# Υπολογιστικά Συστήματα: Αλγόριθμοι

*Α. Ντελόπουλος*

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Νοέμβριος 2003

# Περιεχόμενα

# Κεφάλαιο 1

# Οι αλγόριθμοι και τα χαρακτηριστικά τους

## 1.1 Τί είναι οι αλγόριθμοι;

Ένας αλγόριθμος είναι μία συνταγή που περιγράφει μία συγκεκριμένη διαδικασία επίλυσης ενός προβλήματος.

Ας σκεφτούμε, για παράδειγμα, το πρόβλημα της ανεύρεσης του μεγίστου μεταξύ $n$ πραγματικών αριθμών. Η εκτέλεση του παρακάτω αλγορίθμου παρέχει τη λύση του εν λόγω προβλήματος αν ο $n \times 1$ πίνακας $A$ περιέχει τους $n$ αριθμούς:

**Αλγόριθμος 1.** *Εύρεση του μεγίστου n αριθμών* [1]

```
function max(A)
max_val:=A(1)
for i=2:n
   if (max_val < A(i))
   then max_val:=A(i)
   endif
end
return max_val
```

Είναι ενδιαφέρον να σχολιάσουμε ότι η διαδικασία που περιγράφεται από έναν αλγόριθμο πρέπει να είναι αρκετά *γενική* ώστε η "εκτέλεσή" του να οδηγεί στην επίλυση του αντίστοιχου

[1] Για την περιγραφή των βημάτων του αλγορίθμου χρησιμοποιείται παραπάνω μία ψευδογλώσσα με σχετικά προφανή ερμηνεία. Στην πράξη οποιαδήποτε γλώσσα προγραμματισμού μπορεί να την αντικαταστήσει.

προβλήματος ανεξάρτητα από τις εκάστοτε τιμές των παραμέτρων του. Για παράδειγμα, εύκολα μπορεί να ελεγχθεί ότι ο παραπάνω αλγόριθμος δίνει την επιθυμητή λύση ανεξάρτητα (α) του πλήθους $n$ και (β) των συγκεκριμένων τιμών $A(i)$, $i = 1, \ldots, n$. Ωστόσο είναι επιτρεπτό ένας αλγόριθμος να δίνει τη λύση υπό καλά προσδιορισμένες *συνθήκες*. Για παράδειγμα ο παρακάτω αλγόριθμος (2) λύνει το πρόβλημα εύρεσης του μεγίστου υπό τη συνθήκη $A(i) \geqslant 0$:

**Αλγόριθμος 2.** *Εύρεση του μεγίστου n μη αρνητικών αριθμών*

```
function max(A)
max_val:=0
for i=1:n
   if (max_val < A(i))
   then max_val:=A(i)
   endif
end
return max_val
```

Οι αλγόριθμοι περιέχουν βήματα που κατά περίπτωση αντιστοιχούν σε πράξεις, προσπελάσεις στην μνήμη, ελέγχους αληθείας διαφόρων συνθηκών (π.χ. συγκρίσεις) κ.λπ.

## 1.2 Χαρακτηριστικά των αλγορίθμων

Δοθέντος ενός συγκεκριμένου προβλήματος η κατασκευή ενός αλγορίθμου, κατά προτίμηση ενός καλού αλγορίθμου, που να προσδιορίζει την επίλυσή του είναι το αυτονόητο ζητούμενο. Στη διαδικασία κατασκευής του αλγορίθμου θα αναφερόμαστε με τον όρο *σχεδίαση*. Η σχεδίαση αλγορίθμων είναι κατά βάση μια νοητική διαδικασία που όμως υποβοηθείται από μαθηματικά εργαλεία που επιτρέπουν τη βελτιστοποίησή της.

Τυπικά ερωτήματα που αντιμετωπίζονται στη φάση της σχεδίασης είναι:

1. *Υπάρχει* αλγόριθμος που επιλύει το δοσμένο πρόβλημα;
2. Ο αλγόριθμος που σχεδιάστηκε λύνει ορθά το δοσμένο πρόβλημα; Ως αντιπαράδειγμα, ο αλγόριθμος (2) δεν είναι ορθός στην περίπτωση που το πρόβλημα εύρεσης του μεγίστου επιθυμεί να καλύψει τους αρνητικούς αριθμούς.
3. Ποιό το υπολογιστικό κόστος ενός αλγορίθμου; Μπορούμε να το (προ)υπολογίσουμε πρίν εκτελέσουμε τον αλγόριθμο;

4. *Υπάρχει πρακτικά χρήσιμος* αλγόριθμος που επιλύει το δοσμένο πρόβλημα; Υπάρχει δηλαδή αλγόριθμος του οποίου το υπολογιστικό κόστος να μήν αυξάνει δραματικά όταν μεγαλώνει το μέγεθος του προβλήματος;

5. Μεταξύ δύο ορθών αλγορίθμων που σχεδιάστηκαν για να λύνουν το δοσμένο πρόβλημα ποιός έχει το μικρότερο υπολογιστικό κόστος;

Όσον αφορά στο ερώτημα (1) η απάντηση δεν είναι πάντα καταφατική. Μπορείτε να προβληματιστείτε σχετικά αναζητώντας κάποιες αιτίες για τη μη ύπαρξη αλγοριθμικής επιλυσιμότητας. Στη συνέχεια θα διερευνήσουμε την απάντηση στα ερωτήματα (2)-(5).

## 1.3 Ορθότητα

Ένας αλγόριθμος είναι ορθός όταν ως αποτέλεσμα της εκτέλεσής του παρέχει την ορθή απάντηση σε ένα δοσμένο πρόβλημα ανεξάρτητα των συγκεκριμένων τιμών των παραμέτρων του προβλήματος.

