Θέματα Κατανεμημένων και Παράλληλων Συστημάτων

Μάθημα #3

Αλγόριθμοι Κοινής Μνήμης

Υπολογισμός Προθεμάτων

Έστω $*$ μία δυαδική προσεταιριστική πράξη, η οποία ορίζεται πάνω σε ένα σύνολο *X*.

Δοθέντος ενός διανύσματος *A[1..n]*, *n* στοιχείων του συνόλου *X*, το πρόβλημα του *υπολογισμού προθεμάτων των στοιχείων* του *A*, ζητάει να υπολογισθούν οι επόμενες *n* ποσότητες:

$$S[i] = A[1] * A[2] * \ldots * A[i], \quad i = 1, \ldots, n$$

Προφανής ακολουθιακός αλγόριθμος προθεμάτων:

S[1]:=A[1]

Υπολόγισε το *S[i+1]* για *i=1,…,n-1*, ως εξής:

*S[i+1]:=S[i]*A[i+1]*

Ο αλγόριθμος χρειάζεται *O(n)* χρόνο και είναι συμφυώς (inherently) ακολουθιακός.

Παράλληλος Αλγόριθμος PREFIX_COMPUTATION

begin

1. if $(n = 1)$ **then**

begin

$S[1] := A[1]$

exit

end

2. for $1 \le i \le n/2$ **pardo**

$B[i] := A[2i-1] * A[2i]$

3. Υπολόγισε με αναδρομικό τρόπο τις ποσότητες προθεμάτων

$Q[i] = B[1] * B[2] *\ldots * B[i], \quad i = 1, \ldots, n/2$

4. for $1 \le i \le n/2$ **pardo**

if $(i = 1)$ **then**

$S[1] := A[1]$

else if $(i \bmod 2 = 1)$ **then**

$S[i] := Q[(i-1)/2] * A[i]$

else /* ο i είναι άρτιος αριθμός

$S[i] := Q[i/2]$

end

Λήμμα 4.1 ***Ο αλγόριθμος PREFIX_COMPUTATION υπολογίζει σωστά τα προθέματα ενός διανύσματος A, n στοιχείων, σε χρόνο T(n)=O(log n) εκτελώντας συνολικά, W(n)=O(n) λειτουργίες.***

Απόδειξη. **Η ορθότητα αποδεικνύεται με επαγωγή στο *k*, όπου 2^k *ο αριθμός των στοιχείων του A***

Η βάση της επαγωγής, *k=0* ή *n=1*, αντιμετωπίζεται σωστά από το βήμα 1 του αλγορίθμου.

Επαγωγική υπόθεση: **Ο αλγόριθμος υπολογίζει σωστά τα προθέματα διανυσμάτων με $2^k=n/2$ στοιχεία.**

Επαγωγικό βήμα: **Θα δείξουμε ότι ο αλγόριθμος υπολογίζει σωστά, τα προθέματα διανυσμάτων με $2^{k+1} = n$ στοιχεία.**

Από την επαγωγική υπόθεση, το διάνυσμα *Q* το οποίο υπολογίζεται στο βήμα 3, αποτελείται από τις ποσότητες προθεμάτων των στοιχείων του διανύσματος *B* το οποίο υπολογίζεται στο βήμα 2.

Συγκεκριμένα,

$Q[i] = B[1]*B[2]*\ldots*B[i]$ **για** $i = 1, \ldots, n/2,$

και επειδή $B[i] = A[2i-1]*A[2i]$ ***για*** $i = 1, \ldots, n/2,$ **έχουμε:**

$Q[i] = A[1]*A[2]*\ldots *A[2i-1]*A[2i], \quad i = 1, \ldots, n/2$

Επομένως, $Q[i] = S[2i]$ **για** $i=1, \ldots, n/2$

$S[i]=Q[i/2]$, **για** $i=1, \ldots, n.$

Περίπτωση 1: ***i*** **άρτιος. Τότε ο αλγόριθμος υπολογίζει σωστά τις ποσότητες:**

S[i]=Q[i/2], **για** ***i=1, … ,n.***

Περίπτωση 2: ***i*** **είναι περιττός. Τότε έχουμε τις εξής περιπτώσεις:**

- ***i =1,*** **οπότε** ***S[1] = A[1].***
- ***i ≠1,*** **οπότε αν** ***i=2j+1,*** **τότε** ***S[i] = S[2j+1] = S[2j] *A[2j+1]*** **και συνεπώς,** ***S[i] = Q[(i-1)/2] *A[i].***

Πολυπλοκότητα αλγορίθμου:

- **Το βήμα 1 χρειάζεται σταθερό χρόνο και σταθερό αριθμό λειτουργιών. Τα βήματα 2 και 4 απαιτούν σταθερό παράλληλο χρόνο και *O(n)* λειτουργίες.**
- **Επειδή ο αλγόριθμος καλείται αναδρομικά στο βήμα 3 σε είσοδο μεγέθους *n/2*, ο συνολικός χρόνος *T(n)* και το έργο *W(n)* δίνονται από τις αναδρομικές σχέσεις:**

