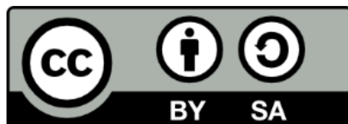
ΕΙΣΑΓΩΓΗ ΣΤΗΝ ΑΝΑΛΥΣΗ ΑΛΓΟΡΙΘΜΩΝ

Ενότητα 1: Εισαγωγή

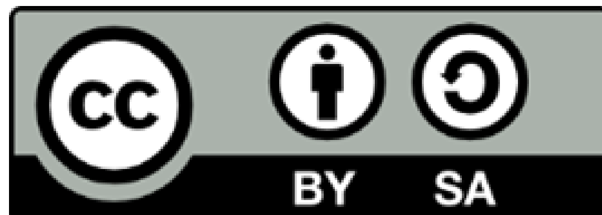
Μαρία Σατρατζέμη
Τμήμα Εφαρμοσμένης Πληροφορικής



Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.



Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Μακεδονίας» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

0. Εισαγωγή

- **Αντικείμενο μαθήματος:** Η θεωρητική μελέτη ανάλυσης των αλγορίθμων.
- **Στόχος:** επιδόσεις των επαναληπτικών και αναδρομικών αλγορίθμων.
- ***Ανάλυση:*** προβλέπει το κόστος ενός αλγορίθμου σε σχέση με τους πόρους και την εκτέλεση/επίδοση.
- (***Σχεδίαση:*** ανάπτυξη τεχνικών με στόχο την ελάττωση του κόστους)

0.1 Αλγόριθμος

- **Αλγόριθμος** = καλά ορισμένη υπολογιστική διαδικασία που δέχεται κάποια τιμή ή κάποιο σύνολο τιμών ως **είσοδο** και δίνει κάποια τιμή ή κάποιο σύνολο τιμών ως **έξοδο**.
- ➢ μια ακολουθία υπολογιστικών βημάτων που μετασχηματίζει την είσοδο σε έξοδο.
- **Υπολογιστικό πρόβλημα**: Ταξινόμηση
- <u>Είσοδος</u>: μια ακολουθία n αριθμών $<a_1,\ldots,a_n>$
- <u>Έξοδος</u>: μια αναδιάταξη $<a'_1,\ldots,a'_n>$ της ακολουθίας εισόδου τέτοια, ώστε $a'_1\leq\ldots\leq a'_n$.
- **Στιγμιότυπο προβλήματος** = η είσοδος η οποία απαιτείται για να υπολογιστεί μια λύση του προβλήματος και που ικανοποιεί όλους τους περιορισμούς που επιβάλλει η διατύπωση του προβλήματος.
- Για παράδειγμα, ένα στιγμιότυπο του προβλήματος ταξινόμησης είναι η ακολουθία <10,4,5,2,15,7> και ένας αλγόριθμος ταξινόμησης επιστρέφει ως έξοδο την ακολουθία <2,4,5,7,10,15>.

- Έχουν αναπτυχθεί διάφοροι αλγόριθμοι ταξινόμησης. Ποιος είναι ο πιο κατάλληλος; Είναι όλοι τους ορθοί; Υπάρχει κάποιος καλύτερος;
  - Το ποιος είναι ο πιο κατάλληλος εξαρτάται από διάφορους παράγοντες $\rightarrow$ αντικείμενο του μαθήματος.
  - Ένας αλγόριθμος χαρακτηρίζεται **ορθός** αν για κάθε στιγμιότυπο εισόδου τερματίζει δίνοντας την ορθή έξοδο.
  - Αν υπάρχει κάποιος καλύτερος $\rightarrow$ αντικείμενο έρευνας.
- Επειδή ο υπολογιστικός χρόνος στους υπολογιστές είναι ένας πεπερασμένος πόρος, όπως και η μνήμη, αναζητούμε αλγορίθμους που είναι δραστικοί (αποδοτικοί) ως προς τον χρόνο και οικονομικοί ως προς τον χώρο.
- Συχνά για το ίδιο πρόβλημα έχουμε αλγορίθμους που διαφέρουν συντριπτικά ως προς την δραστικότητα, και οι διαφορές αυτές μπορεί να είναι πολύ πιο σημαντικές από εκείνες που οφείλονται στο υλικό του υπολογιστή και το λογισμικό.

- Για παράδειγμα, έχουμε δύο αλγορίθμους (μεταξύ άλλων) ταξινόμησης:

1. ταξινόμηση με ένθεση (insert sort)
2. ταξινόμηση με συγχώνευση (merge sort)

- Ο 1ος απαιτεί χρόνο $\simeq c_1 n^2$ για την ταξινόμηση n στοιχείων.
- Ο 2ος απαιτεί χρόνο $\simeq c_2 n \lg n$ για την ταξινόμηση n στοιχείων ($c_1 < c_2$)

➢ Από την ανάλυση προκύπτει ότι από κάποια τιμή του n και μετά η ταξινόμηση με συγχώνευση θα είναι ταχύτερη. Γενικά, όσο αυξάνεται το μέγεθος του προβλήματος, τόσο αυξάνεται και το συγκριτικό πλεονέκτημα του 2ου αλγορίθμου (βλ. CLRS σελ. 11-12(Ελλ.εκδ.) ή 12-13(Αγγ.εκδ.)

