Θεμελιώδη Θέματα Επιστήμης Υπολογιστών

ΣΗΜΜΥ – ΣΕΜΦΕ ΕΜΠ

Ενότητα 4η:

Αριθμητικοί υπολογισμοί

Επιμέλεια διαφανειών: Στάθης Ζάχος, Άρης Παγουρτζής

Αριθμητικοί υπολογισμοί

- Εύρεση ΜΚΔ (Ευκλείδειος αλγόριθμος)
- Ύψωση σε δύναμη
- Αριθμοί Fibonacci
- Πολλαπλασιασμός ακεραίων
- Divide-and-Conquer
- Επίλυση αναδρομών: master theorem

Εύρεση Μέγιστου Κοινού Διαιρέτη (gcd)

Δεν είναι λογικό να ανάγεται στο πρόβλημα εύρεσης πρώτων παραγόντων γιατί αυτό δεν λύνεται αποδοτικά.

Απλός αλγόριθμος: $O(\min(a,b))$

$z := \min(a,b)$

while $(a \bmod z \neq 0)$ **and** $(b \bmod z \neq 0)$ **do** $z := z\text{-}1$

Αλγόριθμος με αφαιρέσεις: $O(\max(a,b))$

$i := a \; ; \; j := b$
while $(i \neq j)$ **do** **if** $i > j$ **then** $i := i - j$ **else** $j := j - i$
return i

Αλγόριθμος του Ευκλείδη: $O(\log(a+b))$

$i := a \; ; \; j := b$
while $(i > 0)$ **and** $(j > 0)$ **do**
  if $i > j$ **then** $i := i \bmod j$ **else** $j := j \bmod i$
return $i + j$

Εύρεση Μέγιστου Κοινού Διαιρέτη (gcd): υλοποίηση με αναδρομή

Αλγόριθμος με αφαιρέσεις: $O(\max(a,b))$

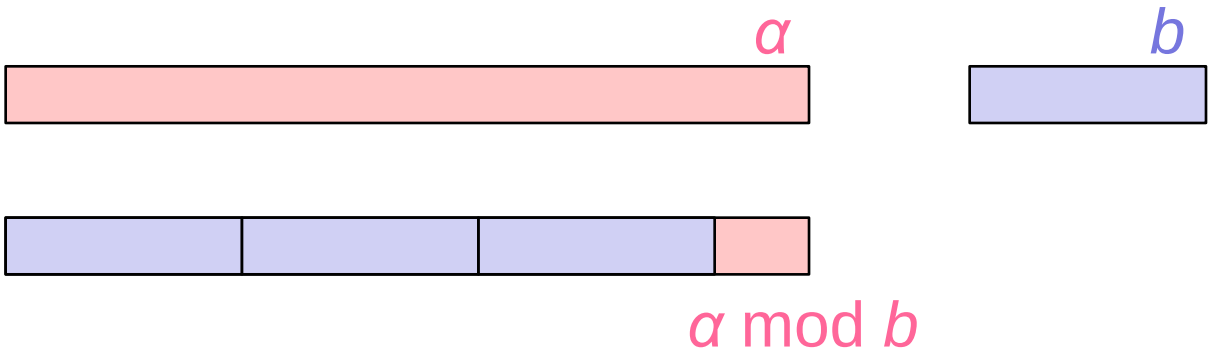
```
if a=b then GCD(a,b):=a
else if a>b then GCD(a,b):= GCD(a-b,b)
            else GCD(a,b):= GCD(a,b-a)
```

Αλγόριθμος του Ευκλείδη: $O(\log(a+b))$

```
if b=0 then GCD(a,b):= a
       else GCD(a,b):= GCD(b, a mod b)
```

Πολυπλοκότητα Ευκλείδειου

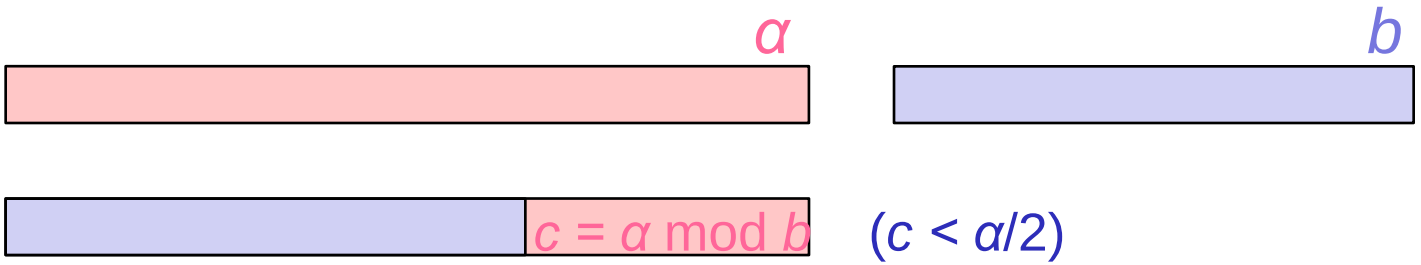
- O(log max(α,b)): σε κάθε 2 επαναλήψεις το πολύ ο μεγαλύτερος αριθμός υποδιπλασιάζεται:
  - Περίπτ. 1. αρχικά: **(α, b)**, $b \le \alpha/2$