$$T(n) = T(n/2) + c$$

$$W(n) = W(n/2) + c'n$$

όπου c και c' σταθερές. Άρα,

$$T(n)=\log n \text{ και } W(n)=O(n)$$

*Παράλληλος **Μη** Αναδρομικός Αλγόριθμος*
PREFIX_COMPUTATION

Prefix_Computation*(A, n, '+')*

for *k = 1* to *log n* do

for all *i in {1,2,..n}* in parallel do

***jump* = 2^{k-1}**

if (*i* ≥ *jump* +1) then

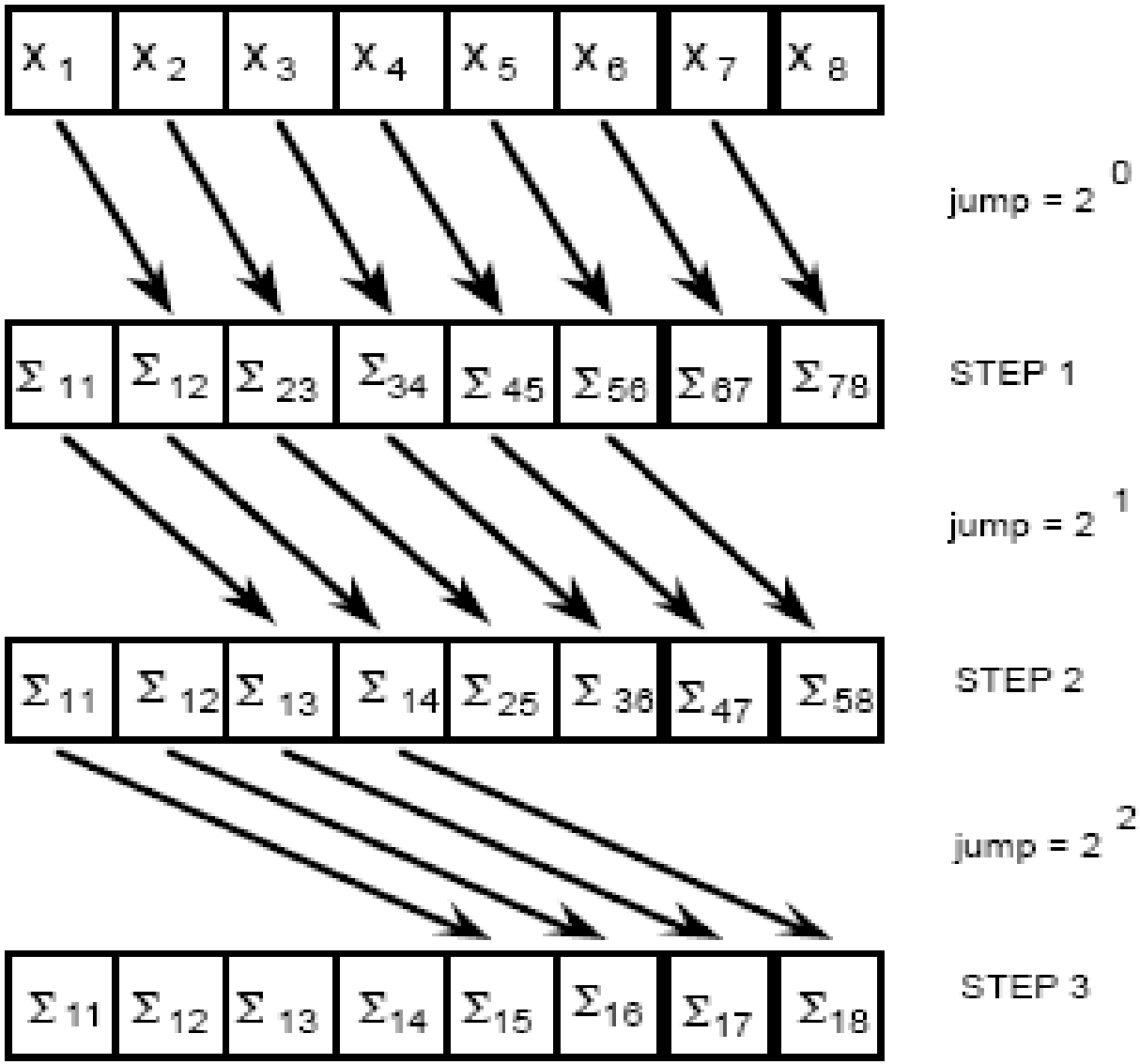
A[i] = A[i - jump] + A[i]

end if

end for

end for

ΠΑΡΑΔΕΙΓΜΑ


x 1 x 2 x 3 x 4 x 5 x 6 x 7 x 8
jump = 2 0
Σ 11 Σ 12 Σ 23 Σ34 Σ 45 Σ56 Σ67 Σ78
STEP 1
jump = 2 1
Σ 11 Σ 12 Σ 13 Σ 14 Σ25 Σ 36 Σ47 Σ58
STEP 2
jump = 2 2
Σ 11 Σ 12 Σ 13 Σ14 Σ15 Σ16 Σ17 Σ18
STEP 3

Suffix Computation

Έστω $*$ μία δυαδική προσεταιριστική πράξη, η οποία ορίζεται πάνω σε ένα σύνολο X.

Δοθέντος ενός διανύσματος $A[1..n]$, n στοιχείων του συνόλου X, το πρόβλημα του ζητάει να υπολογισθούν οι επόμενες n ποσότητες:

$$\Sigma_{in} = A[i] * A[i+1] * \ldots * A[n], \quad i = 1, \ldots, n-1$$

$$\Sigma_{nn} = A[n]$$

Υπολογισμός Suffix Sum σε διάνυσμα *A*, *n* στοιχείων

```
Suffix_Sum (A, n, '+')
    for k = 1 to log n do
        for all i in {1,2,..n} in parallel do
            jump = 2^(k-1)
            if ( i <= n-jump) then
                A[i] = A[i] + A[ i + jump]
            end if
        end for
    end for
```

ΠΑΡΑΔΕΙΓΜΑ

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	
								jump=1
Σ12	Σ23	Σ34	Σ45	Σ56	Σ67	Σ78	Σ88	
								jump=2
Σ14	Σ25	Σ36	Σ47	Σ58	Σ68	Σ78	Σ88	
								jump=4
Σ18	Σ28	Σ38	Σ48	Σ58	Σ68	Σ78	Σ88	