0.2 Ανάλυση Αλγορίθμων

- Ο όρος ανάλυση ενός αλγορίθμου αναφέρεται στην πρόβλεψη των πόρων που απαιτεί η εκτέλεση του αλγορίθμου. Τις περισσότερες φορές η ποσότητα που θέλουμε να μετρήσουμε είναι ο υπολογιστικός χρόνος. Γενικά αναλύοντας διάφορους υποψήφιους αλγορίθμους για ένα πρόβλημα, μπορούμε να εντοπίσουμε τον πλέον δραστικό.
- Προκειμένου να αναλύσουμε έναν αλγόριθμο θα θεωρήσουμε το μοντέλο της μηχανής άμεσης ή τυχαίας προσπέλασης (Random Access Machine - RAM) με έναν επεξεργαστή όπου οι εντολές εκτελούνται ακολουθιακά (η μία κατόπιν της άλλης), δηλ. δεν υπάρχουν πράξεις που να εκτελούνται ταυτόχρονα.
- Το μοντέλο RAM περιλαμβάνει εντολές οι οποίες υπάρχουν σε όλους τους πραγματικούς υπολογιστές:

- αριθμητικές πράξεις (πρόσθεση, αφαίρεση, πολλαπλασιασμός, διαίρεση, υπόλοιπο, ακέραιο μέρος)
- εντολές μεταφοράς δεδομένων (ανάκληση, αποθήκευση, αντιγραφή)
- εντολές ελέγχου (διακλάδωση υπό συνθήκη και άνευ συνθήκης, κλήση υποπρογράμματος και επιστροφή)
- Όλες αυτές οι εντολές απαιτούν κάποιον σταθερό χρόνο για την εκτέλεσή τους. Ο χρόνος εκτέλεσης εξαρτάται από τα δεδομένα εισόδου. Γενικά, ο απαιτούμενος χρόνος αυξάνεται καθώς αυξάνεται το μέγεθος της εισόδου και γι’ αυτό συνηθίζεται να εκφράζουμε τον χρόνο εκτέλεσης ενός προγράμματος ως συνάρτηση του μεγέθους της εισόδου. Είναι αναγκαίο να διευκρινίσουμε τους όρους ‘‘χρόνος εκτέλεσης’’ και ‘‘μέγεθος εισόδου’’.
- Το ποιο είναι το καταλληλότερο μέτρο για το **μέγεθος εισόδου** εξαρτάται από το πρόβλημα. Σε κάθε πρόβλημα που εξετάζουμε πρέπει να δηλώνουμε ρητά ποιο μέτρο χρησιμοποιούμε για το μέγεθος της εισόδου.

- Για πολλά προβλήματα, όπως π.χ. η ταξινόμηση, το μέτρο θα είναι το πλήθος των στοιχείων της εισόδου.
- Για άλλα προβλήματα, όπως π.χ. ο πολ/σιασμός δύο ακεραίων, το μέτρο θα είναι το συνολικό πλήθος bit που απαιτούνται για την αναπαράσταση της εισόδου στο δυαδικό σύστημα.
- Ενίοτε είναι ενδεδειγμένο να περιγράφουμε το μέγεθος της εισόδου με δύο αριθμούς, όπως για παράδειγμα, όταν η είσοδος είναι ένας γράφος (πλήθος κόμβων, πλήθος ακμών).
- Ο **χρόνος εκτέλεσης** ενός αλγορίθμου για δεδομένη είσοδο είναι το πλήθος των στοιχειωδών πράξεων ή ‘‘βημάτων’’ που εκτελούνται, όπου το ‘‘βήμα’’ πρέπει να ορίζεται έτσι ώστε να είναι κατά το δυνατόν πιο ανεξάρτητο από τον τύπο της μηχανής. Θα κάνουμε την παραδοχή (συμβατή με το μοντέλο RAM) ότι ο χρόνος εκτέλεσης για κάθε γραμμή του ψευδοκώδικά μας είναι σταθερός και αν και η μια γραμμή μπορεί να απαιτεί διαφορετικό χρόνο από την άλλη, θα υποθέσουμε ότι κάθε εκτέλεση της i-οστής γραμμής απαιτεί χρόνο c_i, όπου c_i = σταθ.

0.3 Περιπτώσεις ανάλυσης

Χειρότερης περίπτωσης: (συνήθως)

- $T(n)$ = μέγιστος χρόνος εκτέλεσης του αλγορίθμου για οποιαδήποτε είσοδο μεγέθους n.

Μέσης περίπτωσης: (μερικές φορές)

- $T(n)$ = αναμενόμενος χρόνος εκτέλεσης αλγορίθμου επί όλων των εισόδων μεγέθους n.
- Χρειαζόμαστε υπόθεση της κατανομής (στατιστικής) των εισόδων.

0.4 Σχεδίαση Αλγορίθμων

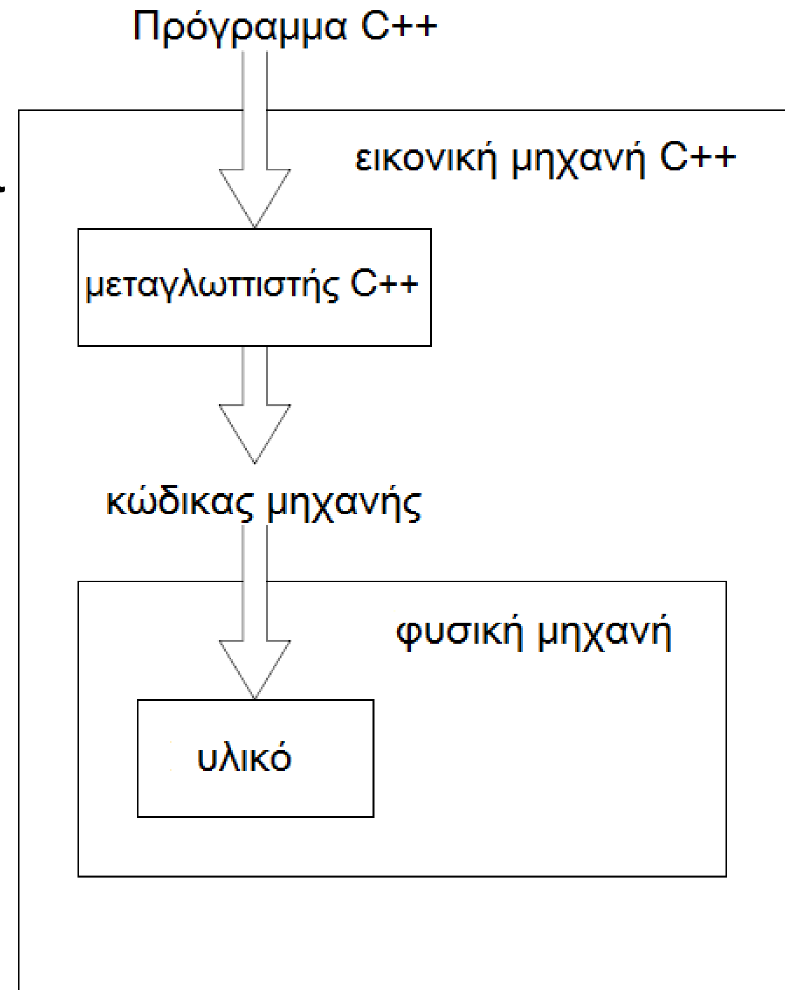
- Πώς σχεδιάζουμε έναν αλγόριθμο για να λύσουμε ένα δεδομένο πρόβλημα;
- ➢ Είναι αντικείμενο του μαθήματος «Σχεδίαση Αλγορίθμων» το να μάθουμε διάφορες γενικές τεχνικές σχεδίασης.
- Τι είναι μια τεχνική σχεδίασης αλγορίθμου;
- ➢ είναι μια γενική προσέγγιση επίλυσης προβλημάτων αλγοριθμικά, η οποία είναι εφαρμόσιμη σε μια ποικιλία προβλημάτων από διάφορες περιοχές των υπολογισμών.
- ✓ Χαρακτηριστικό παράδειγμα: Διαίρει και Βασίλευε (Divide and Conquer)

