $IFFT(Y) = Recursiv\_Transf(Y, \bar{X}_n/n)$, y también es en $\Theta(n \log n)$

Convolución de dos vectores

Lema 11 (Teorema de convolución)
Sean a, b dos vectores de longitud $n = 2^k$ entonces

$$a \otimes b = IFFT_{2n}(FFT_{2n}(a) \cdot FFT_{2n}(b))$$

donde los vectores a, b se completan con 0s hasta la longitud $2n$, y $\cdot$ representa la multiplicación componente a componente.