Η Τεχνική του Διπλασιασμού των Δεικτών

Πρόβλημα: **Δοθείσης μιας συνδεδεμένης λίστας *n* κόμβων ζητείται να αποκτήσει κάθε κόμβος την ετικέτα του τελευταίου κόμβου.**

Λύση: **Σε κάθε βήμα, κάθε κόμβος της λίστας παράλληλα, αντικαθιστά τον δείκτη του με τον δείκτη του επόμενου κόμβου στη λίστα. Σε $\lceil \log n \rceil$ το πολύ, βήματα, κάθε κόμβος της λίστας "δείχνει" στον τελευταίο κόμβο της λίστας.**

Εφαρμογή: **Εύρεση της Απόστασης Κάθε Στοιχείου της Λίστας από τον Τελευταίο Κόμβο**

Αλγόριθμος *LIST_RANKING*

Είσοδος: **Η διαδοχή των κόμβων μιας συνδεδεμένης λίστας *L*, $n=2^k$ στοιχείων, περιέχεται στο διάνυσμα *NEXT[1…n]*. *Για* κάθε *i*, το *NEXT[i]* περιέχει έναν δείκτη στον επόμενο του *i*. Αν *i* είναι ο τελευταίος κόμβος, τότε *NEXT[i]=0*.**

Έξοδος: **Για κάθε κόμβο *i*, η απόσταση *DIST[i]* του κόμβου *i* από τον τελευταίο κόμβο της λίστας.**

```
begin
  1. for 1 ≤ i ≤ n  pardo
        M[i] := NEXT[i]
        if  ( M[i] = 0 ) then  DIST[i] := 0
                       else DIST[i] := 1
  2. for 1 ≤ h ≤ log n  do
        for 1 ≤ i ≤ n  pardo
           if  ( M[i] != 0 ) then
              begin
                 DIST[i] := DIST[i] + DIST[M[i]]
                 M[i] := M[M[i]]
              end
end
```

Παράδειγμα:

	A	B	C	D	E	F	G	H
NEXT:	2	3	0	7	1	4	8	5

L F → $\overline{D}$ → G → H → E → A → B → C

Χρονική Μονάδα 1 και 2:

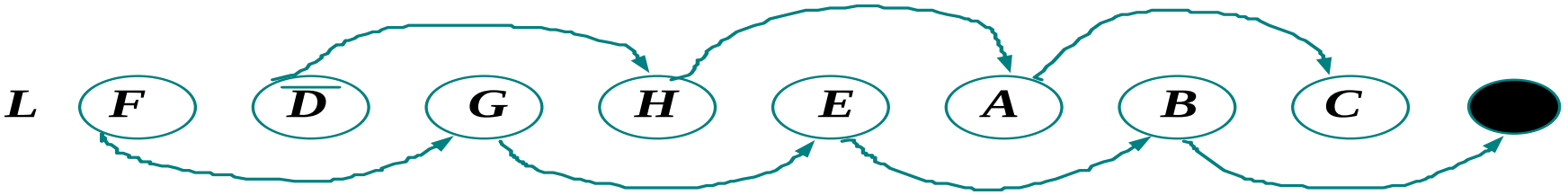
M:	2	3	0	7	1	4	8	5
DIST:	1	1	0	1	1	1	1	1

L F → $\overline{D}$ → G → H → E → A → B → C → ●

Χρονική Μονάδα 3 και 4:

M:	3	0	0	8	2	7	5	1

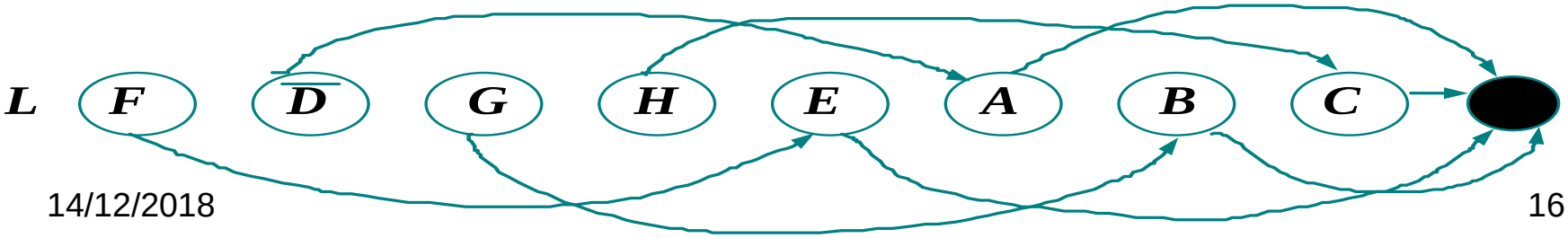
DIST:	2	1	0	2	2	2	2	2



Χρονική Μονάδα 5 και 6:

M:	0	0	0	1	0	5	2	3

DIST:	2	1	0	4	3	4	4	4



Χρονική Μονάδα 7 και 8:

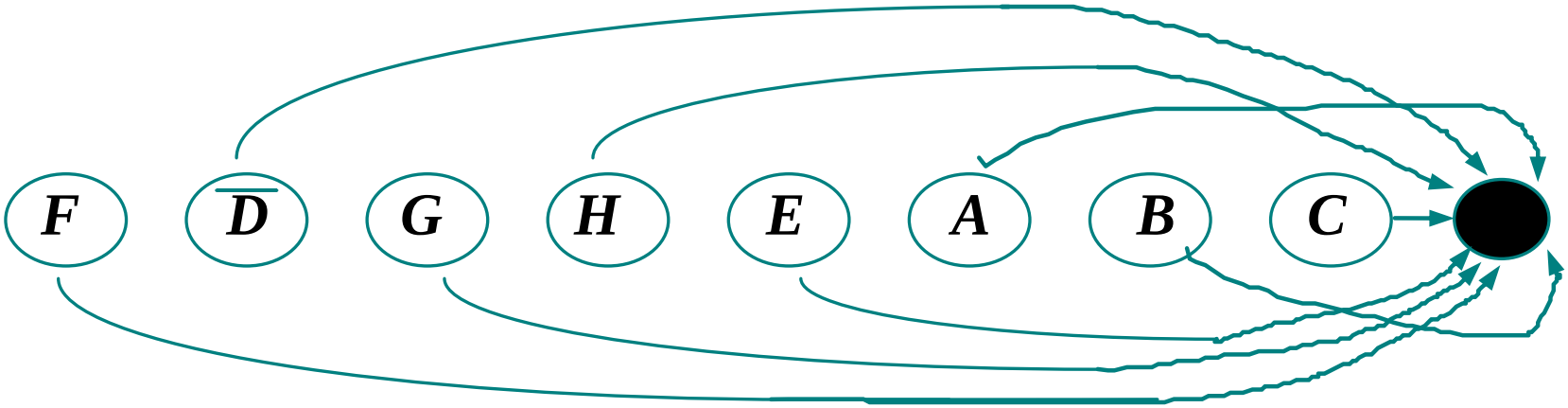
M:

0	0	0	0	0	0	0	0

DIST:

2	*1*	*0*	*6*	*3*	*7*	*5*	*4*

L:



Στο τέλος της επανάληψης *h* του βήματος 2, για *h=1,…, log n*, ισχύει ότι:

"Για κάθε i=1,…,n, όταν M[i]≠0, τότε το στοιχείο DIST[i] περιέχει την απόσταση του i από το M[i], ενώ όταν M[i]=0, τότε το DIST[i] περιέχει την απόσταση του i από το τέλος της λίστας."

Στο τέλος της τελευταίας επανάληψης, για κάθε *i*, το στοιχείο *DIST[i]* θα περιέχει την απόσταση του *i* από το τέλος της λίστας.

Ο χρόνος ο οποίος απαιτείται για την εκτέλεση του αλγορίθμου είναι *O(log n)* ενώ συνολικά εκτελούνται *O(nlog n)* λειτουργίες.

Λήμμα: Ο αλγόριθμος LIST_RANKING υπολογίζει σωστά την τάξη κάθε στοιχείου μιας συνδεδεμένης λίστας L n στοιχείων σε χρόνο T(n)=O(log n) εκτελώντας συνολικά, W(n)=O(n log n) λειτουργίες.

Αλγόριθμοι Ταξινόμησης

Αλγόριθμος MERGE-SORT

Είσοδος: **`Ενα διάνυσμα $X=(x_1,x_2,\ldots,x_n)$, $n=2^k$.**

***`Εξοδος:*** **`Ενα πλήρες δυαδικό δένδρο *n* φύλλων, τέτοιο ώστε σε κάθε επίπεδο *h*, $1 \le h \le \log n$, και σε κάθε κόμβο *v(i)* στο επίπεδο *h*, $1 \le i \le n/2^h$, αντιστοιχεί η ταξινομημένη λίστα $LIST_{v(i)}(h)$, η οποία περιέχει όλα τα στοιχεία του υποδένδρου με ρίζα *v(i)*.**

begin

1. for $1 \le i \le n$ pardo

$LIST_{v(i)}(0)$ **:= X(i)**

2. for $h = 1$ to $\log n$ do

for $1 \le i \le n/2^h$ pardo

συγχώνευσε τις λίστες $LIST_{v(2i-1)}(h-1)$ και $LIST_{v(2i)}(h-1)$ στην $LIST_{v(i)}(h)$

end

Παράδειγμα

`Εστω X=(10, 3, 60, 70, 11, 2, 90, 9). Τότε, το βήμα 1 το αλγορίθμου μας δίνει:

$LIST_1(0)=10,\ LIST_2(0)=3,\ LIST_3(0)=60,\ LIST_4(0)=70,\ LIST_5(0)=11,\ LIST_6(0)=2,\ LIST_7(0)=90,\ LIST_8(0)=9,$

ενώ οι 3 επαναλήψεις του βήματος 2 δημιουργούν τις ακόλουθες λίστες:

$1^η$ επανάληψη *(h=1):*

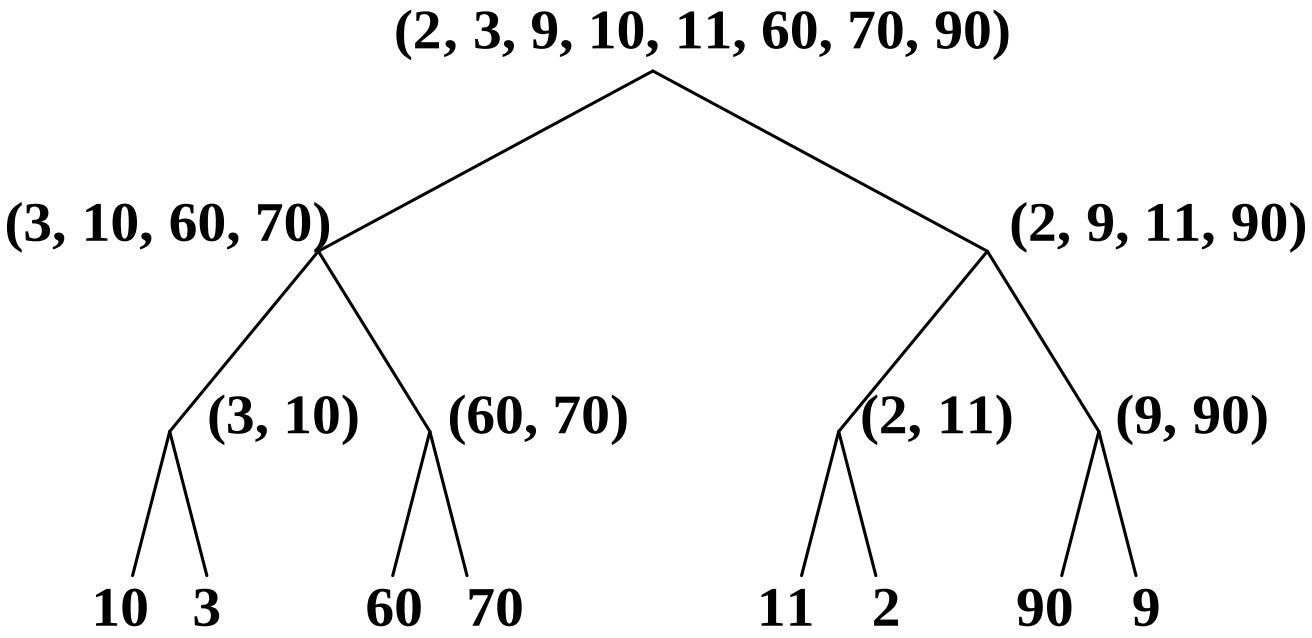
$LIST_1(1)=(3, 10),\ LIST_2(1)=(60, 70),\ LIST_3(1)=(2, 11),\ LIST_4(1)=(9, 90).$