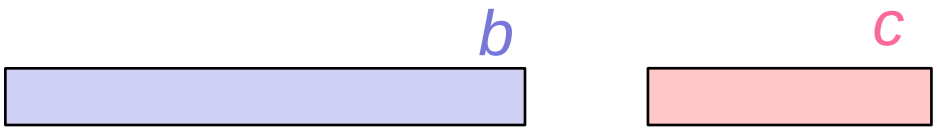
  - σε 1 επανάληψη: **(b, α mod b)**

Πολυπλοκότητα Ευκλείδειου

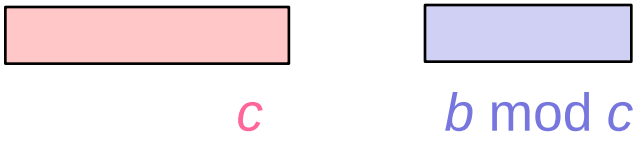
- O(log max(α,b)): σε κάθε 2 επαναλήψεις το πολύ ο μεγαλύτερος αριθμός υποδιπλασιάζεται:
  - Περίπτ. 2. αρχικά: **(α, b)**, $b > \alpha/2$



  - σε 1 επανάληψη: **(b, c)**



  - σε 2 επαναλήψεις: **(c, b mod c)**



Πολυπλοκότητα Ευκλείδειου Αλγόριθμου

- $O(\log \max(\alpha,b))$: σε κάθε 2 επαναλήψεις το πολύ ο μεγαλύτερος αριθμός υποδιπλασιάζεται
- $\Omega(\log \max(\alpha,b))$: για ζεύγη διαδοχικών αριθμών Fibonacci F_{k-1}, F_k, ο αλγόριθμος κάνει k επαναλήψεις (γιατί;), και $k = \Theta(\log F_k)$, αφού $F_k \approx \varphi^k/\sqrt{5}$,

  (ισχύει $F_k = [\varphi^k - (1-\varphi)^k]/\sqrt{5}$, $\varphi = (1+\sqrt{5})/2$: *χρυσή τομή*)
- Άρα η πολυπλοκότητα του Ευκλείδειου είναι $\Theta(\log \max(\alpha,b)) = \Theta(\log (\alpha+b))$
- Bit complexity: $O(\log^3(\max(\alpha,b)))$

Ύψωση σε δύναμη

```
power(a, n)
    result := 1;
    for i := 1 to n do
        result := result*a;
    return result
```

Πολυπλοκότητα: $O(n)$ – εκθετική! (γιατί;)

Ύψωση σε δύναμη

```
power(a, n)
    result := 1;
    for i := 1 to n do
        result := result*a;
    return result
```

Πολυπλοκότητα: $O(n)$ – εκθετική! (γιατί;)

ως προς το μήκος της εισόδου:

$O(n) = (2^{\log n}) = O(2^{\text{len}(n)})$

$\log n < \text{len}(n) \leq \log n + 1$

... με επαναλαμβανόμενο τετραγωνισμό (Gauss)

```
fastpower(a, n)
    result := 1;
    while n>0 do
        if odd(n) then result:=result*a;
        n := n div 2;
        a := a*a
    return result
```

Ιδέα: $a^{13} = a^{8+4+1} = a^8\, a^4\, a^1$

Πολυπλοκότητα: $O(\log n)$ - πολυωνυμική

ως προς το μήκος της εισόδου: $O(\log n) = (\text{len}(n))$

Παράδειγμα σε Python

```
def fastpower(a,n):
  res=1
  while (n>0):
     if (n%2==1):
       res=res*a
     print (n, a, res)
     n=n//2
     a=a*a
  return res
```

Εκτέλεση: print (fastpower(15, 507))

Παράδειγμα σε Python: εκτέλεση

```
a^n computation: a=15, n=507
----------------------------
n a res
----------------------------
507 15 15
253 225 3375
126 50625 3375
63 2562890625 8649755859375
31 6568408355712890625 568151286615952849388122558593 75
15 43143988327398919500410556793212890625
24512312477955347693397999733408432199155413400148972868919372558593 75
7
18614037287947342154674106047557028201233642050738126272335648536682128906 25
45627309847847775440487100544956051260580836489724452716204137204571502556329707022452196723171746311478388924420812600146746262907981872558593 75
```