Υπάρχουν δύο κυρίως τρόποι για τον έλεγχο της ορθότητας:

1. η απόδειξη με μαθηματικά εργαλεία (συνήθως με τη μαθηματική επαγωγή) και

2. ο έλεγχος της απάντησης σε ένα στατιστικά σημαντικό πλήθος εναλλακτικών παραμέτρων του προβλήματος.

Σε κάθε περίπτωση η μη ορθότητα αποδεικνύεται με την ανεύρεση έστω και ενός αντιπαραδείγματος.

**Παράδειγμα 1.1.** *Απόδειξη της ορθότητας με μαθηματική επαγωγή:*

*Έστω ότι ελέγχεται η ορθότητα του αλγορίθμου* (1).

*Για $n = 1$ ο αλγόριθμος είναι ορθός γιατί η τιμή $max\_val = A(1)$ είναι προφανώς η μέγιστη.*

*Έστω ότι ο αλγόριθμος είναι ορθός για $n = k$. Αρκεί να δείξουμε ότι είναι ορθός και για $n = k+1$. (Αν το πετύχουμε τότε σύμφωνα με την μέθοδο της μαθηματικής επαγωγής συνάγουμε ότι είναι ορθός για κάθε φυσικό $n \geqslant 1$). Η εφαρμογή του αλγορίθμου σε έναν πίνακα $A$ διαστάσεων $(k+1) \times 1$ μετά από εκτέλεση του for loop για $k-1$ φορές ισοδυναμεί με την ολοκλήρωση του αλγορίθμου οσάν αυτός να είχε εφαρμοστεί σε πίνακα $B$ διαστάσεων $k \times 1$ με στοιχεία ταυτόσημα προς τα πρώτα $k$ στοιχεία του $A$. Συνεπώς μέχρι αυτό το βήμα η μεταβλητή $max\_val$ περιέχει τη μέγιστη τιμή του $B$. Είμαστε σίγουροι γιαυτό δεδομένου ότι υποθέσαμε ότι ο αλγόριθμος είναι ορθός για $n = k$. Ας ονομάσουμε αυτή την τιμή $MAX\_VAL^k$. Είναι επομένως $MAX\_VAL^k \geqslant B(i) = A(i), \forall\ i = 1, \ldots, k$. Στην τελευταία εκτέλεση του for loop στην μεταβλητή $max\_val$ ανατίθεται η τιμή $max(MAX\_VAL^k, A(k+1))$ δηλαδή το μέγιστο μεταξύ της τρέχουσας τιμής της και του τελευταίου στοιχείου του πίνακα $A$ - του μόνου*

*δηλαδή στοιχείου που περιλαμβάνεται στον A αλλά όχι στο B. Ως αποτέλεσμα η τελική τιμή της max_val είναι* $MAX\_VAL^{k+1} \geqslant A(i), \forall\ i = 1, \ldots, k+1$, *δηλαδή η επιθυμητή.*

## 1.4 Υπολογιστική Πολυπλοκότητα

Ως υπολογιστική πολυπλοκότητα (computational complexity) ενός αλγορίθμου ορίζεται ο αριθμός των στοιχειωδών υπολογισμών που πραγματοποιούνται κατά την εκτέλεση του αλγορίθμου για την επίλυση του αντίστοιχου προβλήματος. Ως στοιχειώδεις υπολογισμοί μπορούν να θεωρηθούν οι μεμονωμένες πράξεις (π.χ., οι ακέραιες ή πραγματικές προσθέσεις, οι πολλαπλασιασμοί, κ.λπ.), οι αριθμητικές συγκρίσεις ή γενικότερα οποιοδήποτε βήμα του αλγορίθμου έχει σταθερό και μετρήσιμο υπολογιστικό κόστος.

Η υπολογιστική πολυπλοκότητα εξαρτάται από το μέγεθος $n$ του προβλήματος, συνεπώς είναι εύλογο να την θεωρούμε συνάρτηση της μορφής $f : \mathbb{N} \to \mathbb{N}$ που για κάθε συγκεκριμένο μέγεθος $n$ ενός προβλήματος μας επιστρέφει το πλήθος $f(n)$ των στοιχειωδών υπολογισμών που απαιτούνται για την εκτέλεση του αλγορίθμου.

**Παράδειγμα 1.2.** *Η υπολογιστική πολυπλοκότητα του αλγορίθμου* (1) *είναι* $f(n) = n - 1$ *αν σαν στοιχειώδη υπολογισμό θεωρήσουμε τη σύγκριση μεταξύ της μεταβλητής max_val και ενός στοιχείου του πίνακα A. Εδώ το μέγεθος του προβλήματος ταυτίζεται με το πλήθος n των στοιχείων του πίνακα A.*

Αξίζει να επισημάνουμε ότι για ένα συγκεκριμένο αλγόριθμο η υπολογιστική πολυπλοκότητα εξαρτάται από την επιλογή του τύπου των στοιχειωδών υπολογισμών που επιθυμούμε να μετρήσουμε. Έτσι στο παραπάνω παράδειγμα αν αντί των αριθμητικών "συγκρίσεων" μετρούσαμε τις "αναθέσεις" της μορφής $max\_val := A(i)$, η πολυπλοκότητα του αλγορίθμου μπορεί να ήταν διαφορετική. Συνεπώς όταν αναφερόμαστε σε αριθμητική πολυπλοκότητα πρέπει να προσδιορίζουμε και την "μονάδα μέτρησής" της.