$2^η$ επανάληψη *(h=2):*

$LIST_1(2)=(3, 10, 60, 70),\ LIST_2(2)=(2, 9, 11, 90).$

$3^η$ επανάληψη *(h=3):*

$LIST_1(3)=(2, 3, 9, 10, 11, 60, 70, 90)$



Το δυαδικό δένδρο του αλγορίθμου *MERGE-SORT* για *n=8*.

Λήμμα. Δοθείσης μιας ακολουθίας $X=(x_1,x_2,\ldots,x_n)$ ο αλγόριθμος MERGE-SORT ταξινομεί τα στοιχεία της X σε χρόνο $O(\log^2 n)$, εκτελώντας συνολικά $O(n\log n)$ λειτουργίες.

Απόδειξη.

Το βήμα 1 απαιτεί σταθερό χρόνο και $O(n)$ λειτουργίες.

Το βήμα 2 αποτελείται από $O(\log n)$ επαναλήψεις, και κάθε επανάληψη απαιτεί $O(\log n)$ χρόνο και $O(n)$ λειτουργίες, εφόσον χρησιμοποιηθεί ένας αλγόριθμος συγχώνευσης, ο οποίος απαιτεί $O(\log n)$ χρόνο και εκτελεί $O(n)$ λειτουργίες.

Επομένως, ο αλγόριθμος είναι βέλτιστος, απαιτεί συνολικά $O(\log^2 n)$ χρόνο και εκτελεί $O(n\log n)$ λειτουργίες.

Η τεχνική «διαίρει και βασίλευε» (divide and conquer)

Βήματα τεχνικής:

1. **Διάσπαση της εισόδου σε *p* τμήματα, τα οποία έχουν περίπου τα ίδια μεγέθη.**
2. **Αναδρομική επίλυση του προβλήματος σε κάθε ένα από τα *p* τμήματα της εισόδου.**
3. **Σύνθεση ή συγχώνευση των λύσεων των *p* υποπροβλημάτων σε μία λύση του προβλήματος στο σύνολο της εισόδου.**

Το πιο δύσκολο μέρος στην εφαρμογή της αποτελεί η συγχώνευση των λύσεων σε μία ενιαία λύση.

Παραλλαγή της τεχνικής αποτελεί η *τεχνική της διαμέρισης (partitioning strategy)*

Η τεχνική της διαμέρισης (partitioning strategy)

Βήματα τεχνικής:

1. **Τη διάσπαση του προβλήματος σε *p* ανεξάρτητα υποπροβλήματα τα οποία έχουν περίπου τα ίδια μεγέθη, όπου *p* ο αριθμός των διαθέσιμων επεξεργαστών.**
2. **Την ταυτόχρονη επίλυση των p υποπροβλημάτων με την χρήση των p επεξεργαστών.**

Η διάσπαση του προβλήματος σε υποπροβλήματα θα πρέπει να γίνει έτσι ώστε, οι λύσεις των υποπροβλημάτων να μπορούν εύκολα να συντεθούν σε μία λύση.

Το πιο δύσκολο μέρος στην εφαρμογή της τεχνικής αποτελεί η διάσπαση του προβλήματος σε ανεξάρτητα υποπροβλήματα.

Συγχώνευση Ακολουθιών

Έστω $R \subseteq A \times A$, η οποία είναι ολική διάταξη (ήτοι η R είναι ανακλαστική, αντισυμμετρική, και μεταβατική, και για κάθε ζεύγος στοιχείων $a, b \in A$, είτε $(a, b) \in R$ ή $(b, a) \in R$).

Αν $X = (x_1, x_2, \ldots, x_n)$ και $Y = (y_1, y_2, \ldots, y_m)$ είναι ακολουθίες στοιχείων του A, οι οποίες είναι διαταγμένες σύμφωνα με τη σχέση R, το πρόβλημα της συγχώνευσης (merging) των X και Y ζητεί να προσδιορισθεί η διαταγμένη ακολουθία $Z = (z_1, z_2, \ldots, z_{n+m})$, η οποία έχει ως στοιχεία τα στοιχεία των ακολουθιών X και Y.

Στη συνέχεια υποθέτουμε ότι η σχέση R είναι η *μικρότερο ή ίσο* ($\leq$) και το A είναι το σύνολο των θετικών ακεραίων.

Παράδειγμα. **Αν $X = (7, 13, 19)$ και $Y = (2, 15, 30, 100)$ τότε $Z = (2, 7, 13, 15, 19, 30, 100)$.** ■

Αλγόριθμος διαμέρισης

Υποθέτουμε ότι, τα στοιχεία των *X* και *Y* είναι ξένα μεταξύ τους, οι *X* , *Y* έχουν *n* στοιχεία, οι αριθμοί *log n* και *n/log n* είναι ακέραιοι.