Παράδειγμα σε Python: εκτέλεση

```
. . . . .
1
120050042531184094299874716051889779577766242877999774926621
511666739054560393666116288901943297970025344754226257166624
937808359064757447892755473928062711328282864028330039243822
312184801024809026818185212475825302998589583459405014766541
044631750295336452974762075918135906249517574906349182128906
25
189787821718375110360762239350954129966731376201805406772602
442272623305647775089332670743514799079383484405610052572959
489905303006182040948649980490955066029181857035802223386098
036059919330482483657720145859824904754399181629709999530085
887946894416436785712070915319327998703108254211607359805831
177526935506324973606577448389957910588756698435943358452701
472306966001783648996571417819411171853449747320497842238598
203675133090389640160833787160530771936112655849181000704050
329471980267281185338842889154689834276189225020172717011238
228371563475037862855909764903117320500314235687255859375
```

Παράδειγμα σε Python: εκτέλεση

```
15 ^ 507 =
189787821718375110360762239350954129966731376201805406772602
442272623305647775089332670743514799079383484405610052572959
489905303006182040948649980490955066029181857035802223386098
036059919330482483657720145859824904754399181629709999530085
887946894416436785712070915319327998703108254211607359805831
177526935506324973606577448389957910588756698435943584527014
723069660017836489965714178194111718534497473204978422385982
036751330903896401608337871605307719361126558491810007040503
294719802672811853388428891546898342761892250201727170112382
283715634750378628559097649031173205003142356872558593750
```

Παρατήρηση: a^n έχει *τεράστιο* πλήθος ψηφίων: $O(n \log a)$.
Bit complexity *αναγκαστικά* εκθετική! (βλ. και παρακάτω)

Modular exponentiantion: a^n mod p

Ένας *αποδοτικός* και *πρακτικά χρήσιμος* αλγόριθμος
(έλεγχος πρώτων κατά Fermat: a^{n-1} mod n =? 1)

```
fastmodpower(a,n,p)
    result := 1;
    while n>0 do
       if odd(n) then res:=res*a mod p;
       n := n div 2;
       a := a*a mod p
    return res
```

Arithmetic complexity: $O(\log n) = O(\text{len}(n))$

Bit complexity: $O(\log n \log^2 p)$ - πολυωνυμική

ως προς το μήκος της εισόδου: $O(\text{len}(n) \cdot \text{len}(p)^2)$

Παράδειγμα σε Python

```
def fastmodpower(a,n,p):
  res=1
  while (n>0):
     if (n%2==1):
       res=res*a % p
     print (n, a, res)
     n=n//2
    a=a*a % p
  return res
```

Εκτέλεση: print (fastmodpower(15,126,127))

(Fermat primality test: $a^{n-1} \bmod n =? 1$)

Παράδειγμα σε Python: εκτέλεση

```
Fermat Primality Test

a^(n-1) mod n computation: a=15, n=127
-----------------------------
n a res
-----------------------------
126 15 1
63 98 98
31 79 122
15 18 37
7 70 50
3 74 17
1 15 1
1
```

Bit complexity για υπολογισμό a^n ;

(*προαιρετικό, εκτός ύλης*)

- 'Αφελής' αλγόριθμος, *i*-οστή επανάληψη: πολ/μός a με $a^{i\text{-}1}$

  $O((i\text{-}1)\log(a)^2)$ Συνολικά: $O(n^2\log(a)^2) = O(4^{\text{len}(n)}\,\text{len}(a)^2)$

  μη αποδοτικός!
- Αλγόριθμος τετραγωνισμού: $O(n^2\log(a)^2)$ επίσης! (γιατί;)
- Τετραγωνισμός με πολλ/σμό Gauss-Karatsuba:

  $O(n^{\log 3}\log a^{\log 3}) = O(3^{\text{len}(n)}\text{len}(a)^{1.59})$ [*προσεχώς*!]
- Περαιτέρω βελτιώσεις [Harvey, van der Hoeven]:

  $O(n\log a\,(\log n + \log\log\alpha)) = O(2^{\text{len}(n)}\text{len}(a)\,(\text{len}(n) + \log(\text{len}(\alpha)))$ ...

