Διάλεξη 04: Παραδείγματα Ανάλυσης Πολυπλοκότητας/Ανάλυση Αναδρομικών Αλγόριθμων

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- Παραδείγματα Ανάλυσης Πολυπλοκότητας : Μέθοδοι, παραδείγματα
- Γραμμική και Δυαδική Αναζήτηση, Ανάλυση Αναδρομικών Αλγορίθμων
- Η Μέθοδος της Αντικατάστασης, Master Theorem

Διδάσκων: Παναγιώτης Ανδρέου

Παράδειγμα 1: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i,k=0;
for (i=0; i<n; i++)
            k+=i;
```

Ανάλυση

- k+=i; Η βασική πράξη η οποία χρειάζεται **O(1)** χρόνο
- for(i=0; i<n; i++) Ο αλγόριθμος εκτελεί την βασική πράξη **n φορές**

$$\sum_{i=0}^{n-1} 1 = n \quad \textbf{ή} \quad \sum_{i \in n} 1 = n$$

- Ο αλγόριθμος έχει πολυπλοκότητα της τάξης **O(n)**
- Στην βέλτιστη περίπτωση χρειάζεται **Ω(n) ➔ O(n) & Ω(n) ➔ Θ(n)**

Υπολογισμός Χρόνου Εκτέλ. με φωλιασμ. βρόγχους

Σε ένα βρόχο (for loop) ο συνολικός χρόνος που απαιτείται είναι: **Βασική Πράξη x Αριθμό Επαναλήψεων**

Φωλιασμένοι Βρόχοι: η ανάλυση γίνεται από τα μέσα προς τα έξω:

Παράδειγμα:

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        //statement e.g., k+=i;
```

Συνεχόμενες Εντολές: Ο χρόνος εκτέλεσης T της εντολής S και μετά S΄ παίρνει χρόνο ίσο του αθροίσματος των χρόνων εκτέλεσης των T(S) + T(S΄).

Συνθήκες if: Ο χρόνος εκτέλεσης T της εντολής if b then S else S΄ παίρνει χρόνο ίσο με max(T(b)+T(S) , T(b)+T(S’))

Παράδειγμα 2: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, sum=0;
  for (i=0; i<n; i++)          Εξωτερικός Βρόγχος
    for (j=0; j<n; j++)        Εσωτερικός Βρόγχος
            sum++;
```

Ανάλυση

- **Εσωτερικός Βρόγχος:** $IL = \sum_{j=0}^{n-1} 1 = \sum_{j=1}^{n} 1 = n$
- **Εξωτερικός Βρόγχος:** $\sum_{i=1}^{n} IL = n \times IL$
- **Σύνολο:** $\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 \in O(n^2), \Omega(n^2) \Rightarrow \Theta(n^2)$

Παράδειγμα 3: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, sum=0;
  for (i=0; i<n; i++)                    Εξωτερικός Βρόγχος
    for (j=0; j< i*i; j++)               Εσωτερικός Βρόγχος
              sum++;
```

Ανάλυση

- **Εσωτερικός Βρόγχος:** $IL = \sum_{j=0}^{i^2} 1 = i^2$

- Παρατηρούμε ότι ο χρόνος εκτέλεσης του εσωτερικού βρόχου **εξαρτάται από την τιμή i**, οποία καθορίζεται από τον εξωτερικό βρόχο.

Παράδειγμα 3: Υπολογισμ. Χρόνου Εκτέλεσης (συν.)

Ο πιο κάτω πίνακας δείχνει το πόσες φορές εκτελείται ο εσωτερικός βρόγχος σαν συνάρτηση του i:

i	0	1	2	3	…	n-1
IL = i^2	0	1	4	9	…	$(n-1)^2$

- **Εξωτερικός Βρόγχος:** **$0 + 1 + 4 + 9 + \ldots + (n-1)^2$**
- **Σύνολο: Ο χρόνος εκτέλεσης του προγράμματος είναι ίσος με το άθροισμα του χρόνου εκτέλεσης κάθε επανάληψης του εσωτερικού βρόχου:**

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{6}(2n^3 + n^2 + 2n^2 + n) \in \Theta(n^3)$$

Παράδειγμα 4: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, k, sum=0;
  for (i=0; i<n; i++)                    Εξωτερικός Βρόγχος
    for (j=0; j<n; j++)                  Εσωτερικός Βρόγχος B
      for (k=0; k<n; k++)                Εσωτερικός Βρόγχος A
              sum++;
```

Ανάλυση

- **Εσωτερικός Βρόγχος A:** $IL_A = \sum_{k \in n} 1 = n$
- **Εσωτερικός Βρόγχος B:** $IL_B = \sum_{j \in n} IL_A = \sum_{j \in n} n = n \times n = n^2$
- **Εξωτερικός Βρόγχος:** $OL = \sum_{i \in n} IL_B = \sum_{j \in n} n^2 = n \times n^2 = n^3$
- **Σύνολο:** $\in \Theta(n^3)$

Παράδειγμα 5: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, sum=0;
for (i=1; i<=n; i=2*i)            Εξωτερικός Βρόγχος
    for (j=0; j< n; j++)          Εσωτερικός Βρόγχος
        sum++;
```

Ανάλυση

- **Εσωτερικός Βρόγχος:** $IL = \sum_{j \in n} 1 = n$
- **Εξωτερικός Βρόγχος:** **Κάθε φορά, το i πολλαπλασιάζεται επί 2.**
- **Παράδειγμα:** για **n=10** το **i** θα είναι 1, 2, 4, 8, 16>n break

Πόσες φορές εκτελείται;;;

Παράδειγμα 5: Υπολογισμ. Χρόνου Εκτέλεσης (συν.)

- **Ας υποθέσουμε ότι το n είναι μία δύναμη του 2 ➔ $n=2^k$**

Ο πιο κάτω πίνακας δείχνει το πόσες φορές εκτελείται ο εσωτερικός βρόγχος σαν συνάρτηση του k:

k	0	1	2	3	4	…	k
$n=2^k$	1	2	4	8	16	…	2^k
τιμές i	-	1	1,2	1,2,4	1,2,4,8	…	
επαναλ.	0	1	2	3	4	…	k

- **Συμπέρασμα:** **i=k,** άρα για **$n=2^i$** θα χρειαστούν **i** επαναλ.
- **Ερώτηση:** πως μπορούμε να βρούμε το την τιμή i έχοντας σαν δεδομένο το 2^i;
- **Απάντηση: με χρήση λογάριθμου (log): $\log_2 2^i = i$**

Παράδειγμα 5: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, sum=0;
for (i=1; i<=n; i=2*i)          Εξωτερικός Βρόγχος
  for (j=0; j< n; j++)          Εσωτερικός Βρόγχος
          sum++;
```

Ανάλυση

- **Εσωτερικός Βρόγχος:** $IL = \sum_{j \in n} 1 = n$
- **Εξωτερικός Βρόγχος: ($\log_2 2^i = \log_2 n$)**

$$\log_2 n \times IL = \log_2 n \times n$$

- **Σύνολο:** $\in \Theta(n \log_2 n)$

Παράδειγμα 6: Υπολογισμός Χρόνου Εκτέλεσης

Υποθέστε ότι ένα πρόβλημα επιλύνεται με τον πιο κάτω κώδικα:

```
int i, j, sum=0;
  for (i=0; i<n; i++)                    Εξωτερικός Βρόγχος
    if( (n%2)==0 )                       Εσωτερικός Βρόγχος (if)
      for (j=0; j<n; j++)
          sum++;
    else                                 Εσωτερικός Βρόγχος (else)
      sum--;
```

Ανάλυση

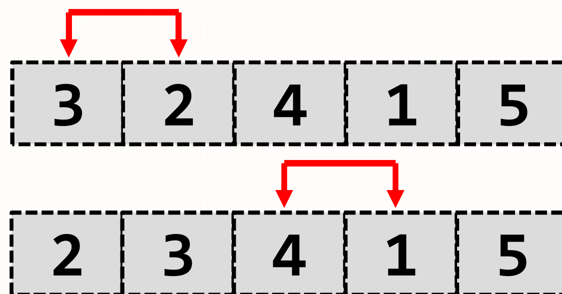
- **n=περιττός → Εσωτερικός Βρόγχος (else):** $IL_{(else)} = 1$
- **n=άρτιος → Εσωτερικός Βρόγχος (if):** $IL_{(if)} = \sum_{j \in n} 1 = n$
- **Εξωτερικός Βρόγχος:** $OL = \sum_{i \in n} \max(IL_{if}, IL_{else}) = \sum_{i \in n} IL_{if} = \sum_{i \in n} n = n^2$
- **Σύνολο:** $\in \Theta(n^2)$

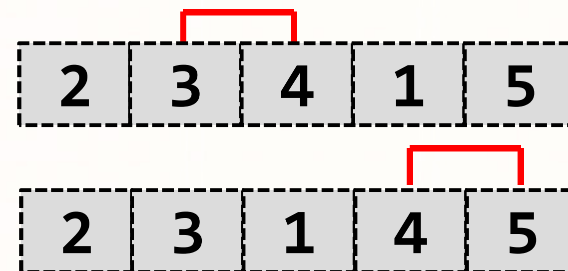
Παράδειγμα 7: Χρόνος Εκτέλεσης BubbleSort

Η BubbleSort ταξινομεί κάποιο πίνακα με συνεχή εναλλαγή των στοιχείων του αν δεν είναι στη σωστή σειρά.

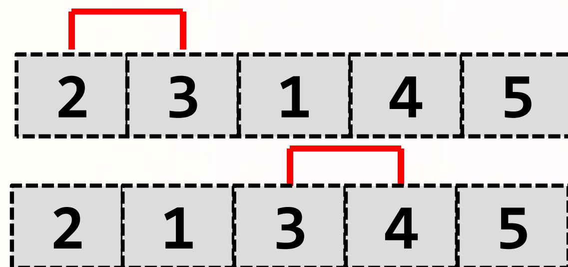
Παράδειγμα: `int x = {3, 2, 4, 1, 5};` swap no swap

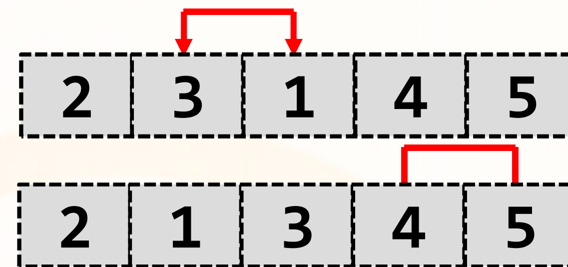
Πέρασμα 1



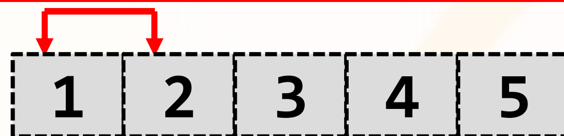


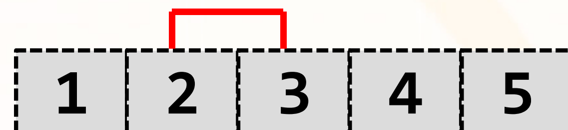
Πέρασμα 2





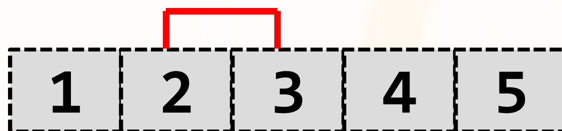
Πέρασμα 3





…

Πέρασμα 4



…

Παράδειγμα 7: Χρόνος Εκτέλεσης BubbleSort (συν.)

```
void bubblesort( int X[], int n){
   int i, j, temp;
   bool swapped;
   for (i=0;i<n-1;i++)  {
      swapped = false;
      for (j=0;j<n-i-1;j++)  {
         if (X[j] > X[j+1]) {
            temp = X[j];
            X[j] = X[j+1];
            X[j+1] = temp;
            swapped = true;
         }
      }
      if (swapped==false) return;
   }
}
```

Εσωτερικός Βρόγχος

Εξωτερικός Βρόγχος

Παράδειγμα 7: Χρόνος Εκτέλεσης BubbleSort (συν.)

```
void bubblesort( int X[], int n){
    …
    for (i=0;i<n-1;i++)  {
       for (j=0;j<n-i-1;j++)  {
          …
       }                    Εσωτερικός Βρόγχος
    }                        Εξωτερικός Βρόγχος
}
```

Ανάλυση

- **Εσωτερικός Βρόγχος:** $IL = \sum_{j=0}^{n-i-1} 1 = n - i - 1$
- **Εξωτερικός Βρόγχος:** $OL = \sum_{i=0}^{n-1} IL = \sum_{i=0}^{n-1} n - i - 1 = \sum_{i=0}^{n-1} n - 1 - \sum_{i=0}^{n-1} i$
  $= (n-1)^2 - \frac{(n-1)n}{2} = \frac{1}{2}(n-2)(n-1)$
- **Σύνολο:** $\in \Theta(n^2)$

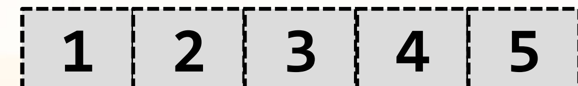
Γραμμική vs. Δυαδική Διερεύνηση

Δεδομένα Εισόδου: Πίνακας X με n στοιχεία, ταξινομημένος από το μικρότερο στο μεγαλύτερο, και ακέραιος k.

Στόχος: Να εξακριβώσουμε αν το k είναι στοιχείο του X.

Γραμμική Διερεύνηση: εξερευνούμε τον πίνακα από τα αριστερά στα δεξιά.

```
int linear( int X[], int n, int k){
  int i=0;
  while ( i < n ) {
    if (X[i] == k) return i;
    if (X[i] > k)  return -1;
    i++;
  }
  return -1;
}
```



Χείριστη περίπτωση: **O(n)** (ο βρόχος εκτελείται n φορές)

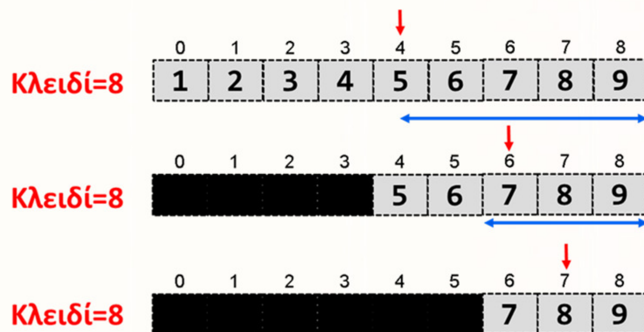
Αναδρομική Γραμμική Διερεύνηση

```
int rlinear( int X[], int n, int k, int pos ){
  if ( pos == n )        return -1;  //not found

  if ( X[pos] == k )     return pos; //found
  elseif ( X[pos] > k ) return -1;  //found larger-skip rest

  return rlinear( X, n, k, pos+1 );
}
```

Χείριστη περίπτωση: **O(n)** (εκτελούνται n αναδρομικές κλήσεις της rlinear)

Δυαδική Διερεύνηση

Δυαδική Διερεύνηση: βρίσκουμε το μέσο του πίνακα και αποφασίζουμε αν το k ανήκει στο δεξιό ή αριστερό μισό. Επαναλαμβάνουμε την ίδια διαδικασία στο μισό που μας ενδιαφέρει:




```
int binary (int X[], int n, int key) {
  int low=0;
  int high = n - 1;
  int mid;
  while( low <= high) {
    mid = (low + high) / 2;
    if( key < X[mid])
      high = mid-1;
    else if (key > X[mid])
      low = mid+1;
    else {
      return mid; break;
    }
  }
  return -1;
}
```

Αναδρομική Δυαδική Διερεύνηση

```
int rbinary_aux (int X[], int low, int high, int key) {
  int mid;
  if( low <= high) {
    mid = (low + high) / 2;
    if( key == X[mid])
      return mid;
    else if( key < X[mid])
      return rbinary_aux( X, low, mid-1, key);
    else
      return rbinary_aux( X, mid+1, high, key);
  }
  return -1;
}
```

```
int rbinary (int X[], int n, int key) {
  int low  = 0;
  int high = n-1;
  return rbinary_aux( X, low, high, key);
}
```

Δυαδική Διερεύνηση – Χρόνος Εκτέλεσης

- Η βασική πράξη (σύγκριση) εκτελείται $O(\log_2 n)$ φορές δηλαδή:

Εκτέλεση **1** -> Μας απομένει* **n/2** του πίνακα,

Εκτέλεση **2** -> Μας απομένει **n/4** του πίνακα,

Εκτέλεση **3** -> Μας απομένει **n/8** του πίνακα,

… … … … … …

Εκτέλεση X -> Μας απομένει **1** στοιχείο του πίνακα,

Στην εκτέλεση X είτε βρήκαμε το στοιχείο είτε όχι

δηλ., έχουμε την ακολουθία **n, n/2, n/4, n/8,…, 4, 2, 1,**

<==> $2^0, 2^1, 2^2, 2^3, \ldots, 2^x \leq n$

- Το x εκφράζει πόσες φορές εκτελούμε το while loop

$$2^x \leq n \Rightarrow \log_2 2^x \leq \log_2 n \Rightarrow x \leq \log_2 n$$

Binary Search $\in O(\log_2 n)$

Πολυπλοκότητα Αναδρομικών Διαδικασιών

- Μέχρι τώρα συζητήσαμε τεχνικές για την **ανάλυση επαναληπτικών αλγορίθμων** (με while, for, κτλ.)
- Ωστόσο, πολλοί **αλγόριθμοι** ορίζονται **αναδρομικά** (π.χ. binary search, Fibonacci, κτλ.)
- Θέλουμε κάποια μεθοδολογία για να αναλύουμε την πολυπλοκότητα τέτοιων αναδρομικών εξισώσεων.

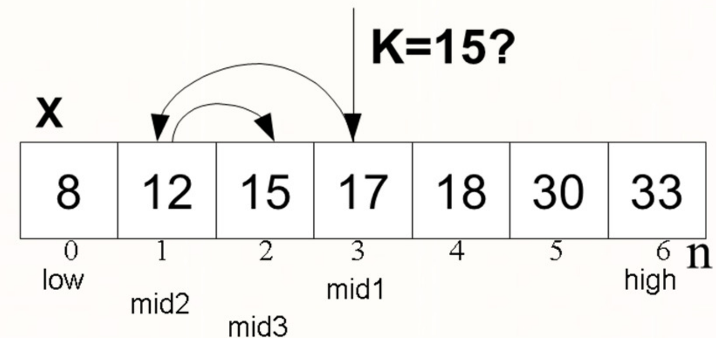
  π.χ. $T(n) = 2 \cdot T(n/2) + 1000n$

  $T(n)=2n*T(n-1), n>0$ κτλ.
- Σημειώστε ότι υπάρχουν διάφοροι τύποι αναδρομικών εξισώσεων. Πολλοί τύποι χρειάζονται ειδικά εργαλεία τα οποία δεν θα δούμε σε αυτό το μάθημα.
- Ένα τύπο που θα μελετήσουμε θα είναι οι Αναδρομικές Εξισώσεις τύπου «Διαίρει και Βασίλευε»
- Θα τις επιλύσουμε με την **Μέθοδο της Αντικατάστασης** και θα τις επαληθεύουμε με το **Θεώρημα Master**

Ανάλυση Αναδρομικής Δυαδικής Διερεύνησης

- Ας ξαναδούμε την αναδρομική έκδοση της δυαδικής αναζήτησης

```
int rbinary_aux (int X[], int low, int high, int key) {
  int mid;
  if( low <= high) {
    mid = (low + high) / 2;
    if( key == X[mid])
      return mid;
    else if( key < X[mid])
      return rbinary_aux( X, low, mid-1, key);
    else
      return rbinary_aux( X, mid+1, high, key);
  }
  return -1;
}
```



- Από ότι βλέπουμε σε κάθε εκτέλεση το binary_search μοιράζει μια ακολουθία n στοιχείων σε n/2 στοιχεία (εάν n είναι ζυγός).
- Επομένως το πρόβλημα μεγέθους n έγινε τώρα n/2.
- Σε κάθε βήμα χρειαζόμαστε και δυο συγκρίσεις (τα δυο if statements)
- Ο χρόνος εκτέλεσης της binary_search εκφράζεται με την αναδρομική συνάρτηση:

$f(n) = f(n/2) + 2$ // αναδρομικό βήμα

$f(1) = 2$ // συνθήκη τερματισμού

Μέθοδος της Αντικατάστασης

Μέθοδος της Αντικατάστασης: Χρησιμοποιούμε το βήμα της αναδρομής επαναληπτικά, μέχρι να εκφράσουμε το T(n) ως συνάρτηση της βασικής περίπτωσης, δυνάμεις του n και σταθερές τιμές.

Εφαρμογή

Έχουμε την αναδρομική εξίσωση της δυαδικής διερεύνησης

(τύπου Διαίρει και Βασίλευε)

$T(n) = T(n/2) + 2,$ για κάθε $n \geq 2$

$T(1) = 2$

Τότε, αντικαθιστώντας το T(n/2) με την τιμή του παίρνουμε

$$
\begin{aligned}
T(n) &= T(n/2) + 2 \\
&= T(n/4) + 2 + 2 \\
&= T(n/8) + 2 + 2 + 2 \\
&= \ldots \text{ (Μπορούμε τώρα να μαντέψουμε ότι)} \\
&= \underbrace{2 + \ldots + 2 + 2 + 2}_{\log_2 n}
\end{aligned}
$$

Επομένως η δυαδική αναζήτηση εκτελείται $\log_2 n$ βήματα

Ανάλυση Αναδρομικής Δυαδικής Διερεύνησης

- Στην δυαδική διερεύνηση η ακολουθία μοιράζεται ως εξής:
  n, n/2, n/4, …, 2, 1.
- **Προσοχή:** <u>Δεν σημαίνει</u> ότι έχουμε n+n/2+ n/4+ …+ 2+ 1=2n-1 εκτελέσεις. Έχουμε μονάχα $\log_2 n$ εκτελέσεις
- Ανάλογα με το σε πόσα κομμάτια **«διαιρείται»** το πρόβλημα κάθε φορά, αλλάζει και η βάση του λογάριθμου.

n	$\log_2 n$	$\log_3 n$
$2^0 = 1$	0	0
$2^1 = 2$	1	0.630929754
3	1.584962501	1
$2^2 = 4$	2	1.261859507
5	2.321928095	1.464973521
6	2.584962501	1.630929754
7	2.807354922	1.771243749
$2^3 = 8$	3	1.892789261
9	3.169925001	2
10	3.321928095	2.095903274
$2^9 = 512$	9	5.678367782
$2^{10} = 1024$	10	6.309297536
$2^{11} = 2048$	11	6.940227289
$2^{30} = 1,073,741,824$	30	18.92789261

Αριθμός πράξεων

$$\underbrace{n, n/2, n/4, \ldots, 2, 1}_{\Sigma\tau o\iota\chi\varepsilon\acute{\iota}\alpha: \log_2 n}$$

$$\underbrace{n, n/3, n/9, \ldots, 3, 1}_{\Sigma\tau o\iota\chi\varepsilon\acute{\iota}\alpha: \log_3 n}$$

- Όσο μεγαλύτερη η βάση του λογάριθμου τόσο πιο λίγες εκτελέσεις του αλγόριθμου έχουμε!
- Ωστόσο, αυξάνονται οι συγκρίσεις σε κάθε εκτέλεση!
- Π.χ., δυαδική διερεύνηση: lg n εκτελέσεις, 1 έλεγχο σε κάθε βήμα.

Master Theorem

- Το **Master Theorem** μας επιτρέπει να βρίσκουμε ή να επαληθεύουμε την χρονική πολυπλοκότητα αναδρομικών εξισώσεων **τύπου διαίρει και βασίλευε.**
- **Διαίρει και Βασίλευε: πχ.** T(n) = T(n/2) + 2 **αλλά όχι** T(n)=2n*T(n-1)
- Αυτό το θεώρημα **δεν χρειάζεται** να το **απομνημονεύσετε** αλλά μπορείτε να τον χρησιμοποιήσετε για την **επαλήθευση ασκήσεων**.

Master Theorem: Έστω ότι η f είναι μία αύξουσα που ικανοποιεί τη λειτουργία της επανάληψης:

$$f(n) = af(n/b) + cn^d$$

όταν το $n = b^k$, όπου k είναι ένας θετικός ακέραιος, $a \geq 1$, b > 1, και c, d είναι πραγματικοί αριθμοί, $c \geq 0, d \geq 0$. Τότε ισχύει:

$$f(n) is \begin{cases} O(n^d) & if\ a < b^d \\ O(n^d log_b(n)) & if\ a = b^d \\ O(n^{log_b(a)}) & if\ a > b^d \end{cases}$$

Master Theorem - Εφαρμογή

- Ο χρόνος εκτέλεσης της δυαδικής διερεύνησης εκφράζεται με την αναδρομική συνάρτηση:

T(n) = T(n/2) + 2 // αναδρομικό βήμα
T(1) = 2 // συνθήκη τερματισμού

T(n) = (a=1) x T (n/(b=2)) + (c=2)x$n^{d=0}$

$\Rightarrow$ a=1, b=2, c=2, d=0

$\Rightarrow$ a=1 και $b^d=2^0= 1$

$\Rightarrow$ $a=b^d$

$\Rightarrow$ T(n) is $O(n^d \log n)$

$\Rightarrow$ T(n) is $O(n^0 \log n)$

$\Rightarrow$ T(n) is O(logn)

Master Theorem: Έστω ότι η f είναι μία αύξουσα που ικανοποιεί τη λειτουργία της επανάληψης:

$$f(n) = af(n/b) + cn^d$$

όταν το $n = b^k$, όπου k είναι ένας θετικός ακέραιος, $a \geq 1$, $b > 1$, και c, d είναι πραγματικοί αριθμοί, $c \geq 0, d \geq 0$. Τότε ισχύει:

$$f(n) is \begin{cases} O(n^d) & if\ a < b^d \\ O(n^d log_b(n)) & if\ a = b^d \\ O(n^{log_b(a)}) & if\ a > b^d \end{cases}$$

Μέθοδος της αντικατάστασης: Παράδειγμα 1

Έχουμε την αναδρομική εξίσωση

T(n) = 4·T(n/2) + n, για κάθε n≥2

T(1) = 1

a=4, b=2, c=1, d=1, a=4>b^d=2^1=2

$$f(n) is \begin{cases} O(n^d) & if\ a < b^d \\ O(n^d log_b(n)) & if\ a = b^d \\ O(n^{log_b(a)}) & if\ a > b^d \end{cases}$$

T(n)∈O($n^{\log 2(4)}$) ➔ T(n)∈O(n^2)

Τότε, αντικαθιστώντας το T(n/2) με την τιμή του παίρνουμε

T(n) = 4·T(n/2) + n	// Εκτέλεση 1
= 4(4·T(n/4) + n/2) + n	// Εκτέλεση 2
= 4^2·T(n/4) + 2n + n	// Πράξεις
= 4^3·T(n/8) + 2^2 n + 2n + n	// Εκτέλεση 3
= ...	
= 4^k·T(1) +2^{k-1}n + ...+2^2 n + 2n + n	//2^k=n ➔ k=$\log_2 n$

$$= 4^{\log_2 n} + n * \sum_{i=0}^{\log_2 n - 1} 2^i = 2^{\log_2 n^2} + n * (2^{\log_2 n} - 1)$$

$$= n^2 + n(n-1) = 2n^2 - n \in O(n^2)$$

Μέθοδος της αντικατάστασης: Παράδειγμα 2

Άσκηση

Να λύσετε την πιο κάτω αναδρομική εξίσωση με την **μέθοδο της αντικατάστασης (προσοχή δεν είναι τύπου διαίρει & βασίλευε)**

$T(0)=1$

$T(n)=2n*T(n-1), n>0$

Λύση

$$
\begin{aligned}
T(n) &= 2*n*T(n-1) \\
&= 2^{2}*n*(n-1)*T(n-2) \\
&= 2^{3}*n*(n-1)*(n-2)*T(n-3) \\
&= \ldots \\
&= 2^{n}*n*(n-1)*(n-2)*\ldots*2*1*T(0) \\
&= 2^{n}*n! \quad \varepsilon\ O(2^{n}n!)
\end{aligned}
$$

Εργασία 1

- Να λύσετε την πιο κάτω αναδρομική εξίσωση:

$T(1) = 1$

$T(n) = 2 \cdot T(n/2) + 1000n$

Εργασία 2

- Να λύσετε την πιο κάτω αναδρομική εξίσωση:

$T(1) = 1$

$T(n) = 7 \cdot T(n/2) + 18n^2$

Εργασία 3

- Να υπολογίσετε τον χρόνο εκτέλεσης της παρακάτω αναδρομικής διαδικασίας λύνοντας οποιαδήποτε αναδρομική εξίσωση συναντήσετε.