Ορισμός. ***Έστω*** $X = (x_1, x_2, \ldots, x_n)$ **μία ακολουθία στοιχείων του συνόλου *A* και** $a \in A$**. Η *τάξη (rank)* του *a* στην *X (rank(a:X))* είναι ο αριθμός των στοιχείων της *X*, που είναι μικρότερα ή ίσα με το *a*.**

Δοθέντων δύο διαταγμένων κατά αύξουσα σειρά ακολουθιών *X* και *Y*, ο αλγόριθμος διαμέρισης

- **διασπά την *X* σε *n/log n* υποακολουθίες** $X_0, X_1, \ldots, X_{n/\log n-1}$**, οι οποίες έχουν τα ίδια μεγέθη,**
- **διασπά την *Y* σε *n/log n* υποακολουθίες** $Y_0, Y_1, \ldots, Y_{n/\log n-1}$**, έτσι ώστε τα στοιχεία της υποακολουθίας** Y_{i-1} **να είναι μικρότερα από τα στοιχεία της** X_i **,** $i=1,\ldots,n/\log n-1$**, ενώ κάποιες από τις** $Y_0, Y_1, \ldots, Y_{n/\log n-1}$ **μπορεί να είναι κενές.**

Αλγόριθμος PARTITION

Είσοδος: **Δύο ακολουθίες** $X=(x_1,x_2,\ldots,x_n)$ **και** $Y=(y_1,y_2,\ldots,y_n)$ **διαταγμένες κατά αύξουσα σειρά.**

`*Εξοδος:* $n/\log n$ **υποακολουθίες** $X_0, X_1, \ldots, X_{n/\log n-1}$, **και** $n/\log n$ **υποακολουθίες** $Y_0, Y_1, \ldots, Y_{n/\log n-1}$, **τέτοιες ώστε** $|X_i|=\log n$, $i=0,\ldots,n/\log n-1$, ***και*** **όλα τα στοιχεία των** Y_{i-1} ***και*** X_{i-1} **είναι μικρότερα από όλα τα στοιχεία των** Y_i ***και*** X_i , $i=1,\ldots,n/\log n-1$.

begin

1. $m(0) := 0$
2. **for** $1 \le i \le n/\log n - 1$ **pardo**

   $m(i) := rank(x_{i\log n}:Y)$
3. $m(n/\log n) := n$
4. **for** $0 \le i \le n/\log n-1$ **pardo**

   $X_i := (x_{i\log n+1}, x_{i\log n+2},\ldots,x_{(i+1)\log n})$

   $Y_i := (y_{m(i)+1}, y_{m(i)+2},\ldots,y_{m(i+1)})$

end

Επειδή η ακολουθία *Y* είναι διαταγμένη, ο υπολογισμός της τάξης ενός στοιχείου x της ακολουθίας *X* στην *Y* γίνεται με τη χρήση της *μεθόδου της δυαδικής αναζήτησης* (*binary search method*).

Παράδειγμα. **Έστω** $X = (3, 6, 7, 12, 19, 30, 45, 60, 65, 80, 90, 100, 110, 120, 130, 180)$ **και** $Y = (2, 10, 11, 85, 88, 92, 98, 105, 109, 140, 200, 205, 208, 210, 220, 230)$. **Η εκτέλεση του αλγορίθμου *PARTITION* δίνει τις εξής ακολουθίες:**

$X0 = (3, 6, 7, 12)$ $\quad Y0 = (2, 10, 11)$

$X1 = (19, 30, 45, 60)$ $\quad Y1 = \varnothing$

$X2 = (65, 80, 90, 100)$ $\quad Y2 = (85, 88, 92, 98)$

$X3 = (110, 120, 130, 180)$ $\quad Y3 = (105, 109, 140, 200, 205, 208, 210, 220, 230)$

.

(Αρκεί να παρατηρήσουμε ότι $m(0)=0,\ m(1)=3,\ m(2)=3,\ m(3)=7$ **και** $m(4)= 16$**.)** ■

Λήμμα: *Δοθέντων δύο ακολουθιών* $X=(x_1,x_2,\ldots,x_n)$ *και* $Y=(y_1,y_2,\ldots,y_n)$, *διαταγμένων κατά αύξουσα σειρά, ο αλγόριθμος PARTITION υπολογίζει σωστά n/logn υποακολουθίες* $X_0, X_1, \ldots, X_{n/\log n-1}$ *και n/logn υποακολουθίες* $Y_0, Y_1, \ldots, Y_{n/\log n-1}$, *τέτοιες ώστε* $|X_i|=\log n$, $i=0,\ldots, n/\log n-1$, *και όλα τα στοιχεία των* Y_{i-1} *και* X_{i-1} **είναι μικρότερα από όλα τα στοιχεία των** Y_i *και* X_i, $i=1,\ldots,n/\log n-1$. *Ο αλγόριθμος χρειάζεται* $O(\log n)$ *βήματα και εκτελεί συνολικά* $O(n)$ *λειτουργίες.*

Απόδειξη.

Η ορθότητα του αλγορίθμου είναι εύκολο να δειχθεί. Αρκεί να παρατηρήσουμε ότι αφού $m(i) := rank(x_{ilog\ n}:Y)$, θα πρέπει $y_{m(i)} < x_{ilogn} < y_{m(i)+1}$. Επομένως, τα στοιχεία της Y_{i-1} είναι μικρότερα από τα στοιχεία της Xi , και τα στοιχεία της X_{i-1} είναι μικρότερα από τα στοιχεία της Y_i , $i=1,\ldots, n/log\ n-1$.

Τα βήματα 1 και 3 απαιτούν σταθερό χρόνο και σταθερό αριθμό λειτουργιών.