0.5 Σημαντικοί Τύποι Προβλημάτων

- Ταξινόμηση (sorting)
- Αναζήτηση (searching)
- Επεξεργασία συμβολοσειρών (String processing).
- Προβλήματα γράφων
- Συνδυαστικά προβλήματα
- Γεωμετρικά προβλήματα
- Αριθμητικά προβλήματα
- Αριθμητική μεγάλων ακεραίων

1. Ένα Λεπτομερές μοντέλο του Υπολογιστή

- **Περιγράφουμε ένα λεπτομερές μοντέλο των επιδόσεων του χρόνου εκτέλεσης ενός προγράμματος πχ σε C ή σε C++, το οποίο είναι ανεξάρτητο από το υλικό και το λογισμικό του συστήματος.**
- **Αντί να αναλύσουμε τις επιδόσεις μιας συγκεκριμένης αλλά αυθαίρετα επιλεγμένης φυσικής μηχανής, θεωρούμε την υλοποίηση της γλώσσας προγραμματισμού C ή C++ ως ένα είδος «εικονικής μηχανήςC++» (βλ. διπλανή εικόνα)**



1.1 Οι Βασικές Παραδοχές (1)

- **Παραδοχή 1.** **Ο χρόνος που απαιτείται για την ανάκληση (fetch) ενός ακέραιου τελεστέου από τη μνήμη είναι μια σταθερά, τ_{fetch}, και ο χρόνος που απαιτείται για την αποθήκευση ενός ακεραίου (αποτελέσματος) στη μνήμη είναι μια σταθερά τ_{store}.**
- **Σύμφωνα με την Παραδοχή 1 η εντολή ανάθεσης**

```
y = x;
```

έχει χρόνο εκτέλεσης $\tau_{fetch} + \tau_{store}$. (Ο χρόνος που απαιτείται για την ανάκληση της τιμής της μεταβλητής x είναι τ_{fetch} και ο χρόνος που απαιτείται για την αποθήκευση της τιμής στη μεταβλητή y είναι τ_{store}).

- **Εφαρμόζουμε την Παραδοχή 1 και για τις σταθερές. Π.χ. η εντολή ανάθεσης**

```
y = 1;
```

έχει επίσης χρόνο εκτέλεσης $\tau_{fetch} + \tau_{store}$, αφού η σταθερά θα πρέπει να είναι αποθηκευμένη στη μνήμη και είναι φυσικό να περιμένουμε ότι το κόστος ανάκλησής της είναι το ίδιο με αυτό της ανάκλησης οποιουδήποτε άλλου τελεστέου.

Οι Βασικές Παραδοχές (2)

- **Παραδοχή 2.** Οι χρόνοι που απαιτούνται για την εκτέλεση στοιχειωδών πράξεων με ακεραίους, όπως είναι η πρόσθεση, η αφαίρεση, ο πολλαπλασιασμός, η διαίρεση και η σύγκριση, είναι όλοι σταθεροί. Τους συμβολίζουμε με τ_+ , τ_- , $\tau_\times$, $\tau_\div$ και $\tau_<$, αντίστοιχα.
- Σύμφωνα με την Παραδοχή 2 όλες οι απλές πράξεις με ακεραίους μπορούν να γίνουν σε σταθερό χρόνο. Για να είναι κάτι τέτοιο εφικτό θα πρέπει το πλήθος των bit που χρησιμοποιούνται για την αναπαράσταση των αριθμών να είναι συγκεκριμένο (σταθερό). Συνήθως το πλήθος αυτό είναι μεταξύ 16 και 64. Επειδή ακριβώς το πλήθος των bit που χρησιμοποιούνται είναι σταθερό, μπορούμε να πούμε ότι οι χρόνοι εκτέλεσης είναι επίσης σταθεροί. Αν επιτρέπονται οσοδήποτε μεγάλοι ακέραιοι, τότε οι βασικές αριθμητικές πράξεις μπορεί να απαιτούν ένα οσοδήποτε μεγάλο χρονικό διάστημα.