  καλύτερος, αλλά επίσης *μη αποδοτικός*
- *Εγγενώς δύσκολο* πρόβλημα: a^n έχει ***εκθετικό*** πλήθος bits

Αριθμοί Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

$$F_0 = 0,\ F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2},\ n >= 2$$

- Πρόβλημα: Δίνεται n, να υπολογιστεί το F_n
- Πόσο γρήγορο μπορεί να είναι το πρόγραμμά μας;

Αριθμοί Fibonacci – αναδρομικός αλγόριθμος

```
Fib1(n)
   if (n<2) then return n
   else return Fib1(n-1)+Fib1(n-2)
```

- Πολυπλοκότητα: $T(n) = T(n\text{-}1) + T(n\text{-}2) + c$,

  δηλ. η $T(n)$ ορίζεται ουσιαστικά όπως η $F(n)$ (συν μια σταθερά), οπότε αποδεικνύεται επαγωγικά ότι:

  $$T(n) \geq c' F(n) = \Omega(1.618^n) = \Omega(1.618^{2^{\text{len}(n)}})$$

  (*διπλά εκθετική* ως προς μέγεθος εισόδου!)

Αριθμοί Fibonacci – καλύτερος αλγόριθμος

```
Fib2(n)
    a:=0; b:=1;
    for i:=2 to n do
        c:=b; b:=a+b; a:=c;
    return b
```

- Πολυπλοκότητα: $O(n)$
- Είναι πολυωνυμική;

Αριθμοί Fibonacci – καλύτερος αλγόριθμος

```
Fib2(n)
    a:=0; b:=1;
    for i:=2 to n do
        c:=b; b:=a+b; a:=c;
    return b
```

- Πολυπλοκότητα: $O(n)$
- Είναι πολυωνυμική; **Όχι!** : $O(n) = O(2^{\text{len}(n)})$

Αριθμοί Fibonacci – ακόμα καλύτερος αλγόριθμος

Μπορούμε να γράψουμε τον υπολογισμό σε μορφή πινάκων:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

Από αυτό συμπεραίνουμε:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

… και το πλήθος των αριθμητικών πράξεων (*αριθμητική πολυπλοκότητα*) μειώνεται σε $O(\log n) = O(\text{len}(n))$

Αριθμοί Fibonacci – ακόμα καλύτερος αλγόριθμος

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

***Άσκηση*:**

- έστω ότι θέλουμε να βρούμε F(*n*) mod *t*, για κάποιο δοσμένο ακέραιο *t* . Ποιος αλγόριθμος υπερτερεί;

- βρείτε την *πολυπλοκότητα ψηφιοπράξεων (bit complexity)* του παραπάνω αλγορίθμου. Συγκρίνετε με τον επαναληπτικό αλγόριθμο.

Σημαντική αλγοριθμική τεχνική

Ποια ιδέα είναι κοινή και στους 3 προηγούμενους αλγόριθμους (Ευκλείδη, Repeated Squaring, Fibonacci με πίνακα);

Divide-and-Conquer !

(Διαίρει-και-Κυρίευε ή Διαίρει-και-Βασίλευε)

Divide-and-Conquer

- Χρονική πολυπλοκότητα αλγόριθμων «διαίρει-και-κυρίευε» με διατύπωση και λύση αναδρομικής εξίσωσης χρόνου εκτέλεσης.
- **MergeSort**
  - ***T*(*n*)** : χρόνος για ταξινόμηση *n* στοιχείων.
    - ***T*(*n* / 2)** : ταξινόμηση αριστερού τμήματος (*n* / 2 στοιχεία).
    - ***T*(*n* / 2)** : ταξινόμηση δεξιού τμήματος (*n* / 2 στοιχεία).
    - **O(*n*)** : συγχώνευση ταξινομημένων τμημάτων.