Το βήμα 2 απαιτεί $O(log\ n)$ χρόνο και $O(log\ n * n/log\ n)=O(n)$ αριθμό λειτουργιών, διότι η δυαδική αναζήτηση απαιτεί χρόνο $O(log\ n)$ και εκτελείται παράλληλα για όλα τα στοιχεία $x_{ilog\ n}$, $i=1,\ldots,n/log\ n-1$.

Το βήμα 4 απαιτεί σταθερό χρόνο και $O(n)$ αριθμό λειτουργιών. Επομένως, ο αλγόριθμος χρειάζεται συνολικά $O(log\ n)$ χρόνο και $O(n)$ λειτουργίες και είναι βέλτιστος. ■

Αλγόριθμος MERGE

Είσοδος: Δύο ακολουθίες $X=(x_1,x_2,\ldots,x_n)$ και $Y=(y_1,y_2,\ldots,y_n)$ διαταγμένες κατά αύξουσα σειρά.

`Εξοδος: Μία διαταγμένη ακολουθία $Z = (z_1, z_2, \ldots, z_{2n})$, η οποία έχει ως στοιχεία τα στοιχεία των ακολουθιών X και Y.

begin

1. **Υπολόγισε τις υποακολουθίες $X_0, X_1, \ldots, X_{n/\log n-1}$, και $Y_0, Y_1, \ldots, Y_{n/\log n-1}$, εφαρμόζοντας τον αλγόριθμο *PARTITION*.**
2. **for $0 \le i \le n/\log n-1$ pardo**

   2.1 if $|Y_i| \le \log n$ then

   συγχώνευσε (merge) τις X_i και Y_i σε μία υποακολουθία Z_i /* με τη χρήση ενός γραμμικού ακολουθιακού αλγορίθμου */

   2.2 else ($|Y_i| > \log n$)

   2.2.1 εκτέλεσε τον αλγόριθμο *PARTITION* με είσοδο τις ακολουθίες Y_i και X_i. `Εστω $Y_{i1}, Y_{i2}, \ldots, Y_{i\lceil |Yi/\log |Yi|\rceil}$ και $X_{i1}, X_{i2}, \ldots, X_{i\lceil |Yi/\log |Yi|\rceil}$ οι υποακολουθίες, που αποτελούν την έξοδο του αλγορίθμου.

   2.2.2 for $0 \le j \le \lceil |Yi|/\log |Yi| \rceil - 1$ pardo

   συγχώνευσε (merge) τις Y_{ij} και X_{ij} σε μία υποακολουθία Z_{ij}

   2.2.3 $Z_i := (Z_{i1}, Z_{i2}, \ldots, Z_{i\lceil |Yi/\log |Yi|\rceil})$

3. **$Z := (Z_1, Z_2, \ldots, Z_{n/\log n})$**

end

Λήμμα Δοθέντων δύο ακολουθιών $X=(x_1,x_2,\ldots,x_n)$ και $Y=(y_1,y_2,\ldots,y_n)$, διαταγμένων κατά αύξουσα σειρά, ο αλγόριθμος MERGE συγχωνεύει τις X και Y σε μία διαταγμένη ακολουθία $Z = (z_1, z_2, \ldots, z_{2n})$, η οποία έχει ως στοιχεία τα στοιχεία των ακολουθιών X και Y, σε χρόνο O(log n), εκτελώντας συνολικά O(n) λειτουργίες.

<u>Απόδειξη.</u> Τα στοιχεία της διαταγμένης ακολουθίας Z_{i-1}, τα οποία προσδιορίζονται από τη συγχώνευση των Y_{i-1} και X_{i-1}, είναι μικρότερα από τα στοιχεία της Z_i, τα οποία προσδιορίζονται από τη συγχώνευση των Y_i και X_i, *i=1,…, n/log n-1.*

Ο υπολογισμόÂΤ Ākολουθίας Z_i, *i=1,…, n/logn-1*, από το βήμα 2, γίνεται ως εξής:

1. **Όταν $|Y_i| \leq log\ n$, καλείται ένας ακολουθιακός αλγόριθμος συγχώνευσης, ο οποίος συγχωνεύει τις ακολουθίες Y_i και X_i μήκους το πολύ $log\ n$, στην ακολουθία Z_i μήκους το πολύ $2log\ n$. Έτσι, το βήμα 2.1 απαιτεί $O(log\ n)$ χρόνο.**

2. **Όταν $|Y_i| > log\ n$, καλείται ο *PARTITION* με είσοδο τις ακολουθίες Y_i (η οποία παίζει το ρόλο της X) και X_i (η οποία παίζει το ρόλο της Y). Παρατηρούμε ότι, οι υποακολουθίες $Y_{i1}, Y_{i2}, \ldots, Y_{i\lceil |Yi/log\ |Yi|\rceil}$ και $X_{i1}, X_{i2}, \ldots, X_{i\lceil |Yi/log\ |Yi|\rceil}$, που αποτελούν την έξοδο του αλγορίθμου, έχουν μήκος το πολύ ίσο με $logn$. `Έτσι, το βήμα 2.2.1 απαιτεί το πολύ $O(log\ n)$ χρόνο και $O(n)$ λειτουργίες. Το βήμα 2.2.2 εκτελείται με τη κλήση ενός ακολουθιακού αλγορίθμου συγχώνευσης, και αφού οι υποακολουθίες Y_{ij} και X_{ij} έχουν μήκος το πολύ ίσο με $log\ n$, ο χρόνος που απαιτείται, είναι το πολύ $O(log\ n)$ και ο αριθμός των λειτουργιών που εκτελούνται, είναι το πολύ $O(n)$.**

Επειδή το βήμα 1 απαιτεί επίσης $O(log\ n)$ χρόνο και $O(n)$ λειτουργίες, συμπεραίνουμε ότι ο αλγόριθμος *MERGE* είναι βέλτιστος και απαιτεί χρόνο $O(log\ n)$, εκτελώντας συνολικά $O(n)$ λειτουργίες. ■

Πιθανοτικός Αλγόριθμος Γρήγορης Ταξινόμησης

Ακολουθιακός αλγόριθμος

Με είσοδο διάνυσμα *A[1..n]*:

- Επιλέγεται τυχαία ένας αριθμός από το *A* ως pivot στοιχείο.
- Το *A* διαμερίζεται (αναδιατάσσεται) σε δύο υποδιανύσματα *A[1..q]* και *A[q+1..n]* έτσι ώστε κάθε στοιχείο του *A[1..q]* να είναι μικρότερο ή ίσο κάθε στοιχείου του *A[q+1..n]*.
- Τα *A[1..q]* και *A[q+1..n]* ταξινομούνται μέσω αναδρομικών κλήσεων του ίδιου του αλγορίθμου.

Παράλληλος Πιθανοτικός Αλγόριθμος Γρήγορης Ταξινόμησης

RANDOMIZED_QUICKSORT(A[p..r])

Είσοδος: Διάνυσμα *A[p..r]* με ξένα μεταξύ τους στοιχεία.

Έξοδος: Ταξινομημένα τα στοιχεία του *A[p..r]*

begin

1. *n := r – p+1*
2. επέλεξε ένα τυχαίο στοιχείο a από το διάνυσμα *A[p..r]*
4. *q:= SPLIT(A[p..r],*a*)*
5. **in parallel do**

   RANDOMIZED_QUICKSORT(A[p.. q-1])

   RANDOMIZED_QUICKSORT(A[q+1.. r])

end

Η Συνάρτηση *SPLIT(A[p..r],a)*

Η *SPLIT(A[p..r],a)* διαμοιράζει τα στοιχεία του *A[p..r]* στα διανύσματα *A[p..q-1]* και *A[q+1..r]* ως εξής:

- Ορίζει το δυαδικό διάνυσμα *L[1..n]* *(n := r – p+1)* με *L[i]=1* αν και μόνον εάν *A[p+i-1]* < a και το διάνυσμα *G[1..n]* με *G[i]=1* αν και μόνον εάν *A[p+i-1]* > a .

- Υπολογίζει τα αθροίσματα προθεμάτων *PSL[1..n]* και *PSG[1..n]* *τ*ων διανυσμάτων *L[1..n]* και *G[1..n]* αντίστοιχα.

- Χρησιμοποιεί τα *PSL[1..n]* , *PSG[1..n]* και τοποθετεί τα *q-1* στοιχεία του *A* που είναι μικρότερα από το a, στις πρώτες *q-1* θέσεις του *A* και τα *r-q* στοιχεία του *A* που είναι μεγαλύτερα από το a, στις τελευταίες *r- q* θέσεις του *A*.

SPLIT(A[p..r],α)

Είσοδος: Διάνυσμα *A[p..r]* με ξένα μεταξύ τους στοιχεία και ένα στοιχείο α του *A[p..r]*

Έξοδος: Τα διανύσματα *A[p..q-1]* και *A[q+1..r]* έτσι ώστε τα στοιχεία του *A[p..q-1]* να είναι μικρότερα του α και τα στοιχεία του *A[q+1..r]* να είναι μεγαλύτερα του α. Επιστρέφει το δείκτη *q*, *A[q]*=α.

```
begin
    array L[1..n], G[1..n], PSL[1..n], PSG[1..n]
    1.  n := r – p+1
    2.  if  n = 1 then
                return p
    3. for 1 ≤ i ≤ n pardo
            if   A[p+i-1] < α   then   L[i]:= 1
                                else   L[i]:= 0
            if   A[p+i-1] > α   then   G[i]:= 1
                                else   G[i]:= 0
```

4. *PSL[1..n]:=PREFIX_COMPUTATION(L,n,+)*

5. *PSG[1..n]:=PREFIX_COMPUTATION(G,n,+)*

6. *q: = p + PSL[n]*

7. *A[q] :=* α

8. **for** $1 \leq i \leq n$ **pardo**

if *A[p+i-1]* < α **then**

A[p+PSL[i]-1] := A[p+i-1]

else **if** *A[p+i-1]* > α **then**

A[q+PSG[i]] := A[p+i-1]

9. **return** *q*

end

Παράδειγμα

A:	3	18	9	1	15	2	20	14	4	16
α = 9										
L:	1	0	0	1	0	1	0	0	1	0
PSL:										
	1	1	1	2	2	3	3	3	4	4
G:	0	1	0	0	1	0	1	1	0	1
PSG:										
	0	1	1	1	2	2	3	4	4	5

$q=1+PSL[10]=5$ και $A[5]=9$.

Επειδή *A[1] < 9, A[4] < 9, A[6] < 9, A[9] < 9,*

A[1+PSL[1]-1]=A[1] := A[1]=3

A[1+PSL[4]-1]=A[2] := A[4]=1

A[1+PSL[6]-1]=A[3] := A[6]=2

A[1+PSL[9]-1]=A[4] := A[9]=4

Επειδή *A[2] > 9, A[5] > 9, A[7] > 9, A[8] > 9, A[10] > 9,*

A[5+PSG[2]]=A[6]:= A[2]=18

A[5+PSG[5]]=A[7]:= A[5]=15

A[5+PSG[7]]=A[8]:= A[7]=20

A[5+PSG[8]]=A[9]:= A[8]=14

A[5+PSG[10]]=A[10]:= A[10]=16

Η έξοδος του αλγορίθμου έχει ως εξής:

A[1..4]:	**3**	**1**	**2**	**4**	**A[5]=9**
A[6..10]:	**18**	**15**	**20**	**14**	**16**