Παράδειγμα

- **Εφαρμόζοντας τις Παραδοχές 1 και 2 μπορούμε να προσδιορίσουμε ότι ο χρόνος εκτέλεσης μιας εντολής όπως η**

```
y = y + 1;
```

είναι $2\tau_{fetch} + \tau_{+} + \tau_{store}$, αφού πρέπει να ανακληθούν δύο τελεστέοι, οι `y` και `1`, να προστεθούν και να αποθηκευτεί το αποτέλεσμα πάλι στην `y`.

- **Η σύνταξη της C παρέχει και άλλους τρόπους έκφρασης της παραπάνω εντολής:**

```
y += 1;
++y;
y++;
```

- **Θα υποθέσουμε ότι αυτοί οι εναλλακτικοί τρόποι απαιτούν ακριβώς τον ίδιο χρόνο εκτέλεσης με αυτόν της παραπάνω αρχικής εντολής.**

Οι Βασικές Παραδοχές (3)

- **Παραδοχή 3.** Ο χρόνος που απαιτείται για την κλήση μιας συνάρτησης είναι μια σταθερά, τ_{call} , και ο χρόνος που απαιτείται για την επιστροφή από μια συνάρτηση είναι μια σταθερά, $\tau_{\texttt{return}}$.

❖ Εννοείται ότι η Παραδοχή 3 δεν αναφέρεται στον χρόνο εκτέλεσης που απαιτείται από τους υπολογισμούς που γίνονται στην ίδια τη συνάρτηση πριν επιστρέψει το αποτέλεσμα.

- **Παραδοχή 4.** Ο χρόνος που απαιτείται για το πέρασμα ενός (ακέραιου) ορίσματος σε μια συνάρτηση ή διαδικασία είναι ο ίδιος με τον χρόνο που απαιτείται για την αποθήκευση ενός ακεραίου στη μνήμη.

➢ Ο λόγος είναι ότι το πέρασμα ενός ορίσματος εννοιολογικά είναι το ίδιο με την ανάθεση της τιμής της πραγματικής παραμέτρου στην τυπική παράμετρο της συνάρτησης.

- Σύμφωνα με την Παραδοχή 4, ο χρόνος εκτέλεσης της εντολής

```
y = f(x);
```

θα είναι $\tau_{fetch} + 2\tau_{store} + \tau_{call} + T_{f(x)}$, όπου είναι ο χρόνος εκτέλεσης της συνάρτησης για την είσοδο `x`. Η πρώτη από τις δύο αποθηκεύσεις οφείλεται στο πέρασμα της παραμέτρου `x` στη συνάρτηση f και η δεύτερη προκύπτει από την ανάθεση στη μεταβλητή `y`.

1.2 Παράδειγμα – άθροισμα Α.Π (1)

- **Εφαρμογή των Παραδοχών 1, 2 και 3 στην ανάλυση του χρόνου εκτέλεσης ενός προγράμματος για τον υπολογισμό του αθροίσματος**

$$\sum_{i=1}^{n} i$$

που είναι το άθροισμα των n όρων της Αριθμητικής Προόδου με πρώτο όρο α_1 = 1 και διαφορά ω = 1.

Πρόγραμμα 1.1: Πρόγραμμα για τον υπολογισμό του $\sum_{i=1}^{n} i$

```
1. int Sum (int n)
2. {
3.      int result = 0;
4.      for (int i = 1; i <= n; ++i)
5.           result += i;
6.      return result;
7. }
```

- **Οι εκτελέσιμες εντολές στο Πρόγραμμα 1.1 περιλαμβάνονται στις γραμμές 3-6. Στον Πίνακα 1.1 δίνονται οι χρόνοι εκτέλεσης κάθε μίας απ' αυτές τις εντολές:**

Παράδειγμα – άθροισμα Α.Π (2)

Πίνακας 1.1: Υπολογισμός χρόνου εκτέλεσης για το Πρόγραμμα 1.1

εντολή	χρόνος	κώδικας
3	$\tau_{\text{fetch}} + \tau_{\text{store}}$	`result = 0`
4a	$\tau_{\text{fetch}} + \tau_{\text{store}}$	`i = 1`
4b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n+1)$	`i <= n`
4c	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	`++i`
5	$(2\tau_{\text{fetch}} + \tau_{+} + \tau_{\text{store}}) \times n$	`result += i`
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$	`return result`
ΣΥΝΟΛΟ	$(6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + 2\tau_{+}) \times n$ $+ (5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{\text{return}})$	

- Η εντολή for στη γραμμή 4 παρουσιάζεται σε τρεις γραμμές διότι αναλύουμε τον χρόνο εκτέλεσης καθενός από τα στοιχεία μιας εντολής for χωριστά. Το 1ο στοιχείο, *ο κώδικας αρχικοποίησης*, εκτελείται μία φορά πριν την πρώτη επανάληψη του βρόχου. Το 2ο στοιχείο, *ο έλεγχος τερματισμού βρόχου*, εκτελείται πριν αρχίσει η κάθε μια επανάληψη του βρόχου. Συνολικά ο αριθμός των εκτελέσεων του ελέγχου τερματισμού είναι μία παραπάνω από τον αριθμό των εκτελέσεων του σώματος του βρόχου. Το 3ο τελικά στοιχείο, *το βήμα αύξησης του μετρητή βρόχου*, εκτελείται μία φορά ανά επανάληψη του βρόχου.

```
1. int Sum (int n)
2. {
3.      int result = 0;
4.      for (int i = 1; i <= n; ++i)
5.           result += i;
6.      return result;
7. }
```

Παράδειγμα – άθροισμα Α.Π (3)

- **Αθροίζοντας τις καταχωρήσεις του Πιν. 1.1 προκύπτει ότι ο χρόνος εκτέλεσης, $T(n)$, για το Πρόγραμμα 1.1, είναι**

$$T(n) = t_1 + t_2 n \qquad (1.1)$$

όπου

$$t_1 = 5\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{\texttt{return}}$$

$$t_2 = 6\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + 2\tau_{\texttt{return}}$$