$$T(n) = 2\,T(n/2) + O(n),\ T(1) = O(1)$$

  - Χρόνος εκτέλεσης MergeSort: ***T*(*n*) = ?**

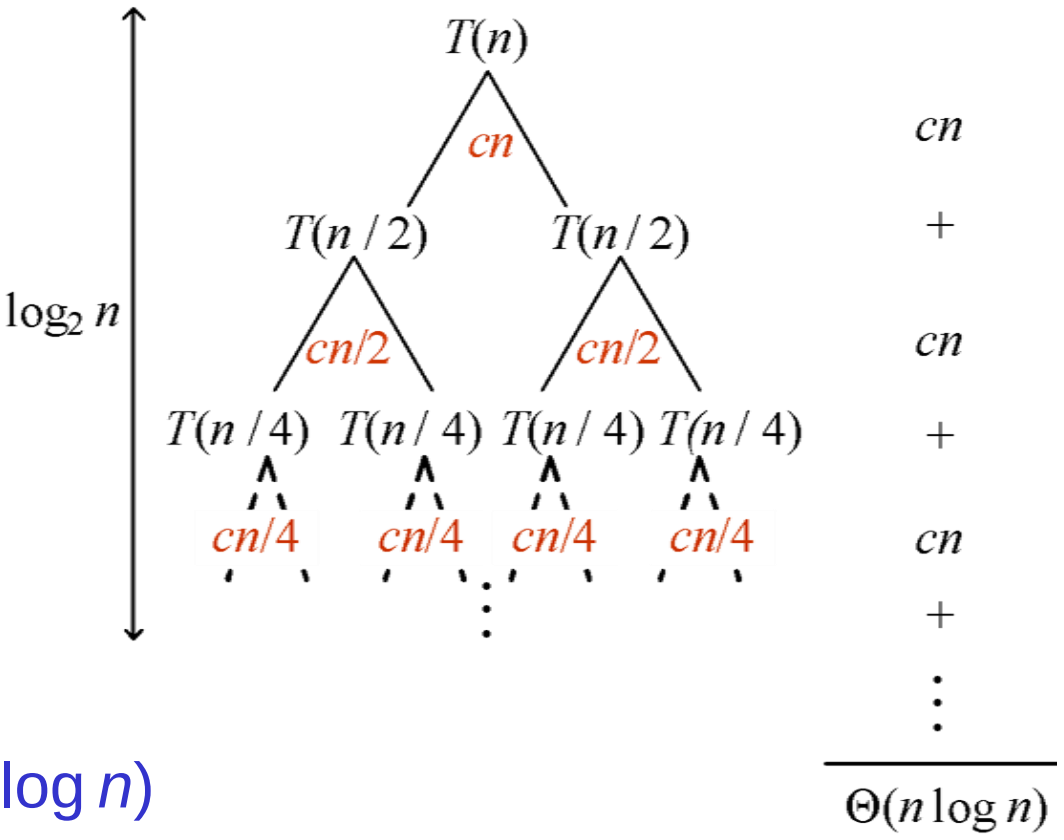
Δέντρο Αναδρομής

$T(n) = 2\,T(n/2) + O(n)$,
$T(1) = O(1)$

Δέντρο αναδρομής :

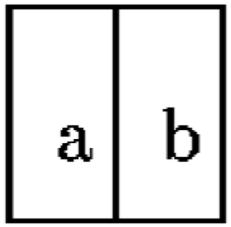
Ύψος : $O(\log n)$
#κορυφών : $O(n)$

Χρόνος / επίπεδο : $O(n)$

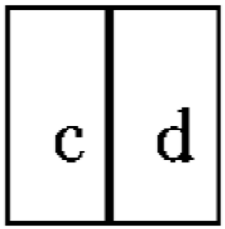


Συνολικός χρόνος : $O(n \log n)$

Πολλαπλασιασμός Ακεραίων

X : 

$= a \cdot 2^{\frac{n}{2}} + b$

Y : 