```
public static int recursive1(int n){
int sum=0;
   for ( int i=1; i <= n; i++)
         sum++;
   if (n>1)  return recursive1(n/2);
   else return 1;
}
```

Εργασία 4

- Να υπολογίσετε τον χρόνο εκτέλεσης της παρακάτω αναδρομικής διαδικασίας λύνοντας οποιαδήποτε αναδρομική εξίσωση συναντήσετε.

```
public static int recursive2(int n){
int sum=0;
   for ( int i=1; i <= n; i++)
        sum++;
   if (n>1)  return (recursive2(n/2) +
                       recursive2(n/2));
   else return 1;
  }
```

Περίληψη των όσων έχουμε πει

- Μας ενδιαφέρει να αναλύουμε την πολυπλοκότητα των αλγορίθμων που φτιάχνουμε.
- Έχουμε δύο επιλογές:
  1. Αν υλοποιήσουμε τον αλγόριθμο και να τρέξουμε πολλά παραδείγματα έτσι ώστε να αξιολογήσουμε την απόδοση του αλγόριθμού μας.
  2. Να εφαρμόσουμε τη θεωρητική προσέγγιση ανάλυσης αλγορίθμων.

– Τι κερδίζουμε επιλέγοντας το (2) πιο πάνω;

A. Αποφεύγουμε την υλοποίηση

B. Προφέρουμε απάντηση η οποία δεν βασίζεται στην υλοποίηση που έχουμε κάνει και κατά συνέπεια δεν βασίζεται στον υπολογιστή στον οποίο τρέξαμε τα πειράματα, στη γλώσσα που χρησιμοποιήσαμε για την υλοποίηση ή στις προγραμματιστικές ικανότητες του προγραμματιστή.

C. Επιπλέον: Το αποτέλεσμα μας καλύπτει πληροφορίες για όλα τα πιθανά στιγμιότυπα εισόδου.

Περίληψη των όσων έχουμε πει

- Πως μπορούμε να υποσχόμαστε τόσα πολλά;

- Επειδή βασιζόμενοι στην Αρχή της Σταθερότητας αγνοούμε διάφορες σταθερές που συναντούμε:
  - Υποθέτουμε ότι όλες οι πράξεις ενός αλγόριθμου χρειάζονται τον ίδιο χρόνο για να εκτελεστούν παρόλο που διαφέρουν στον χρόνο εκτέλεσης τους