1.3 Πράξεις δεικτοδότησης

- **Γενικά, τα στοιχεία μιας μονοδιάστατης συστοιχίας (array) αποθηκεύονται σε διαδοχικές θέσεις της μνήμης. Επομένως δοθείσης της διεύθυνσης του 1^{ου} στοιχείου της συστοιχίας, μια απλή πρόσθεση αρκεί για τον προσδιορισμό της διεύθυνσης του οποιουδήποτε στοιχείου της συστοιχίας.**
- **Παραδοχή 5. Ο χρόνος που απαιτείται για τον υπολογισμό της διεύθυνσης που προκύπτει από μια πράξη δεικτοδότησης συστοιχίας, π.χ. `a[i]`, είναι μια σταθερά $\tau_{[\cdot]}$. Ο χρόνος αυτός δεν συμπεριλαμβάνει τον χρόνο για τον υπολογισμό της έκφρασης δείκτη ούτε τον χρόνο για την προσπέλαση (δηλ. ανάκληση ή αποθήκευση) του στοιχείου της συστοιχίας.**
- **Εφαρμόζοντας την Παραδοχή 5 μπορούμε να προσδιορίσουμε ότι ο χρόνος εκτέλεσης της εντολής**

```
y = a[i];
```

είναι $3\tau_{fetch} + \tau_{[\cdot]} + \tau_{store}$. Απαιτούνται τρεις ανακλήσεις τελεστέων: η 1^{η} είναι για την ανάκληση της `a`, της διεύθυνσης βάσης της συστοιχίας˙ η 2^{η} για την ανάκληση του `i`, του δείκτη μέσα στη συστοιχία˙ και η 3^{η} για την ανάκληση του στοιχείου `a[i]` της συστοιχίας.

1.4 Παράδειγμα – Κανόνας του Horner (1)

- **Εφαρμογή των Παραδοχών 1, 2, 3 και 4 στην ανάλυση του χρόνου εκτέλεσης ενός προγράμματος που υπολογίζει την τιμή ενός πολυωνύμου. Δηλ. δοθέντων των $n + 1$ συντελεστών a_0, a_1, …, a_n και της τιμής x, υπολογίζει το άθροισμα** $$\sum_{i=0}^{n} a_i x^i$$
- **Ένας γνωστός τρόπος υπολογισμού είναι ο κανόνας του Horner που είναι στην ουσία ένας αλγόριθμος για τον υπολογισμό του αθροίσματος χωρίς να απαιτείται ο υπολογισμός οποιασδήποτε δύναμης του x.**

Πρόγραμμα 1.2: Υπολογισμός του $\sum_{i=0}^{n} a_i x^i$ με τον κανόνα του Horner

```
1. int Horner (int a[], int n, int x)
2. {
3.      int result = a[n];
4.      for ( int i = n - 1; i >= 0; --i )
5.           result = result * x + a[i];
6.      return result;
7. }
```

Παράδειγμα – Κανόνας του Horner (2)

- **Ο χρόνος εκτέλεσης των εκτελέσιμων εντολών του Προγράμματος 1.2 δίνεται στον Πιν. 1.2:**

Πίνακας 1.2: Υπολογισμός του χρόνου εκτέλεσης για το Πρόγραμμα 1.2

εντολή	χρόνος
3	$3\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{\text{store}}$
4a	$2\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}}$
4b	$(2\tau_{\text{fetch}} + \tau_{<}) \times (n+1)$
4c	$(2\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}}) \times n$
5	$(5\tau_{\text{fetch}} + \tau_{[\cdot]} + \tau_{+} + \tau_{\times} + \tau_{\text{store}}) \times n$
6	$\tau_{\text{fetch}} + \tau_{\text{return}}$
ΣΥΝΟΛΟ	$(9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{[\cdot]} + \tau_{+} + \tau_{\times} + \tau_{-}) \times n$ $+ (8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[\cdot]} + \tau_{-} + \tau_{<} + \tau_{\text{return}})$

- **Ο χρόνος εκτέλεσης $T(n)$ προκύπτει ότι είναι**

$$T(n) = t_1 + t_2 n \qquad (1.2)$$

όπου

$$t_1 = 8\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{[\cdot]} + \tau_{-} + \tau_{<} + \tau_{\text{return}}$$

$$t_2 = 9\tau_{\text{fetch}} + 2\tau_{\text{store}} + \tau_{<} + \tau_{[\cdot]} + \tau_{+} + \tau_{\times} + \tau_{-}$$

1.5 Ανάλυση Αναδρομικών Προγραμμάτων (1)

- **Ανάλυση της επίδοσης ενός αναδρομικού αλγορίθμου που υπολογίζει το παραγοντικό ενός ακεραίου. Υπενθυμίζουμε ότι το παραγοντικό ενός μη αρνητικού ακεραίου n, συμβολικά $n!$, ορίζεται ως**

$$n! = \begin{cases} 1, & n = 0 \\ \prod_{i=1}^{n} i, & n > 0 \end{cases}$$

- **Μπορούμε όμως να ορίσουμε το παραγοντικό *αναδρομικά*:**

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

- **Με βάση τον ορισμό αυτό οδηγούμαστε στον αλγόριθμο που δίνεται στο Πρόγραμμα 1.3 για τον υπολογισμό του παραγοντικού του φυσικού αριθμού n. Στον Πίν. 1.3 δίνονται οι χρόνοι εκτέλεσης των εκτελέσιμων εντολών του προγράμματος.**

Ανάλυση Αναδρομικών Προγραμμάτων (2)

Πρόγραμμα 1.3: Αναδρομικό πρόγραμμα για τον υπολογισμό του n!

```
1. int Factorial (int n)
2. {
3.     if (n == 0)
4.         return 1;
5.     else
6.         return n * Factorial (n - 1);
7. }
```