Συχνά είναι δύσκολο ή και αδύνατο να προσδιορίσουμε την υπολογιστική πολυπλοκότητα επειδή ο αριθμός των απαιτούμενων στοιχειωδών υπολογισμών εξαρτάται από τα ίδια τα δεδομένα και όχι μόνο από το μέγεθος, $n$, του προβλήματος. Σ' αυτή την περίπτωση καταφεύγουμε συνήθως στον προσδιορισμό της "πολυπλοκότητας στη χειρότερη περίπτωση" (worst case computational complexity). Ως "πολυπλοκότητα στη χειρότερη περίπτωση" εννοούμε το μέγιστο αριθμό στοιχειωδών υπολογισμών που είναι δυνατό να απαιτηθούν για την ολοκλήρωση του αλγορίθμου.[2]

[2] Εναλλακτικά μπορεί να χρησιμοποιηθεί η "μέση πολυπλοκότητα" που ορίζεται ως η αναμενόμενη τιμή (μέση τιμή) του αριθμού των στοιχειωδών υπολογισμών όταν ο αλγόριθμος εκτελεστεί σε ένα στατιστικά αντιπροσωπευτικό δείγμα εμφανίσεων του προβλήματος.

**Παράδειγμα 1.3.** *Η υπολογιστική πολυπλοκότητα του αλγορίθμου* (1) *ως προς τις "αναθέσεις" της μορφής* $max\_val := A(i)$ *δεν είναι δυνατόν να προσδιοριστεί επακριβώς. Αν για παράδειγμα τα στοιχεία του πίνακα* $A$ *τύχει να είναι τοποθετημένα κατά φθίνουσα σειρά, τότε, ανεξαρτήτως του* $n$ *η πολυπλοκότητα είναι ίση με* $1$ *καθώς μόνο μία ανάθεση (η* $max\_val := A(1)$*) πρόκειται να εκτελεστεί. Αντίθετα αν τύχει τα στοιχεία του* $A$ *να είναι τοποθετημένα κατά αύξουσα σειρά και μάλιστα είναι ανά δύο διαφορετικά μεταξύ τους, θα απαιτηθούν συνολικά* $f_{\text{wc}}(n) = n$ *αναθέσεις. Αυτή είναι και η "πολυπλοκότητα στη χειρότερη περίπτωση".*

**Παράδειγμα 1.4.** *Ταξινόμηση* $n$ *αριθμών με τη μέθοδο της φυσαλίδας*

**Αλγόριθμος 3.** *Bubble sort: Έστω* $n$ *αριθμοί τοποθετημένοι σε πίνακα* $A$ *κατά τυχαία σειρά. Ο παρακάτω αλγοριθμος τους ταξινομεί κατά αύξουσα (για την ακρίβεια μή φθίνουσα) σειρά:*

```
function bubblesort(A,n)
for j=n-1:-1:1
    for i=1:j
       if (A(i)>A(i+1))
       then            %interchange A(i)<-->A(i+1)
           tmp:=A(i)
           A(i):=A(i+1)
           A(i+1):=tmp
       endif
   end
end
return A
```

*Αν ως μονάδα πολυπλοκότητας του αλγορίθμου* bubblesort *χρησιμοποιηθεί η "ανάθεση" σε μεταβλητές των στοιχείων του πίνακα* $A$ *ή μεταβλητών σε στοιχεία του ίδιου πίνακα, τότε η εκτέλεση των τριών εντολών μετά το* then *έχει κόστος* $3$*. Στη χειρότερη περίπτωση* [3] *αυτές οι τρεις εντολές εκτελούνται για κάθε* $i$*, δηλαδή* $j$ *φορές. Άρα το υπολογιστικό κόστος για κάθε φορά που εκτελείται το εξωτερικό* for loop *είναι* $3j$ *και συνεπώς το συνολικό κόστος είναι:*

$$f(n) = \sum_{j=1}^{n-1} 3j = \frac{3}{2}n(n-1) \tag{1.1}$$

[3] για τον αλγόριθμο bubblesort η χειρότερη περίπτωση είναι όταν ο πίνακας $A$ είναι αρχικά διατεταγμένος σε αύξουσα σειρά

**Παράδειγμα 1.5.** *Συγχώνευση δύο διατεταγμένων πινάκων A, B. Έστω πίνακες A, B μήκους $m$ και $n$ αντίστοιχα με στοιχεία διατεταγμένα κατά μή φθίνουσα σειρά. Ζητείται πίνακας C μήκους $m+n$ που περιέχει τα στοιχεία των δύο αρχικών επίσης διατεταγμένα κατά μη φθίνουσα σειρά.*

**Αλγόριθμος 4.** *Έστω πίνακες A, B μήκους $m$ και $n$ αντίστοιχα με στοιχεία διατεταγμένα κατά μή φθίνουσα σειρά. Ο παρακάτω αλγόριθμος παράγει πίνακα C μήκους $m+n$ που περιέχει τα στοιχεία των δύο αρχικών επίσης διατεταγμένα κατά μη φθίνουσα σειρά.*

```
function merge(A,B,m,n)
A(m+1):=INF
B(n+1):=INF
i:=1
j:=1
for k=1:m+n
   if (A(i) <= B(j))
   then
       C(k):=A(i)
       i:=i+1
   else
       C(k):=B(j)
       j:=j+1
   endif
end
return C
```

*Οι σταθερά INF στην αρχή του αλγορίθμου αντιπροσωπεύει μια πολύ μεγάλη τιμή (μεγαλύτερη από κάθε στοιχείο των A, B).*

*Είναι εύκολο να ελεγχθεί η ορθότητα του αλγορίθμου.*

*Όσον αφορά στην υπολογιστική πολυπλοκότητα: Αν ως μονάδα πολυπλοκότητας του αλγορίθμου merge χρησιμοποιηθεί η "ανάθεση" στοιχείων των πινάκων A ή B σε θέσεις του πίνακα C, τότε κάθε εκτέλεση του for loop έχει κόστος $1$ και άρα το συνολικό κόστος του αλγορίθμου είναι $f(m,n) = m+n$.*

*Στην περίπτωση αυτή το μέγεθος του προβλήματος εξαρτάται και από τις δύο διαστάσεις $m$, $n$. Έτσι εδώ η $f$ είναι συνάρτηση της μορφής $f : \mathbb{N}^2 \to \mathbb{N}$*

### 1.4.1 Τάξη μεγέθους

Εξετάζοντας την υπολογιστική πολυπλοκότητα $f(n)$ ενός αλγορίθμου είναι συχνά χρησιμότερο να προσδιορίσουμε την *τάξη μεγέθους* της παρά αυτή καθ' αυτή την ακριβή της έκφραση.

Εκφραζόμενοι κάπως ανεπίσημα, μία συνάρτηση $f(n)$ ανήκει στην τάξη μεγέθους $\mathcal{O}(g)$ της συνάρτησης $g(n)$, αν αυξάνεται το πολύ όσο γρήγορα αυξάνεται η $g(n)$.

Αυστηρότερα η τάξη μεγέθους $\mathcal{O}(g)$ μιας συνάρτησης $g : \mathbb{N} \to \mathbb{N}$ ορίζεται ως το σύνολο των συναρτήσεων που τελικά (δηλαδή για $n$ μεγαλύτερο κάποιου συγκεκριμένου $n_0$) φράσονται από τα πάνω από ένα πολλαπλάσιο $cg(n)$ της $g(n)$, δηλαδή:

$$\mathcal{O}(g) = \{f / f : \mathbb{N} \to \mathbb{N} \text{ και } \exists \text{ σταθερές } c,\ n_0 :\ f(n) \leqslant c\, g(n)\ \forall n > n_0\} \tag{1.2}$$

Ένας ισοδύναμος εναλλακτικός ορισμός είναι ο ακόλουθος:

$$\mathcal{O}(g) = \{f / f : \mathbb{N} \to \mathbb{N} \text{ και } \exists \text{ σταθερές } c, d :\ f(n) \leqslant c\, g(n)\ +\ d, \forall n \in N\} \tag{1.3}$$

όπου η σταθερά $d \geqslant \max_{n=1,\ldots,n_0}\{f(n) - c\, g(n)\}$

Από τους παραπάνω ορισμούς αν $f \in \mathcal{O}(g)$ τότε η $f$ ανήκει και στην τάξη μεγέθους οποιασδήποτε συνάρτησης $h$ αυξάνεται γρηγορότερα από την $g$. Στην πράξη πάντως έχει ενδιαφέρον να βρούμε τη "λιγότερο αύξουσα" συνάρτηση $g$ στης οποίας την τάξη μεγέθους ανήκει μία δεδομένη $f$.

**Παράδειγμα 1.6.** *Τάξη μεγέθους πολυωνυμικών συναρτήσεων:*

*Η συνάρτηση* $f(n) = 2\,n^2 + 5\,n + 4 \in \mathcal{O}(n^3)$. *Πράγματι, για* $n > 3$ *εύκολα διαπιστώνουμε ότι* $f(n) < g(n)$ *οπότε η συνθήκη ικανοποιείται για* $c = 1$ *και* $n_0 = 3$. *Πιό χρήσιμο είναι όμως το γεγονός ότι* $f \in \mathcal{O}(n^2)$. *Πράγματι, αν επιλέξουμε* $c = 2 + 5 + 4 = 11$ *και* $d = 4$ *μπορούμε εύκολα να επαληθεύσουμε ότι* $f(n) \leqslant 11n^2 + 4$.

*Η διαπίστωση αυτή γενικεύεται. Εύκολα μπορείτε να δείξετε ότι οποιαδήποτε πολυωνυμική συνάρτηση της μορφής* $f(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0 \in \mathcal{O}(n^k)$ *ικανοποιώντας τη συνθήκη της εξίσωσης* (1.3) *για* $c = \sum_{i=0\ldots k} a_i$ *και* $d = a_0$.

**Παράδειγμα 1.7.** *Τάξη μεγέθους του αλγορίθμου* (3) *(bubblesort): Το υπολογιστικό κόστος του αλγορίθμου αυτού βρέθηκε (βλ. παραπάνω) να είναι* $f(n) = \frac{3}{2}n(n-1) = \frac{3}{2}n^2 - \frac{3}{2}n \in \mathcal{O}(n^2)$.

### 1.4.2 Ρυθμός αύξησης

Ορίζουμε ότι δύο συναρτήσεις $f(n)$, $g(n)$ έχουν τον ίδιο *ρυθμό αύξησης* και γράφουμε $f \approx g$ όταν $f \in \mathcal{O}(g)$ και ταυτόχρονα $g \in \mathcal{O}(f)$. Η σχέση $\approx$ είναι σχέση ισοδυναμίας (δηλαδή ανακλαστική: $f \approx f$, συμμετρική: $f \approx g \Rightarrow g \approx f$ και μεταβατική: $f \approx g$ και $g \approx h \Rightarrow f \approx h$).

Αν από την άλλη μεριά εάν $f \in \mathcal{O}(g)$ αλλά $g \notin \mathcal{O}(f)$ τότε ο ρυθμός αύξησης της $f(n)$ είναι μικρότερος από το ρυθμό αύξησης της $g(n)$.

### 1.4.3 Μία χρήσιμη ιεραρχία της υπολογιστικής πολυπλοκότητας αλγορίθμων

Εύκολα μπορεί να αποδειχθεί ότι:

$$\mathcal{O}(n^k) \subset \mathcal{O}(n^k \log(n)) \subset \mathcal{O}(n^{k+1}) \subset \mathcal{O}(2^n) \subset \mathcal{O}(n^n) \subset \mathcal{O}(n!)\,. \tag{1.4}$$

Συνεπώς αν για παράδειγμα ένας αλγόριθμος έχει υπολογιστική πολυπλοκότητα $f(n) \approx n\log(n)$ είναι προτιμότερος από έναν άλλο με πολυπλοκότητα $g(n) \approx n^2$ και αυτός με τη σειρά του είναι προτιμότερος από έναν τρίτο με πολυπλοκότητα $h(n) \approx 2^n$.

### 1.4.4 Σύγκριση της υπολογιστικής πολυπλοκότητας αλγορίθμων

Ας υποθέσουμε ότι δύο εναλλακτικοί αλγόριθμοι (που επιλύουν το ίδιο πρόβλημα) έχουν υπολογιστικές πολυπλοκότητες $f(n)$, $g(n)$ όπου $n$ το μέγεθος του προβλήματος. Τότε ισχύουν οι παρακάτω ποιοτικές παρατηρήσεις:

1. Αν ο ρυθμός αύξησης της $f(n)$ είναι μικρότερος από το ρυθμό αύξησης της $g(n)$ τότε με κριτήριο την υπολογιστική πολυπλοκότητα επιλέγεται ο δεύτερος έναντι του πρώτου.

2. Αν $f \approx g$ τότε η υπολογιστική τους πολυπλοκότητα είναι παρεμφερής και η τελική μεταξύ τους επιλογή κρίνεται στις λεπτομέρειες της προγραμματιστικής τους υλοποίησης.

3. Οι αλγόριθμοι με υπολογιστική πολυπλοκότητα της μορφής $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \in \mathcal{O}(n^k)$ ονομάζονται πολυωνυμικοί αλγόριθμοι βαθμού $k$. Μεταξύ δύο πολυωνυμικών αλγορίθμων προτιμότερος είναι αυτός με το μικρότερο βαθμό.

### 1.4.5 Η κλάση προβλημάτων $\mathcal{P}$

Αν ένα πρόβλημα αποσκοπεί στη λήψη μίας διαδικής απόφασης (ναι/όχι) τότε λέγεται πρόβλημα απόφασης. Από τα προβλήματα απόφασης όσα επιδέχονται λύση από πολυωνυμικούς αλγόριθμους αποτελούν την κλάση (οικογένεια) $\mathcal{P}$.

### 1.4.6 Η κλάση προβλημάτων $\mathcal{NP}$

Έστω ένα πρόβλημα απόφασης μεγέθους $n$ της μορφής: "Τα δεδομένα εισόδου $\mathbb{A} = (A(1), \ldots, A(n))$ ικανοποιούν τη συνθήκη $\mathbb{C}$;".

Για παράδειγμα, θεωρήστε το εξής πρόβλημα: Δοθέντος ενός φυσικού αριθμού $m$ με διαδιακή αναπαράσταση $a = a_n a_{n-1} \ldots a_1,\ a_i \in \{0, 1\}$ ελέγξτε αν αυτός είναι σύνθετος (έχει ακέραιους διαιρέτες).

Έστω τώρα ότι η εν λόγω συνθήκη ικανοποιείται αν βρεθεί έστω και ένας μάρτυρας αληθείας της. Για το προηγούμενο παράδειγμα αρκεί να βρεθεί έστω και ένας ακέραιος που να διαιρεί τον $a$.

Υποθέστε τώρα ότι ένας από μηχανής θεός προτείνει έναν μάρτυρα $\mathbb{B}$ μεγέθους $m \leqslant n^c$ (όπου $c \in \mathbb{N}$ μία σταθερά) προκειμένου να μας απαλλάξει από τη διαδικασία ανεύρεσης που κανονικά αποτελεί μέρος της λύσης του προβλήματος. Για το προηγούμενο παράδειγμα έστω ότι μας προτείνεται ο ακέραιος $b = b_n b_{n-1} \dots b_1,\ b_i \in \{0, 1\}$ μεγέθους $m = n^1$ ($c = 1$).

Πλέον η επίλυση του αρχικού προβλήματος περιορίζεται στον έλεγχο του κατά πόσο ο μάρτυρας αυτός είναι όντως αξιόπιιστος. Ο εν λόγω έλεγχος αποτελεί ένα νέο πρόβλημα με είσοδο τα δεδομένα του αρχικού προβλήματος και τον προτεινόμενο μάρτυρα. Για το παράδειγμά μας το νέο πρόβλημα έχει τη μορφή: Ελέγξτε αν ο $b$ διαιρεί τον $a$.

Αν το (νέο) πρόβλημα ελέγχου του μάρτυρα ανήκει στην κλάση $\mathcal{P}$, τότε ορίζουμε πως το αρχικό πρόβλημα ανήκει στην κλάση $\mathcal{NP}$.

Είναι εύκολο να ελεγχθεί ότι

$$\mathcal{P} \subseteq \mathcal{NP}\,. \tag{1.5}$$

Παραμένει ωστόσο αναπόδεικτη η εικασία ότι

$$\mathcal{P} \subset \mathcal{NP}\,. \tag{1.6}$$

Η τελευταία παρατήρηση υποδηλώνει την εξής άγνοιά μας: Γνωρίζουμε μία σειρά προβλημάτων της κλάσης $\mathcal{NP}$ αλλά δεν μπορούμε με βεβαιότητα να αποκλείσουμε ότι η εξεύρεση κάποιων καλύτερων αλγορίθμων επίλυσής τους θα τα κατέτασε στην κλάση $\mathcal{P}$.

### 1.4.7 Η κλάση των $\mathcal{NP}$-complete προβλημάτων

Ως συνέπεια της προαναφερθείσας άγνοιάς μας ορίζεται μία κλάση $\mathcal{NPC}$ δύσκολων προβλημάτων με την ονομασία $\mathcal{NP}$-complete προβλήματα. Τα προβλήματα της κλάσης αυτής έχουν δύο ιδιότητες:

1. Ανήκουν στην κλάση $\mathcal{NP}$
2. Αν έστω και για ένα από αυτά αποδειχθεί (στο μέλλον) ότι ανήκει στην κλάση $\mathcal{P}$ τότε αυτομάτως θα έχει αποδειχθεί ότι $\mathcal{NPC} \subset \mathcal{P}$, δηλαδή όλα τα $\mathcal{NP}$-complete προβλήματα θα ήταν πολυωνυμικά επιλύσιμα.

### 1.4.8 Σχόλια και παρατηρήσεις

Από την παραπάνω συζήτηση πρέπει να έχει γίνει σαφές ότι από τα $\mathcal{NP}$ προβλήματα, τα $\mathcal{P}$ είναι τα εύκολα και τα $\mathcal{NP}$-complete τα δύσκολα ώς προς την υπολογιστική τους πολυπλοκότητα.

Από την άλλη μεριά μπορεί να αποδειχθεί ότι όλα τα $\mathcal{NP}$ προβλήματα, περιλαμβανομένων και των $\mathcal{NP}$-complete επιδέχονται αλγόριθμους με πολυπλοκότητα $f(n) \in \mathcal{O}(2^n)$. Είναι δηλαδή λιγότερο ή το πολύ ισοδύναμοι με αλγόριθμους εκθετικής πολυπλοκότητας.

# Κεφάλαιο 2

# Γενικευμένοι αλγόριθμοι

Αν και κάθε συγκεκριμένο πρόβλημα επιλύεται με τη χρήση εξειδικευμένων αλγορίθμων, υπάρχουν ορισμένες οικογένειες πρότυπων αλγορίθμων με κοινά χαρακτηριστικά όσον αφορά στη μεθοδολογία τους και κατ' επέκταση την υπολογιστική τους πολυπλοκότητα. Στη συνέχεια παρουσιάζονται τρεις βασικές τέτοιες οικογένειες καθώς και αντίστοιχα παραδείγματα χρήσης τους για την επίλυση συγκεκριμένων γνωστών προβλημάτων.

## 2.1 Διαίρει και βασίλευε

### 2.1.1 Ορισμός

Η αλγόριθμοι της οικογένειας "διαίρει και βασίλευε" ανάγουν τη λύση ενός προβλήματος μεγέθους $n$ στην επίλυση δύο υποπροβλημάτων του ίδιου τύπου μεγέθους $\sim n/2$ (για την ακρίβια $\lfloor n/2 \rfloor$ και $\lceil n/2 \rceil$). Η συνολική λύση προκύπτει από το συνδυασμό των δύο επιμέρους λύσεων. Αυτή η τακτική του "διαίρει" εφαρμόζεται αναδρομικά $\log_2(n)$ φορές μέχρι την κατάτμηση του αρχικού προβλήματος σε στοιχειώδη υποπροβλήματα μεγέθους $1$.

### 2.1.2 Εύρεση μεγίστου με διαίρει και βασίλευε

**Αλγόριθμος 5.** *Εύρεση του μεγίστου $n = 2^k$ αριθμών Ο ακόλουθος αναδρομικός αλγόριθμος υλοποιεί στο συγκεκριμένο πρόβλημα τη μέθοδο διαίρει και βασίλευε.*

```
function maxdc[i,j]
if (i-j)<=1
then return max[A(i),A(j)]
else
   medij=floor[(i+j)/2]
```

```
    max1:=maxdc[i,medij]
    max2:=maxdc[medij+1,j]
    return max[max1,max2]
endif
max_val:=maxdc[1,n]
```

*Όσο οι δείκτες $i, j$ διαφέρουν ο πίνακας $A$ υποδιαιρείται αναδρομικά. Για $n = 2^k$ η διαδικασία αυτή θα επαναληφθεί $k - 1$ φορές έως ότου ο $A$ να έχει χωριστεί σε $n/2$ ζεύγη. Τότε η αναδρομική διαδικασία θα αρχίσει να επιστρέφει. Οι πρώτες τιμές που θα επιστραφούν θα είναι τα μέγιστα από κάθε διάδα. Στη συνέχεια τα μέγιστα ανάμεσα στα αποτελέσματα των $1ης$-$2ης$, $3ης$-$4ης$, κ.λπ. διάδων, και ούτω καθεξής μέχρι τελικά να υπολογιστεί το συνολικό μέγιστο.*

*Η υπολογιστική πολυπλοκότητα, $f(n)$, του αλγορίθμου (ως προς του αριθμό συσγκρίσεων κατά τον υπολογισμό της συνάρτησης $\max[x, y]$) υπολογίζεται ως εξής:*

$$f(n) = \begin{cases} 1 & n = 2 \\ 2f(n/2) + 1 & n > 2 \end{cases} \tag{2.1}$$

*Άρα για $n = 2^k$*

$$\begin{aligned} f(n) &= 2f(n/2) + 1 && (2.2)\\ 2f(n/2) &= 4f(n/4) + 2 && (2.3)\\ 4f(n/4) &= 8f(n/8) + 4 && (2.4)\\ &\cdots && (2.5)\\ 2^{k-1}f(2) &= 2^{k-1} && (2.6) \end{aligned}$$

*Οπότε προσθέτοντας κατά μέλη $f(n) = 1 + 2 + 4 + 2^{k-1} = 2^k - 1 = n - 1$*

Παρατηρούμε ότι η υπολογιστική πολυπλοκότητα του αλγορίθμου (5) ισούται με αυτήν του αλγορίθμου (1). Ωστόσο σε άλλες περιπτώσεις η χρήση της μεθοδολογίας διαίρει και βασίλευε επιφέρει σημαντική βελτίωση.

### 2.1.3 Ταξινόμηση με διαίρει και βασίλευε

**Αλγόριθμος 6.** *Ο αλγόριθμος αυτός ταξινομεί $n$ αριθμούς που περιέχονται με αρχικά τυχαία σειρά στον πίνακα $A$. Έχουμε ήδη δεί ότι το ίδιο πρόβλημα λύνει ο αλγόριθμος* (3) *(bubblesort) με πολυπλοκότητα $\mathcal{O}(n^2)$.*

```
function sortdc[A,p,q]
if (p<q) then
```

```
    r:=floor[(p+q)/2 ]
    sortdc[A,p,r]
    sortdc[A,r+1,q]
    A(p:q):=merge[A(p:r),A(r+1:q),r-p,q-r-1]
 endif
```

*Το επιθυμητό αποτέλεσμα παράγεται με την κλήση του αλγορίθμου με ορίσματα:*

```
sortdc[A,1,n]
```

*Έλεγχος ορθότητας: Για απλούστευση θα εξετάσουμε τη συμπεριφορά του αλγορίθμου για* $n = 2^k$*. Όσο οι δείκτες* $p$*,* $q$ *διαφέρουν ο πίνακας* $A$ *υποδιαιρείται αναδρομικά. Για* $n = 2^k$ *η διαδικασία αυτή θα επαναληφθεί* $k$ *φορές έως ότου ο* $A$ *να έχει χωριστεί σε* $n$ *στοιχειώδεις πίνακες μήκους* $1$*. Τότε η αναδρομική διαδικασία θα αρχίσει να επιστρέφει (γιατί δεν θα ικανοποιείται η συνθήκη* $p < q$*). Η κλήση του αλγορίθμου merge (βλ. αλγόριθμο* (4)*) θα δημιουργήσει αρχικά ταξινομημένες διάδες, μετά ταξινομημένες τετράδες, οκτάδες κ.λπ. μέχρι να ταξινομηθεί τελικά ολόκληρος ο πίνακας* $A$*.*

*Έλεγχος πολυπλοκότητας: Η υπολογιστική πολυπλοκότητα,* $f(n)$*, του αλγορίθμου (ως προς του αριθμό αναθέσεων υπολογίζεται αναδρομικά ως εξής:*

$$f(n) = \begin{cases} 2 & n = 2 \\ 2f(n/2) + 2\,n/2 & n > 2 \end{cases} \tag{2.7}$$

*οπου το κόστος* $2$ *μονάδων για* $n = 2$ *και* $2, n/2$ *μονάδων για* $n > 2$ *προκύπτει από την εκτέλεση του αλγορίθμου merge*$[,]$ *για δύο πίνακες διαστάσεων* $1,\ 1$ *στην πρώτη περίπτωση και* $n/2,\ n/2$ *στη δεύτερη περίπτωση. Με απλή αντικατάσταση μπορούμε να διαπιστώσουμε ότι λύση της εξίσωσης* (2.7) *είναι η* $f(n) = n\,log(n) \in \mathcal{O}(nlog(n))$*.*

Όντως λοιπόν σ' αυτή την περίπτωση η μέθοδος διαίρει και βασίλευε βελτίωσε την υπολογιστική πολυπλοκότητα σε σχέση με τον αλγόριθμο bubblesort που είχε πολυπλοκότητα $\mathcal{O}(n^2)$.

## 2.2 Δυναμικός προγραμματισμός

### 2.2.1 Ορισμός

Ο δυναμικός προγραμματισμός είναι μια αλγοριθμική μεθοδολογία που μοιάζει αρκετά με το διαίρει και βασίλευε αλλά επιπρόσθετα χρησιμοποιεί ένα πίνακα καταχώρησης των ενδιάμεσων αποτελεσμάτων για εκείνα τα υποπροβλήματα των οποίων η επίλυση απαιτείται πολλές φορές. Έτσι αποφεύγεται η δαπάνη της επίλυσης ξανά-και-ξανά του ίδιου προβλήματος.

### 2.2.2 Το πρόβλημα "έχω ρέστα;..."

**Παράδειγμα 2.1.** *Ο ταμίας σε ένα κατάστημα πρέπει να δώσει ρέστα $r$ λεπτά του Ευρώ ($r \in N$). Στο ταμείο έχει διαθέσιμα κέρματα συνολικού πλήθους $n$ με αντίστοιχες αξίες $v_1, v_2, \ldots, v_n$ λεπτών.*

*Καλείται να απαντήσει στο ερώτημα αν ένας συνδυασμός κερμάτων, απ' αυτά που έχει, δίνει άθροισμα $r$ (όχι ποιά είναι αυτά τα κέρματα).*

**Αλγόριθμος 7.** *Ο αλγόριθμος κατασκευάζει τον πίνακα $S$ διαστάσεων $(n+1) \times (r+1)$ με τιμές TRUE/FALSE έτσι ώστε $S(i,j) = TRUE$ όταν ένα υποσύνολο των πρώτων $i$ κερμάτων έχει άθροισμα ακριβώς $j$*

*Με την ολοκλήρωση του αλγορίθμου αρκεί να ελεγχθεί η τιμή $S(n,r)$. Αν $S(n,r) = TRUE$ τότε η απάντηση στο ερώτημα είναι καταφατική.*

```
 function changedp[V, r]
% Initialization
 S(0,0):=TRUE
 for j=1:r
    S(0,j):=FALSE
 end
% Main part
 for i=1:n
    for j=0:r
        S(i,j):=S(i-1,j)
        if j-V(i) >= 0 then
            S(i,j):= ( S(i,j) OR S(i-1,j-V(i)) )
        endif
    end
 end
 return S(n,r)
```

*Έλεγχος ορθότητας: Για να είναι $S(i,j) = TRUE$ θα πρέπει στα πρώτα $i$ κέρματα να υπάρχει ένα (οποιοδήποτε) υποσύνολο με άθροισμα ακριβώς $j$. Τούτο μπορεί να αποφασιστεί με βάση τα πρώτα $i-1$ κέρματα ως έξης:*

*(α) Αν στα πρώτα $i-1$ κέρματα υπήρχε συνδυασμός με άθροισμα $j$ τότε σίγουρα θα υπάρχει και στα πρώτα $i$.*

*(β) Αν αντίθετα στα πρώτα $i-1$ κέρματα δεν υπήρχε τέτοιος συνδυασμός (δηλ. $S(i-1,j) = FALSE$) τότε η μόνη ελπίδα είναι η προσθήκη του κέρματος $i$ (αξίας $V(i)$) να τον δημιούργησε. Για να ισχύει*

*όμως κάτι τέτοιο θα πρέπει (β.1) Η αξία $V(i)$ του κέρματος $i$ να μήν υπερβαίνει το $j$. (β.2) Οπωσδήποτε στα πρώτα $i-1$ κέρματα να υπάρχει ένα υποσύνολο που δίνει άθροισμα $j-V(i)$ ή ισοδύναμα $S(i-1, j-V(i)) = TRUE$.*

*Ο παραπάνω αλγόριθμος κάνει ακριβώς αυτούς τους ελέγχους και συνεπώς είναι ορθός. Αξίζει να σημειωθεί ότι:*

1. *Οι έλεγχοι γίνονται με τη σωστή σειρά δηλαδή η τιμή που ανατίθεται στο $S(i,j)$ υπολογίζεται βάσει στοιχείων του πίνακα $S$ που έχουν νωρίτερα ενημερωθεί.*

2. *Η αρχικοποίηση της πρώτης γραμμής $S(0,j)$, $j = 0, \ldots, r$ του πίνακα είναι συμβατή με τον ορισμό του $S$ καθώς μεταξύ $0$ κερμάτων μόνο $0$ μπορεί να είναι το άθροισμα τιμών. Έτσι η αναδρομή που περιγράφηκε παραπάνω εφαρμόζεται χωρίς πρόοβλημα από τη γραμμή $S(1,j)$ κι έπειτα.*

*Έλεγχος Υπολογιστικής πολυπλοκότητας: Ως μονάδα υπολογιστικής πολυπλοκότητα θα χρησιμοποιήσουμε την ανάθεση τιμής σε στοιχείο του πίνακα $S$.*

*Η αρχικοποίηση έχει κόστος $\mathcal{O}(r)$. Κάθε εκτέλεση του εσωτερικού for loop έχει κόστος $1$ ή $2$ αναθέσεων ανάλογα με το πρόσημο της $j-V(i)$ δηλαδή ανήκει στην τάξη μεγέθους $\mathcal{O}(1)$.Η υλοποίσή του $r+1$ φορές λόγω του εσωτερικού for loop έχει κόστος $\mathcal{O}(r)$. Αυτό επαναλαμβάνεται $n$ φορές λόγω του εξωτερικού for loop και άρα το κυρίως μέρος του αλγορίθμου έχει πολυπλοκότητα στην τάξη μεγέθους $\mathcal{O}(nr)$. Συνεπώς συνολικά ο αλγόριθμος έχει πολυπλοκότητα $\mathcal{O}(nr)$*

## 2.3 Άπληστοι αλγόριθμοι (greedy algorithms)

### 2.3.1 Ορισμός

Κατά την επίλυση ενός προβλήματος βελτιστοποίησης [1], οι άπληστοι αλγόριθμοι επιλύουν διαδοχικά ένα πλήθος μικρότερων προβλημάτων με κριτήριο, σε κάθε βήμα, το άμεσο κέρδος.

Συχνά οι άπληστοι αλγόριθμοι δεν δίνουν τη βέλτιστη λύση, άλλες φορές όμως την προσεγγίζουν και άλλοτε την επιτυγχάνουν.

### 2.3.2 Βέλτιστη τοποθέτηση αρχείων σε μαγνητική ταινία

Έστω ότι ένα πλήθος, $n$, αρχείων με μέγεθος $f_i$, $i = 1, \ldots, n$ πρόκειται να εγγραφεί σε μαγνητική ταινία. Η εγγραφή γίνεται χωρίς κενά μεταξύ των αρχείων. Η ανάγνωση κάθε αρχείου γίνεται ως εξής: (α) η κεφαλή παίρνει θέση στην αρχή της ταινιάς, (β) για να ξεκινήσει το διάβασμα του αρχείου $J$ απαιτείται χρόνος ίσος (ή ανάλογος) προς το συνολικό μήκος των προηγούμενων $J-1$

[1] ελαχιστοποίησης μιας συνάρτησης κόστους

αρχείων. Με την υπόθεση ότι όλα τα αρχεία θα διαβαστούν ζητείται η βέλτιστη σειρά τοποθέτησης των $n$ αρχείων στην ταινία.

**Αλγόριθμος 8.** *Ο παρακάτω αλγόριθμος βιάζεται να τοποθετήσει σε κάθε βήμα το μικρότερο διαθέσιμο αρχείο.*

```
for J=1:n
   choose the shortest of the remaining files
   put it on the tape
end
```