Διάλεξη 12: Παραδείγματα Ανάλυσης Πολυπλοκότητας / Ανάλυση Αναδρομικών Αλγόριθμων

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

- *Παραδείγματα Ανάλυσης Πολυπλοκότητας : Μέθοδοι, 6 παραδείγματα*
- *Γραμμική και Δυαδική Αναζήτηση, Ανάλυση Αναδρομικών Αλγορίθμων*

Διδάσκων: Δημήτρης Ζεϊναλιπούρ

Υπολογισμός Χρόνου Εκτέλεσης με **φωλιασμένους βρόχους** και συνθήκες **if**

- 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**

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

Φωλιασμένος Βρόχος

- **Συνεχόμενες Εντολές:** Ο χρόνος εκτέλεσης της εντολής **`S και μετά S΄`** παίρνει χρόνο ίσο του αθροίσματος των χρόνων εκτέλεσης των **`S + S΄.`**
- **Συνθήκες if:** Ο χρόνος εκτέλεσης 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

```
2.    int i, j, sum=0;
      for (i=0; i<n; i++)
          for (j=0; j<n; j++)
              sum++;
```

Ανάλυση

Εσωτερικός βρόχος: $\sum_{j \in n} 1 = n$

Εξωτερικός βρόχος: $\sum_{i \in n} 1 = n$

Συνολικά: $\sum_{i \in n} \sum_{j \in n} 1 = n * n = n^2 \in \Theta(n^2)$

Παράδειγμα 3: Υπολογισμός Χρόνου Εκτέλεσης

```
3.  int i, j, sum=0;
    for (i=0; i<n; i++)
       for (j=0; j<i*i; j++)
            sum++;
```

Για i, δείχνει πόσες φορές εκτελείται το j

i	j
0	0
1	1
2	4
3	9
….	…
n-1	$(n-1)^2$

Συνολικά; $\sum_{i \in n} i^2$

Ανάλυση

Εσωτερικός βρόχος: $B1 = \sum_{j \in i^2} 1 = i^2$

Εξωτερικός βρόχος: Εκτελείται n φορές

Συνολικά: $\sum_{i \in n} B1$ **δηλ.,** $\sum_{i \in n} i^2 = \frac{n \cdot (n+1)(2n+1)}{6}$

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

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

Παράδειγμα 4: Υπολογισμός Χρόνου Εκτέλεσης

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

Ανάλυση

Βρόχος 1 (B1): $\sum_{k \in n} 1 = n$

Βρόχος 2 (B2): $\sum_{j \in n} B1 = \sum_{j \in n} n = n * n$

Εξωτερικός βρόχος:

$$\sum_{i \in n} B2 = \sum_{i \in n} n^2 = n * n^2 = n^3$$

Συνολικά:

$$\varepsilon\, \Theta\, (n^3)$$

Παράδειγμα 5: Υπολογισμός Χρόνου Εκτέλεσης

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

Για i, δείχνει πόσες φορές εκτελείται το j

i	j
1	n
2	n
4	n
8	n
….	…

Συνολικά $\log_2 n$ βήματα

Ανάλυση

Εσωτερικός βρόχος: $\sum_{j \in n} 1 = n$

Εξωτερικός βρόχος: 1, 2, 4, 8, …, $2^i \leq n$

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

Επομένως το i εκτελείται τουλαχιστο $\log_2 n$ φορές (Θα υποθέτουμε ότι n είναι δύναμη του 2)

Συνολικά: $\sum_{i \in \log_2 n} n \in \Theta(n * \log_2 n)$

Παράδειγμα 6: Υπολογισμός Χρόνου Εκτέλεσης (Με συνθήκη if)

```
int i, j, sum=0;
for (i=0; i<n; i++)
     if ((n % 2) == 0)          // n άρτιος
        for (j=0; j<n; j++)
              sum++
     else                                  // n περιττός
              sum--;
```

Ανάλυση (Βασική πράξη πρόσθεση/αφαίρεση)

α) N περιττός:

$$\sum_{i \in n} 1 = n \in \Theta(n)$$

β) N άρτιος:

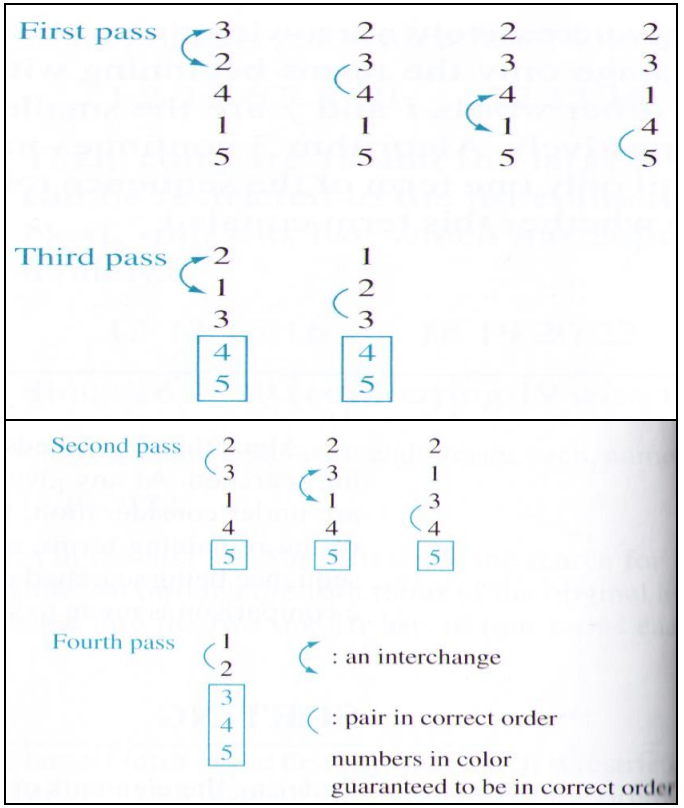
$$\sum_{i \in n} n = n^2 \in \Theta(n^2)$$

Παρατηρούμε ότι δεν καταλήγουμε στο MAX(odd, even), εφόσον πάντα θα εκτελείται μόνο ένα από τα δυο σκέλη της συνθήκης.

Εάν η συνθήκη ήταν **`((i % 2) == 0)`** τότε θα μπορούσαμε να πούμε ότι έχουμε O(n) χρόνο.

Παράδειγμα 7: Χρόνος Εκτέλεσης Bubblesort

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



Ανάλυση

Εσωτερικός βρόχος: $\sum_{i\in(n-i-1)} 1 = n-i-1$

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

Συνολικά:

$$\sum_{i\in(n-1)} n-i-1 = \sum_{i\in(n-1)} (n-1) - \sum_{i\in(n-1)} i = (n-1)(n-1) - \sum_{i\in(n-1)} i$$

$$= (n-1)(n-1) - \frac{(n-1)n}{2} = \frac{2(n-1)(n-1)-(n-1)n}{2} =$$

$$= \frac{(n-1)(2n-2-n)}{2} = \frac{(n-1)(n-2)}{2} \in \Theta(n^2)$$

Γραμμική Vs Δυαδική Διερεύνηση (Ανάλυση)

- *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;
}
```

X

8	12	15	17	18	30	33

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

Αναδρομική Γραμμική Διερεύνηση

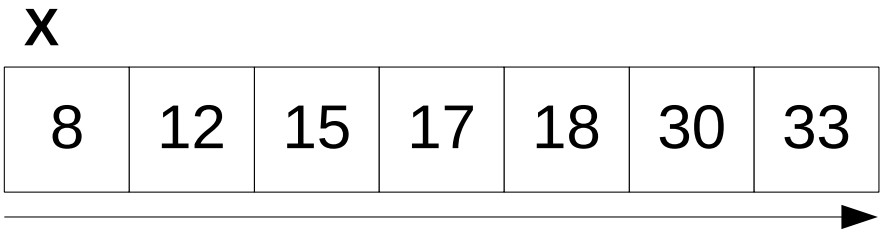
```
int rlinear( int X[], int n, int k, int pos){

   if (pos == n)  return -1;  // not found

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

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

}
```

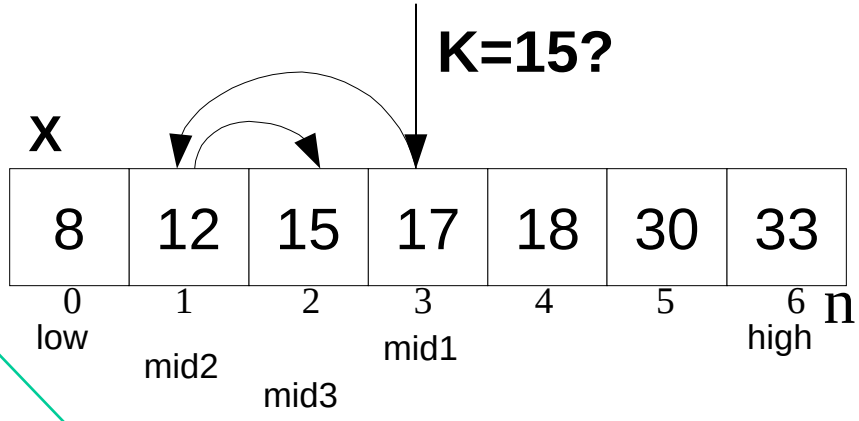


- *Χείριστη περίπτωση*: **O(n)** (εκτελούνται n αναδρομικές κλήσεις της rlinear)

Δυαδική Διερεύνηση

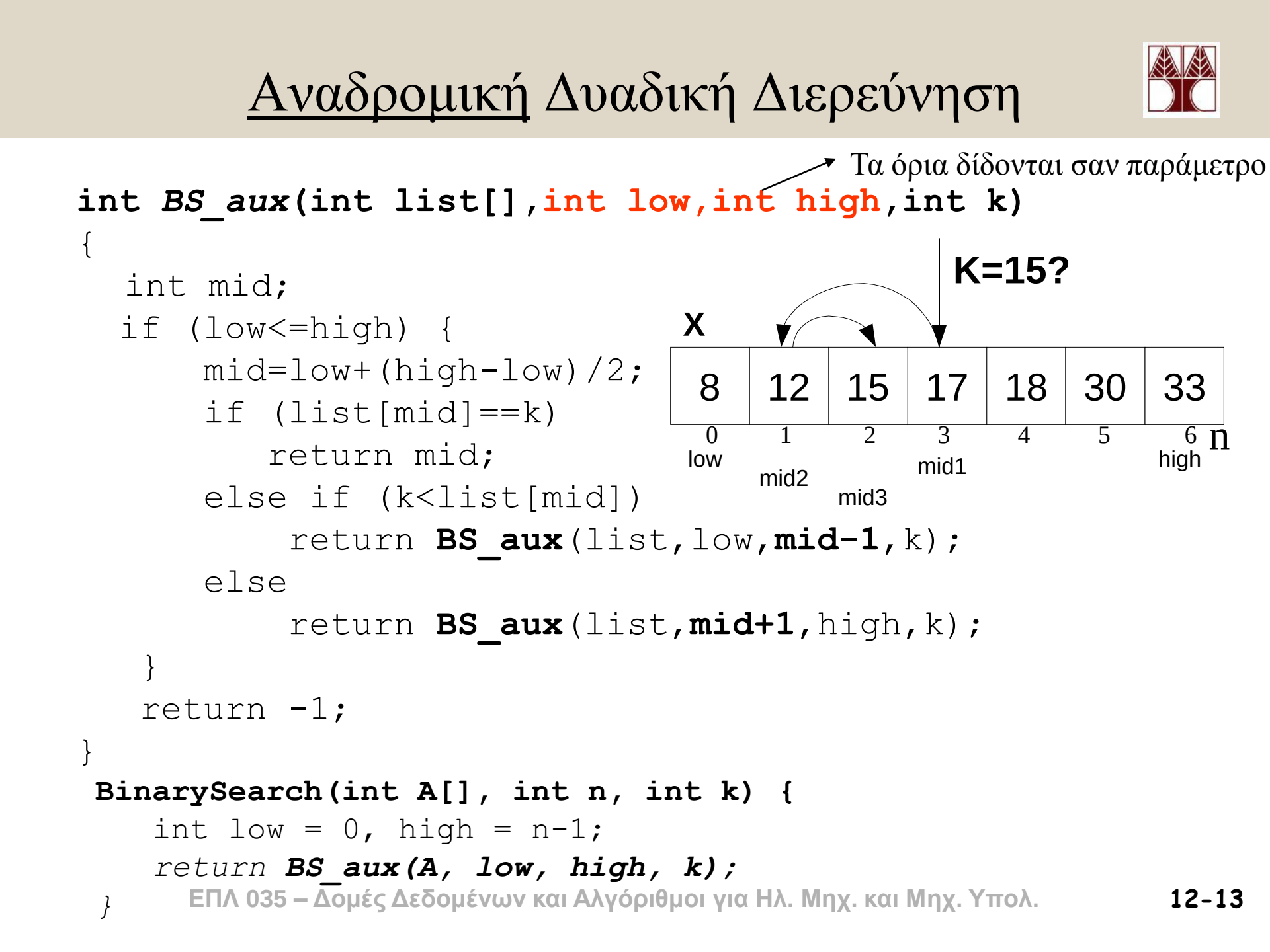
- ***Δυαδική Διερεύνηση:*** βρίσκουμε το μέσο του πίνακα και αποφασίζουμε αν το k ανήκει στο δεξιό ή το αριστερό μισό. Επαναλαμβάνουμε την ίδια διαδικασία στο "μισό" που μας ενδιαφέρει.

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



Ίδιο με (low+high)/2. Ωστόσο η έκδοση αυτή μπορεί να υποφέρει από υπερχείλιση ακεραίου.

Αναδρομική Δυαδική Διερεύνηση



Τα όρια δίδονται σαν παράμετρο

```
int BS_aux(int list[],int low,int high,int k)
{
   int mid;
   if (low<=high) {
        mid=low+(high-low)/2;
        if (list[mid]==k)
            return mid;
        else if (k<list[mid])
             return BS_aux(list,low,mid-1,k);
        else
             return BS_aux(list,mid+1,high,k);
    }
    return -1;
}
```



```
BinarySearch(int A[], int n, int k) {
    int low = 0, high = n-1;
    return BS_aux(A, low, high, k);
}
```

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 ε $O(\log_2 n)$

* Χωρίς βλάβη της γενικότητας, θεωρήστε ότι n είναι ζυγός

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, etc)
- Θέλουμε κάποια μεθοδολογία για να αναλύουμε την πολυπλοκότητα τέτοιων αναδρομικών εξισώσεων.

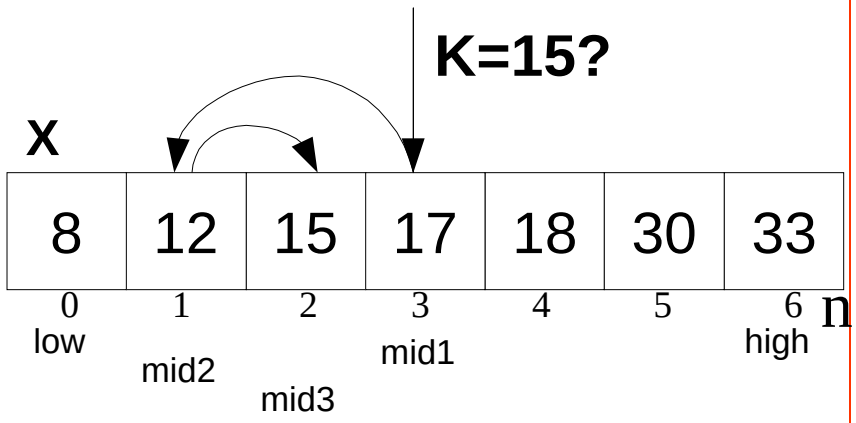
  π.χ. $T(n) = 2 \cdot T(n/2) + 1000n$

  $T(n)=2n*T(n-1), n>0$ 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 BS_aux(int list[],int low,int high,int k)
{
  int mid;
  if (low<=high) {
      mid=low+(high-low)/2;
      if (list[mid]==a)
       return mid;
      else if (k<list[mid])
          return BS_aux(list,low,mid-1,k);
      else
          return BS_aux(list,mid+1,high,k);
   }
   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 Τ(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 Τ(n/2) με την τιμή του παίρνουμε

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

Επομένως η δυαδική αναζήτηση εκτελείται $\log_2 n$ βήματα

Binary Search ε $O(\log_2 n)$

Ανάλυση Αναδρομικής Δυαδικής Διερεύνησης

- Στην δυαδική διερεύνηση η ακολουθία μοιράζεται ως εξής:
  n, n/2, n/4, …, 2, 1.
- **Προσοχή:** Δεν σημαίνει 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

$\underbrace{n, n/2, n/4, \ldots, 2, 1}_{\Sigma\tau οιχεία:\log_2 n}$

$\underbrace{n, n/3, n/9, \ldots, 3, 1}_{\Sigma\tau οιχεία:\log_3 n}$

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

Ανάλυση Αναδρομικής Δυαδικής Διερεύνησης

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 τον **απομνημονεύσετε** αλλά μπορείτε να τον χρησιμοποιήσετε για την **επαλήθευση ασκήσεων**.

MASTER THEOREM Let f be an increasing function that satisfies the recurrence relation

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

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

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

Master Theorem - Εφαρμογή

Αναδρομική Δυαδική Αναζήτηση

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

$T(1) = 2$

a=1, b=2, c=2, d=0 (δες τύπο Master)

$\Rightarrow$ a=1 και $b^d = 2^0 = 1$

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

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

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

MASTER THEOREM Let f be an increasing function that satisfies the recurrence relation

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

whenever $n = b^k$, where k is a positive integer, $a \geq 1$, b is an integer greater than 1, and c and d are real numbers with c positive and d nonnegative. Then

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

Παράδειγμα 2 - Μέθοδος της αντικατάστασης

Έχουμε την αναδρομική εξίσωση

$T(n) = 4 \cdot T(n/2) + n$, για κάθε $n \geq 2$

$T(1) = 1$

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

$T(n) = 4 \cdot T(n/2) + n$ // Εκτέλεση 1

$= 4(4 \cdot T(n/4) + n/2) + n$ // Εκτέλεση 2

$= 4^2 \cdot T(n/4) + 2n + n$ // Πράξεις

$= 4^3 \cdot T(n/8) + 2^2 n + 2n + n$ // Εκτέλεση 3

= ... Μπορούμε τώρα να μαντέψουμε ότι …

$= 4^k \cdot T(1) + 2^{k-1} n + \ldots + 2^2 n + 2n + n$ // $k = \log_2 n$

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

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

Παράδειγμα 3 - Μέθοδος της αντικατάστασης

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

$T(n) = 2*n*T(n-1)$

$= 2^{2*}n*(n-1)*T(n-2)$

$= 2^{3*}n*(n-1)*(n-2)*T(n-3)$

….(Μπορούμε τώρα να μαντέψουμε ότι)

$= 2^{n}*n*(n-1)*(n-2)*...*2*1*T(0)$

$= 2^{n}*n!$ $\quad \varepsilon\ O(2^{n}n!)$