Πίνακας 1.3: Υπολογισμός του χρόνου εκτέλεσης του Προγράμματος 1.3

	χρόνος	
εντολή	$n = 0$	$n > 0$
3	$2\tau_{\text{fetch}} + \tau_{<}$	$2\tau_{\text{fetch}} + \tau_{<}$
4	$\tau_{\text{fetch}} + \tau_{\text{return}}$	—
6	—	$3\tau_{\text{fetch}} + \tau_{-} + \tau_{\text{store}} + \tau_{\times}$ $+ \tau_{\text{call}} + \tau_{\text{return}} + T(n-1)$

Ανάλυση Αναδρομικών Προγραμμάτων (3)

- **Είχαμε να αναλύσουμε τον χρόνο εκτέλεσης των δύο πιθανών αποτελεσμάτων του ελέγχου συνθήκης στη γραμμή 3 χωριστά. Είναι φανερό ότι ο χρόνος εκτέλεσης του προγράμματος εξαρτάται από το αποτέλεσμα αυτού του ελέγχου.**

```
1. int Factorial (int n)
2. {
3.     if (n == 0)
4.         return 1;
5.     else
6.         return n * Factorial (n – 1);
7. }
```

- **Επιπρόσθετα, η συνάρτηση Factorial καλεί τον εαυτό της αναδρομικά στη γραμμή 6. Επομένως, για να γράψουμε τον χρόνο εκτέλεσης της γραμμής 6 πρέπει να γνωρίζουμε τον χρόνο εκτέλεσης της T(·) της Factorial. Αλλά αυτό είναι ακριβώς που προσπαθούμε να προσδιορίσουμε από την αρχή. Υποθέτουμε λοιπόν ότι γνωρίζουμε ήδη ποια είναι η συνάρτηση T(·) και ότι μπορούμε να κάνουμε χρήση αυτής της συνάρτησης για τον προσδιορισμό του χρόνου εκτέλεσης της γραμμής 6.**
- **Αθροίζοντας τις στήλες του Πιν. 1.3 παίρνουμε ότι ο χρόνος εκτέλεσης για το Πρόγραμμα 1.3 είναι**

$$T(n) = \begin{cases} t_1, & n = 0 \\ T(n-1) + t_2, & n > 0 \end{cases} \quad (1.4)$$

όπου $t_1 = 3\tau_{fetch} + \tau_{<} + \tau_{return}$

$t_2 = 5\tau_{fetch} + \tau_{<} + \tau_{-} + \tau_{store} + \tau_{\times} + \tau_{call} + \tau_{return}$

Ανάλυση Αναδρομικών Συναρτήσεων (4)

- **Η (1.4) είναι μια *αναδρομική σχέση* επειδή η συνάρτηση ορίζεται συναρτήσει του εαυτού της αναδρομικά.**
- **Επίλυση αναδρομικών σχέσεων (σε επόμενη διάλεξη)**
- **Η επίλυση της (1.4) μας δίνει**

$$T(n) = t_1 + t_2 n$$

- **όπου t_1 και t_2 οι σταθερές που έχουμε αναφέρει στην προηγούμενη διαφάνεια.**

2. Απλοποιημένο μοντέλο του υπολογιστή

- **Το λεπτομερές μοντέλο του υπολογιστή που χρησιμοποιήσαμε στην προηγούμενη ενότητα έχει πολλές παραμέτρους χρονομέτρησης: τ_{fetch} , τ_{store} , τ_{+} , τ_{-} , $\tau_{\times}$, $\tau_{\div}$, $\tau_{<}$, τ_{call} , τ_{return} και $\tau_{[\cdot]}$. Είναι αρκετά επίπονο να παρακολουθούμε κατά την ανάλυση όλες τις παραμέτρους.**
- **Σε μια πραγματική μηχανή κάθε μία απ' αυτές τις παραμέτρους είναι ένα πολλαπλάσιο της περιόδου του ρολογιού της μηχανής, *kT*.**
- **Το απλοποιημένο μοντέλο απαλείφει όλες αυτές τις παραμέτρους:**
  - **όλες οι παράμετροι χρονομέτρησης εκφράζονται σε κύκλους ρολογιού. Ουσιαστικά, *T* = 1.**
  - **Η σταθερά *k* για όλες τις παραμέτρους χρονομέτρησης υποθέτουμε ότι είναι η ίδια: *k* = 1.**
- **Το αποτέλεσμα αυτών των δύο υποθέσεων είναι ότι δεν χρειάζεται να διαχωρίζουμε τις διάφορες πράξεις. Για τον προσδιορισμό του χρόνου εκτέλεσης ενός προγράμματος απλά μετράμε τον συνολικό αριθμό των κύκλων που πραγματοποιούνται.**

2.1 Παράδειγμα – Άθροισμα δυναμοσειράς (Ι.1)

- **Θεωρούμε το πρόβλημα του χρόνου εκτέλεσης ενός προγράμματος για τον υπολογισμό της ακόλουθης δυναμοσειράς: δοθείσης μιας τιμής x και ενός μη αρνητικού ακεραίου n, θέλουμε να υπολογίσουμε το άθροισμα**

$$\sum_{i=0}^{n} x^i$$

- **Ένας αλγόριθμος υπολογισμού του αθροίσματος δίνεται στο Προγ. 2.5.**

Πρόγραμμα 2.5: Υπολογισμός του $\sum_{i=0}^{n} x^i$

```
1. int GeometricalSeriesSum (int x, int n)
2. {
3.       int sum = 0;
4.       for (int i = 0; i <= n; ++i)
5.       {
6.           int prod = 1;
7.           for (int j = 0; j < i; ++j)
8.              prod *= x;
9.           sum += prod;
10.      }
11.      return sum;
12.}
```

Παράδειγμα – Άθροισμα δυναμοσειράς (Ι.2)

- **Ο Πιν. 2.5 παρουσιάζει τον χρόνο εκτέλεσης, όπως προβλέπεται με το απλοποιημένο μοντέλο, για κάθε μία από τις εκτελέσιμες εντολές του Προγ. 2.5.**