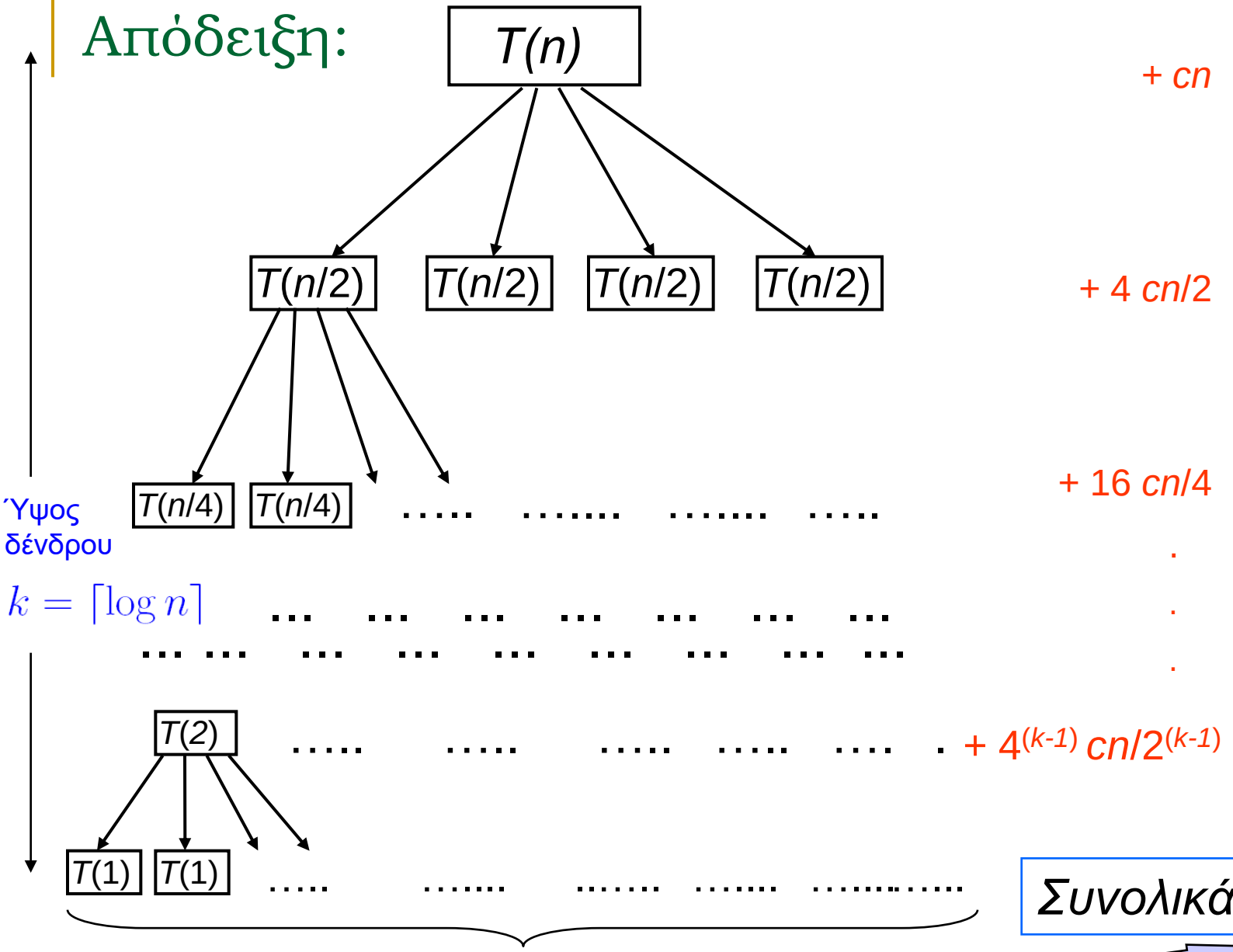
$= c \cdot 2^{\frac{n}{2}} + d$

$$X\ Y = ac \cdot 2^n + (ad + bc) \cdot 2^{\frac{n}{2}} + bd$$

Πολυπλοκότητα Πολλαπλασιασμού

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 4T(\frac{n}{2}) + cn & , \text{για } n > 1 \end{cases}$$

Απόδειξη:



4^k 'φύλλα', χρονική πολυπλ/τα $\alpha \cdot 4^k = O(n^2)$

Πολυπλοκότητα Πολλαπλασιασμού

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 4T(\frac{n}{2}) + cn & , \text{για } n > 1 \end{cases}$$

$$T(n) = O(n^2)$$

Βελτιωμένος Πολλαπλασιασμός (Gauss-Karatsuba)

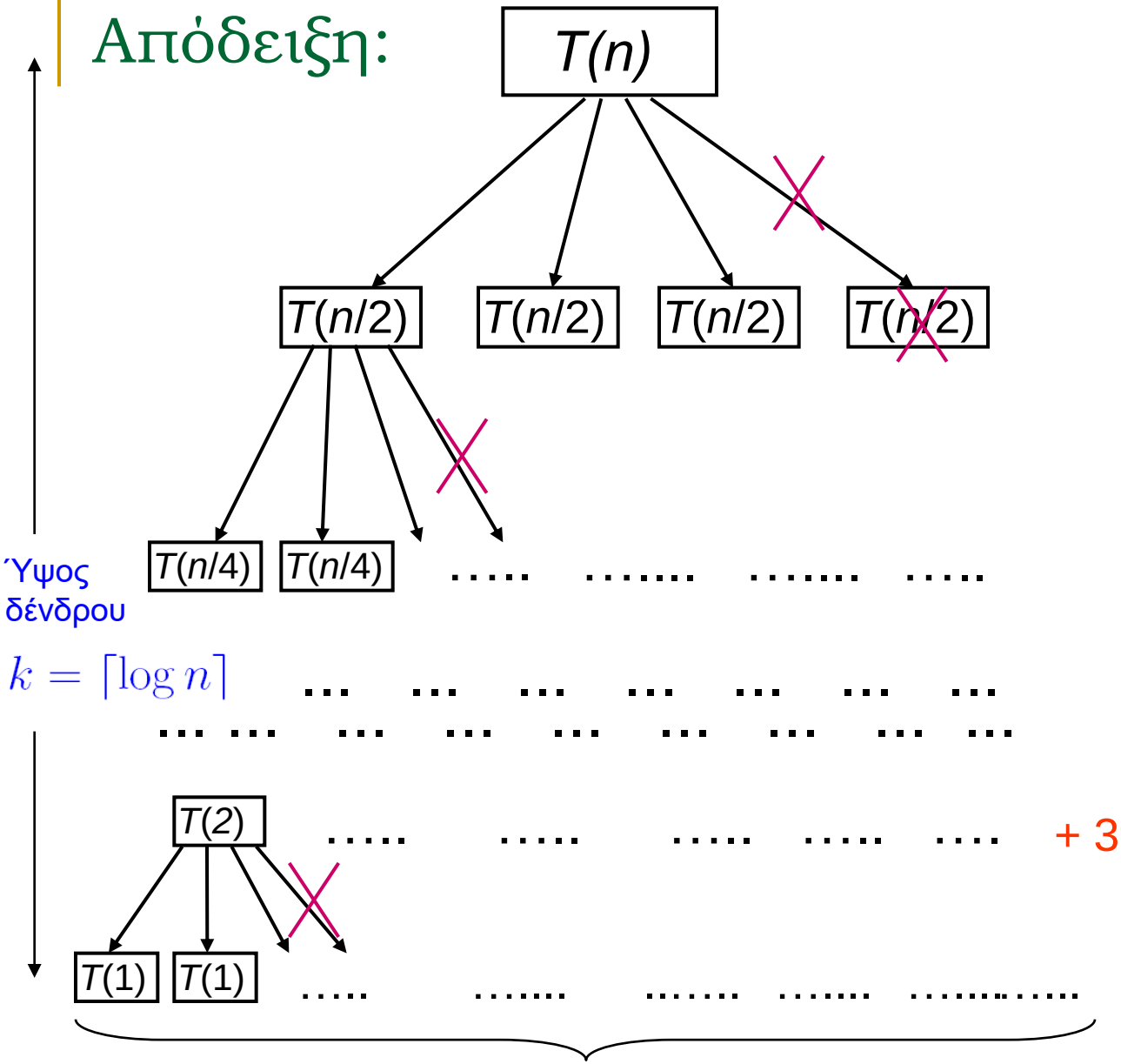
$$(ad + bc) = [(a - b)(d - c) + ac + bd]$$

$$X\ Y = ac \cdot 2^n + [(a - b)(d - c) + ac + bd]\ 2^{\frac{n}{2}} + bd$$

Πολυπλοκότητα Βελτίωσης

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 3T(\frac{n}{2}) + cn & , \text{για } n > 1 \end{cases}$$

Απόδειξη:



3^k 'φύλλα', χρονική πολυπλ/τα $\alpha \cdot 3^k = O(3^{\log n}) = O(n^{\log 3})$

Πολυπλοκότητα Βελτίωσης

$$T(n) = \begin{cases} a & , \text{για } n = 1 \\ 3T(\frac{n}{2}) + cn & , \text{για } n > 1 \end{cases}$$

$$T(n) = O(n^{\log_2 3}) = O(n^{1.59})$$

Master Theorem (απλή μορφή)

Αν $T(n) = aT(n/b) + O(n)$,

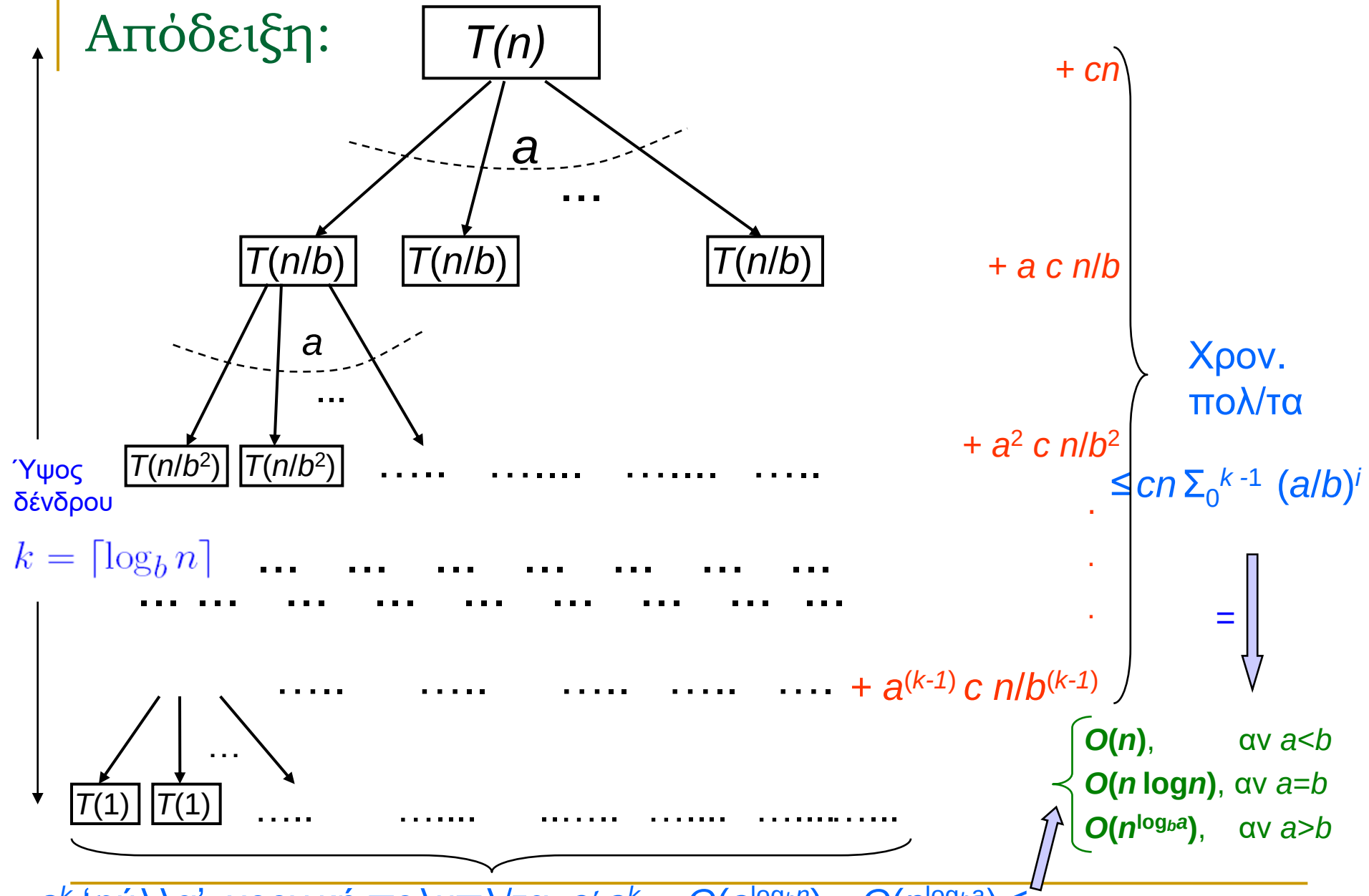
για θετικούς ακέραιους a, b

και $T(1) = O(1)$

τότε:

$$T(n) = \begin{cases} O(n), & \text{αν } a<b \\ O(n \log n), & \text{αν } a=b \\ O(n^{\log_b a}), & \text{αν } a>b \end{cases}$$

Απόδειξη:


T(n)
a
T(n/b)
T(n/b)
T(n/b)
a
T(n/b²)
T(n/b²)
Ύψος δένδρου
k = ⌈log_b n⌉
T(1)
T(1)
+ cn
+ a c n/b
+ a² c n/b²
+ a^(k-1) c n/b^(k-1)
Χρον. πολ/τα
≤ cn Σ_0^(k-1) (a/b)^i
=
O(n), αν a<b
O(n logn), αν a=b
O(n^(log_b a)), αν a>b
a^k 'φύλλα', χρονική πολυπλ/τα c'·a^k = O(a^(log_b n)) = O(n^(log_b a)) ≤

Master Theorem (γενική μορφή)

Αν $T(n) = aT(n/b) + O(n^d)$,

για θετικούς ακέραιους a, b, d

και $T(1) = O(1)$

τότε:

$$T(n) = \begin{cases} O(n^d), & \text{αν } a<b^d \\ O(n^d \log n), & \text{αν } a=b^d \\ O(n^{\log_b a}), & \text{αν } a>b^d \end{cases}$$

Master Theorem: εφαρμογή

Αν $T(n) = aT(n/b) + O(n^d)$, για θετικούς ακέραιους a, b, d

και $T(1) = O(1)$ τότε:

$$T(n) = \begin{cases} O(n^d), & \text{αν } a<b^d \\ O(n^d \log n), & \text{αν } a=b^d \\ O(n^{\log_b a}), & \text{αν } a>b^d \end{cases}$$

Matrix Multiplication

- 'Standard' divide-and-conquer: $T(n) = 8T(n/2) + O(n^2)$

  $\Rightarrow T(n) = O(n^3)$

- Strassen's algorithm: $T(n) = 7T(n/2) + O(n^2)$

  $\Rightarrow T(n) = O(n^{\log 7})$

Αλγόριθμοι divide & conquer

$O(\log n)$	if x<A[n/2] search(A[1,n/2])…	δυαδική αναζήτηση
$O(\log^3(\max(a,b)))$	GCD(a,b) := GCD(b,a **mod** b)	εύρεση ΜΚΔ
$O(\log n \log^2 a)$	modpow(a,n,p) := modpow(a^2,n/2,p)	modular exponentiation
$O(\log n)$ *	αλγόριθμος πίνακα	υπολογισμός n-οστού
[*αριθμ. πολυπλ/τα]	fast doubling	αριθμού Fibonacci
$O(n \log n)$	mergesort(A[1, n/2])	ταξινόμηση
	mergesort(A[n/2+1, n])	με συγχώνευση
	merge(A[1, n/2], A[n/2+1, n])	
$O(n^{\log 3})$	Gauss-Karatsuba	πολλ/σμός n-bit αριθμών
$O(n^{\log 7})$	Strassen	πολλ/σμός πινάκων n x n