Πίνακας 2.5: Υπολογισμός του χρόνου εκτέλεσης του Προγ. 2.5

εντολή	χρόνος
3	2
4a	2
4b	$3(n+2)$
4c	$4(n+1)$
6	$2(n+1)$
7a	$2(n+1)$
7b	$3\sum_{i=0}^{n}(i+1)$
7c	$4\sum_{i=0}^{n} i$
8	$4\sum_{i=0}^{n} i$
9	$4(n+1)$
10	2
ΣΥΝΟΛΟ	$\frac{11}{2}n^2 + \frac{47}{2}n + 24$

```
1.  int GeometricalSeriesSum (int x,
    int n)
2.  {
3.     int sum = 0;
4.     for (int i = 0; i <= n; ++i)
5.     {
6.        int prod = 1;
7.        for (int j = 0; j < i; ++j)
8.           prod *= x;
9.           sum += prod;
10.     }
11.     return sum;
12. }
```

Παράδειγμα – Άθροισμα δυναμοσειράς (Ι.3)

- **Για να υπολογίσουμε τον συνολικό αριθμό κύκλων πρέπει να υπολογίσουμε τα δύο αθροίσματα**

$$\sum_{i=0}^{n}(i+1) \quad \textbf{και} \quad \sum_{i=0}^{n} i$$

τα οποία είναι αθροίσματα n + 1 όρων μιας Αριθμητικής Προόδου και υπολογίζονται με βάση τον τύπο

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- **Χρησιμοποιώντας αυτόν τον τύπο μπορούμε να αθροίσουμε το πλήθος των κύκλων που δίνονται στον Πιν. 2.5 και να βρούμε τελικά ότι ο συνολικός χρόνος εκτέλεσης είναι**

$$\frac{11}{2}n^2 + \frac{47}{2}n + 24$$

2.2 Παράδειγμα – Άθροισμα δυναμοσειράς (ΙΙ.1)

- **Μπορούμε να δούμε το προηγούμενο πρόβλημα υπολογισμού του αθροίσματος της δυναμοσειράς ως πρόβλημα υπολογισμού της τιμής ενός πολυωνύμου του οποίου όλοι οι συντελεστές είναι ίσοι με 1. Έτσι θα μπορούσαμε να χρησιμοποιήσουμε τον κανόνα του Horner που είδαμε στην ενότητα 1. 4. Ένας αλγόριθμος βασισμένος σ' αυτή την ιδέα παρουσιάζεται στο Προγ. 2.6.**

Πρόγραμμα 2.6: Υπολογισμός του $\sum_{i=0}^{n} x^i$ με τον κανόνα του Horner

```
1. int GeometricalSeriesSum (int x, int n)
2. {
3.       int sum = 0;
4.       for (int i = 0; i <= n; ++i)
5.            sum = sum * x + 1;
6.       return sum;
7. }
```

- **Οι εκτελέσιμες εντολές του Προγ. 2.6 συνιστούν τις γραμμές 3-6. Ο Πιν. 2.6 παρουσιάζει τους χρόνους εκτέλεσης, σύμφωνα με το απλοποιημένο μοντέλο, για κάθε μία από τις εντολές αυτές.**

Παράδειγμα – Άθροισμα δυναμοσειράς (ΙΙ.2)

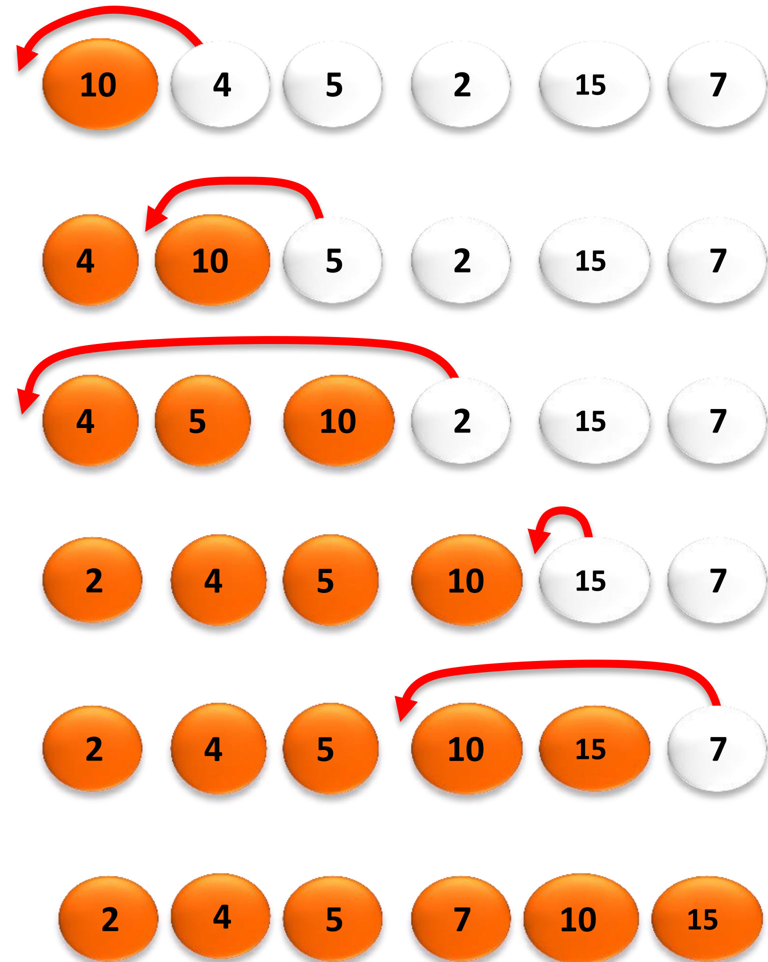
Πίνακας 2.6: Υπολογισμός του χρόνου εκτέλεσης του Προγ. 2.6

εντολή	χρόνος
3	2
4a	2
4b	$3(n+2)$
4c	$4(n+1)$
5	$6(n+1)$
6	2
ΣΥΝΟΛΟ	$13n+22$

```
1. int GeometricalSeriesSum (int x, int n)
2. {
3.       int sum = 0;
4.       for (int i = 0; i <= n; ++i)
5.             sum = sum * x + 1;
6.       return sum;
7. }
```

- **Στα Προγράμματα 2.5 και 2.6 έχουμε δύο διαφορετικούς αλγόριθμους για τον υπολογισμό του ίδιου αθροίσματος μιας δυναμοσειράς. Ο χρόνος εκτέλεσης του πρώτου είναι $(11/2)n^2 + (47/2)n + 27$ κύκλοι και του δεύτερου $13n + 22$ κύκλοι. Επειδή $(11/2)n^2 + (47/2)n + 27 > 13n + 22$ για κάθε n, μπορούμε να πούμε ότι, σύμφωνα με το απλοποιημένο μοντέλο του υπολογιστή, το Προγ. 2.6 που χρησιμοποιεί τον κανόνα του Horner εκτελείται πάντα πιο γρήγορα από το Προγ. 2.5.**

Παράδειγμα: Ταξινόμηση με ένθεση (εισαγωγή)



Παράδειγμα: Ταξινόμηση με ένθεση

```
InsertionSort(A, n) {
  for j = 2 to n {
      key = A[j]
      i = j - 1;
      while (i > 0) and (A[i] > key) {
          A[i+1] = A[i]
          i = i - 1
      }
      A[i+1] = key
  }
}
```

Πόσες φορές θα εκτελεστεί αυτός ο βρόχος;

Παράδειγμα: Ταξινόμηση με ένθεση

Εντολή	κόστος
`InsertionSort(A, n) {`	
`   for j = 2 to n {`	$c_1 n$
`        key = A[j]`	$c_2(n-1)$
`        i = j - 1;`	$c_3(n-1)$
`        while (i > 0) and (A[i] > key) {`	$c_4 \sum_{j=2}^{n} t_j$
`             A[i+1] = A[i]`	$c_5 \sum_{j=2}^{n} (t_j - 1)$
`             i = i - 1`	$c_6 \sum_{i=2}^{n} (t_j - 1)$
`        }`	0
`        A[i+1] = key`	$c_7(n-1)$
`   }`	0
`}`	

$T = t_2 + t_3 + \ldots + t_n = \sum_{j=2}^{n} t_j$, όπου t_j είναι το πλήθος των αποτιμήσεων της παράστασης **whil**e για την j-οστή επανάληψη του βρόχου **for**.

Ανάλυση του InsertionSort

- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 T + c_5(T - (n-1)) + c_6(T - (n-1)) + c_7(n-1)$

  $= c_8 T + c_9 n + c_{10}$
- Τι μπορεί να είναι το T;
  - **Καλύτερη περίπτωση** – η είσοδος είναι ήδη ταξινομημένη, οπότε το σώμα του εσωτερικού βρόχου δεν εκτελείται ποτέ
    - $t_j = 1 \Rightarrow T(n)$ είναι μια γραμμική συνάρτηση
  - **Χειρότερη περίπτωση** – η είσοδος είναι ταξινομημένη αντίστροφα, οπότε το σώμα του εσωτερικού βρόχου εκτελείται για όλα τα προηγούμενα στοιχεία
    - $t_j = j \Rightarrow T(n)$ είναι μια τετραγωνική συνάρτηση
  - **Μέση περίπτωση** – Κατά μέσο όρο θα πρέπει να ελέγξουμε τα μισά στοιχεία της $A[1..j-1]$
    - $t_j = j/2 \Rightarrow T(n)$ είναι μια τετραγωνική συνάρτηση.

Ταξινόμηση με εισαγωγή: Χρονική πολυπλοκότητα

Αλγόριθμος InsertionSort(A[1..n])
Είσοδος: A[1..n] μια συστοιχία n αριθμών
Έξοδος: A[1..n] μια συστοιχία n αριθμών ταξινομημένη

```
1.  for j ← 2 .. n
        // η A[1..j –1] είναι ταξινομημένη, η A[j..n] είναι ανέγγιχτη.
        // ενθέτουμε το A[j] στην ταξινομημένη ακολουθίαA[1..j–1]
2.      key ← A[j]
3.      i ← j –1
4.      while i > 0 and A[i] > key
5.          A[i+1] ← A[i]
6.          i ← i –1
7.      A[i+1] ← key
```

$$T(n) = n + \sum_{j=2}^{n}\left(2 + (t_j + t_j - 1 + t_j - 1) + 1\right)$$

$$= n + \sum_{j=2}^{n}\left(1 + 3t_j\right)$$

$$= n + n - 1 + 3\sum_{j=2}^{n} t_j$$

$$= 2n - 1 + 3\sum_{j=2}^{n} j$$ Χειρότερη περίπτωση: $t_j = j$ επαναλήψεις (συστοιχία ταξινομημένη με αντίστροφη σειρά).

$$= \frac{3}{2}n^2 + \frac{7}{2}n - 4 \approx n^2$$

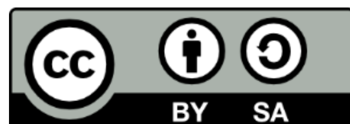
$$\sum_{j=2}^{n} j = \sum_{j=1}^{n} j - 1 = \frac{n(n+1)}{2} - 1.$$

Βιβλιογραφία

- **T. Cormen, C. Leiserson, R. Rivest, C. Stein,** ***Εισαγωγή στους αλγορίθμους***
- **B.R. Preiss,** ***Data Structures and Algorithms with Object-Oriented Design Patterns in C++***

  http://www.brpreiss.com/books/opus4/
- **A. Aho – J. Ullman,** ***Ο Χρόνος Εκτέλεσης Προγραμμάτων***

Τέλος Ενότητας



Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης