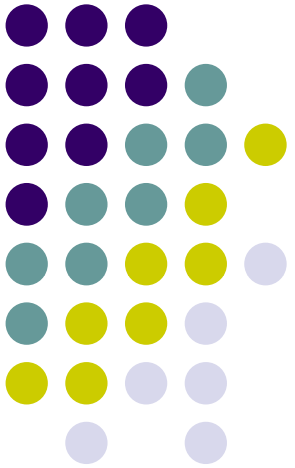
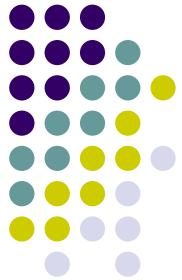
Polytechnic School, University of Patras, CEID

Searching and Sorting

S.Sioutas, 7/10/2018

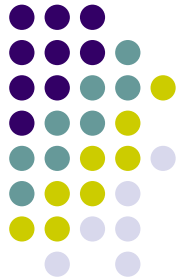


Introduction



- Task of searching for a key in a list or in an array
- Searching algorithms will be discussed:
  - Linear Search
  - Binary Search
  - Interpolation Search

Linear Search

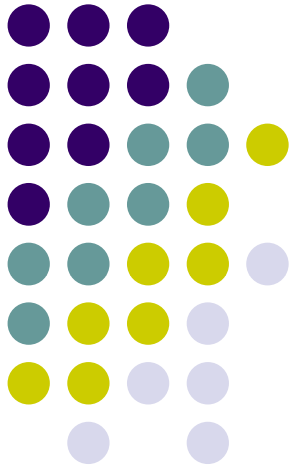


- Or called Sequential Search
- checking every element of a list sequentially until a match is found

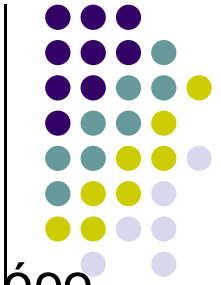
- Complexity: O(N)
- On average, N/2 comparisons is needed
- Best case: the first element searched is the value we want
- Worst case: the last element searched is the value we want

Linear or Sequential Searching

```
int sequential(key)
1.      i = 0;
2.      while (i < n) do
3.              if A[i] = key then return i;
4.              else i = i + 1;
5.      return -1;
```



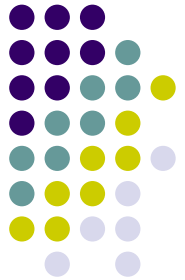
Linear Searching - Complexity



- Στην μέση περίπτωση, θα πρέπει να υπολογίσουμε κατά μέσο όρο, πόσες φορές θα εκτελεστεί ο βρόχος, καθώς είναι το μόνο σημείο που επιβαρύνει σημαντικά την πολυπλοκότητα του αλγορίθμου
- Η βασική ιδέα από την οποία ξεκινάμε, είναι όπως αναφέραμε και προηγουμένως, ότι η πιθανότητα να βρεθεί ένα στοιχείο σε κάποια θέση του πίνακα είνα 1/n, όπου n το πλήθος των στοιχείων του πίνακα. Επιπλέον, στην περίπτωση που το στοιχείο βρίσκεται στη θέση 1, ο βρόχος θα εκτελεστεί 1 φορά, στην περίπτωση που το στοιχείο βρίσκεται στη θέση 2, ο βρόχος θα εκτελεστεί 2 φορές κ.ο.κ. Επομένως, ο βρόχος θα εκτελεστεί στη μέση περίπτωση:

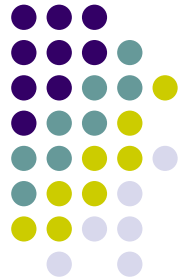
$$E = \frac{1 + 2 + 3 + \cdots + n}{n} = \frac{n + 1}{2}$$

Binary Search



- Requirement: the list of data is sorted
- The idea is based on the elimination of impossible region

Binary Search



- Algorithm:

Let the current region is from A[low] to A[high]

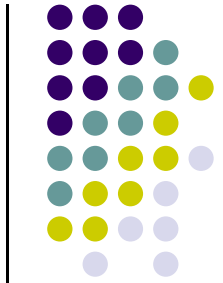
Compare key with A[midpoint]

If key<A[midpoint], then A[midpoint] to A[high] does not contain the key

If key>A[midpoint], then A[low] to A[midpoint] does not contain the key

Repeat the above process until the key is found

Binary Search



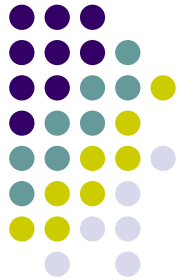
int binary_iterate(key)

```
1.  bottom = 0; top = n - 1;
2.  while (bottom <= top) do
3.          middle = (top + bottom) div 2;
4.          if A[middle] = key then return middle;
5.          else if A[middle] > key then top = middle - 1;
6.          else bottom = middle + 1;
7.  return -1;
```

Int binary_rec(key, left, right)

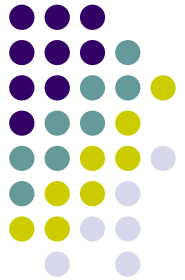
```
1.  if left > right then return -1;
2.  middle = (top + bottom) div 2;
3.  if A[middle] = key then return middle;
4.  else if A[middle] > key then
5       binary_rec(key, left, middle - 1);
6.  else binary_rec(key, middle + 1, right);
```

Binary Search



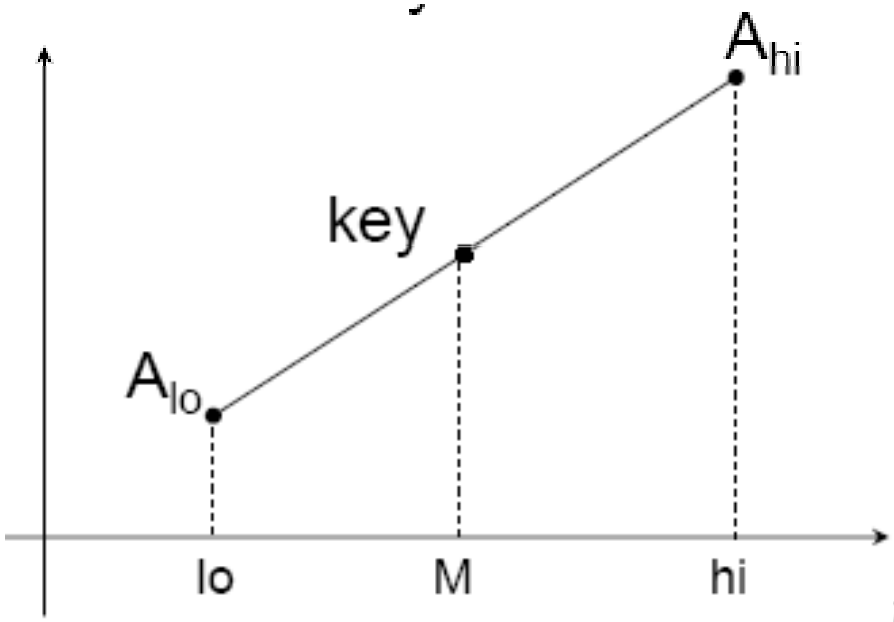
- Complexity: O(log N)

Interpolation Search



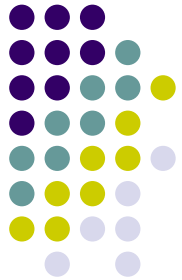
- Requirement: the list of data is sorted

Interpolation Step



$$M \approx lo + \frac{hi - lo}{A_{hi} - A_{lo}} \times (key - A_{lo})$$

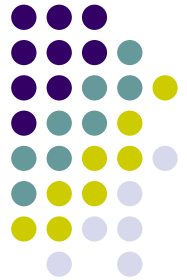
Interpolation Search



Int interpolation(key)

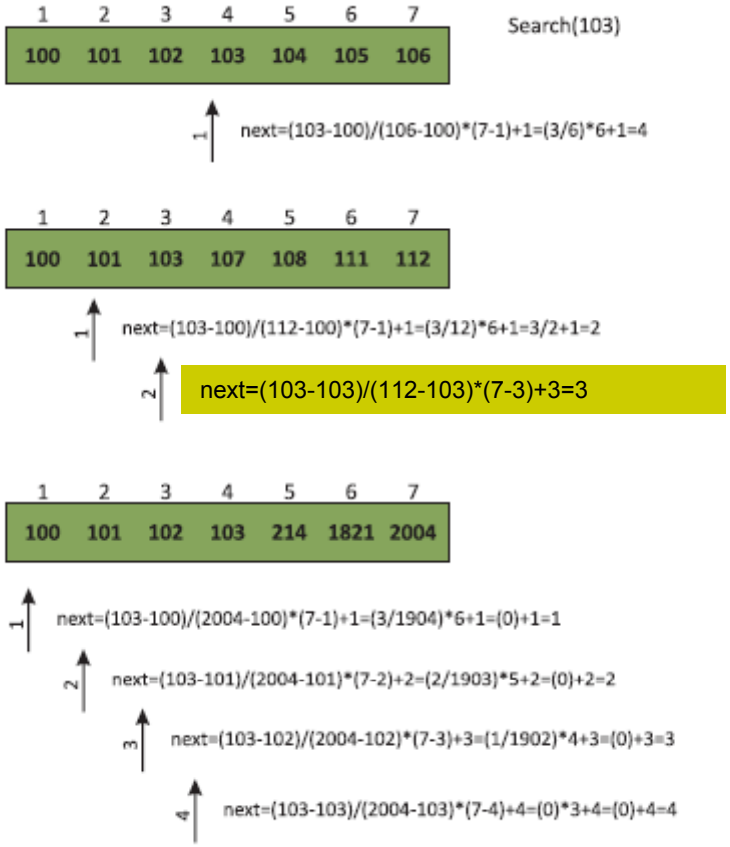
1. low = 0; high = n - 1;
2. while (A[high] >= key) and (key > A[low]) do
3. next = low + [(key-A[low])/(A[high]-A[low])]*(high-low);
4. if key > A[next] then low = next+1;
5. else if key < A[next] then high = next-1;
6. else low = next;
7. if key == A[low] then return low; else return -1;

Interpolation Searching (simple)



- Complexity: O(loglogN) expected for Uniform Distributions
- O(N) in worst-case

Example of Interpolation Searching



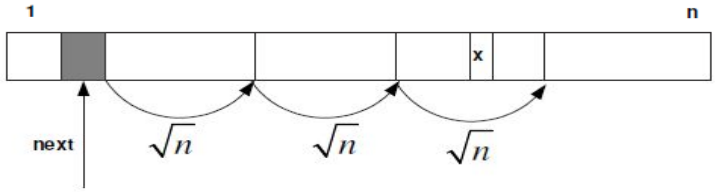
Εικόνα 3.1

Αναζήτηση παρεμβολής

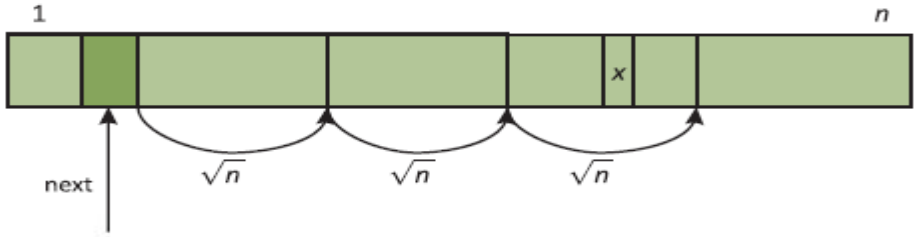
Binary Interpolation Searching

Η **δυαδική αναζήτηση παρεμβολής**, λειτουργεί όπως και η αναζήτηση παρεμβολής με τη διαφορά ότι όταν φτάσουμε σε πολύ μικρά διαστήματα αναζήτησης, τότε εφαρμόζουμε δυαδική αναζήτηση. Ακολουθεί ψευδοκώδικας που υλοποιεί τη μέθοδο:

```
   procedure interpolation_binary(key)
1. left = 1; right = n; size = right-left+1;
2. next = ⌈size * (key - A[left]) / (A[right] - A[left])⌉;
3. while (key <> A[next]) do
4. i = 0; size = right – left + 1;
5. if (size <= 3) {Απευθείας αναζήτηση}
6. if (key >= A[next])
7. while (key > A[next + i * ⌊sqrt(size)⌋ - 1] do
8. i = i + 1;
9. right = next + i * ⌊sqrt(size)⌋;
10.left = next + (i - 1) * ⌊sqrt(size)⌋;
11.else if (key < A[next])
12.while (key < A[next – i * ⌊sqrt(size)⌋ + 1] do
13.i = i + 1;
14.right = next - (i - 1) * ⌊sqrt(size)⌋;
15.left = next – i * ⌊sqrt(size)⌋;
16.next = left + ⌈((right-left+1) * (key-A[left]) /
   (A[right]-A[left]))⌉;
17.if (key = A[next]) then return next;
18.else return -1;
```

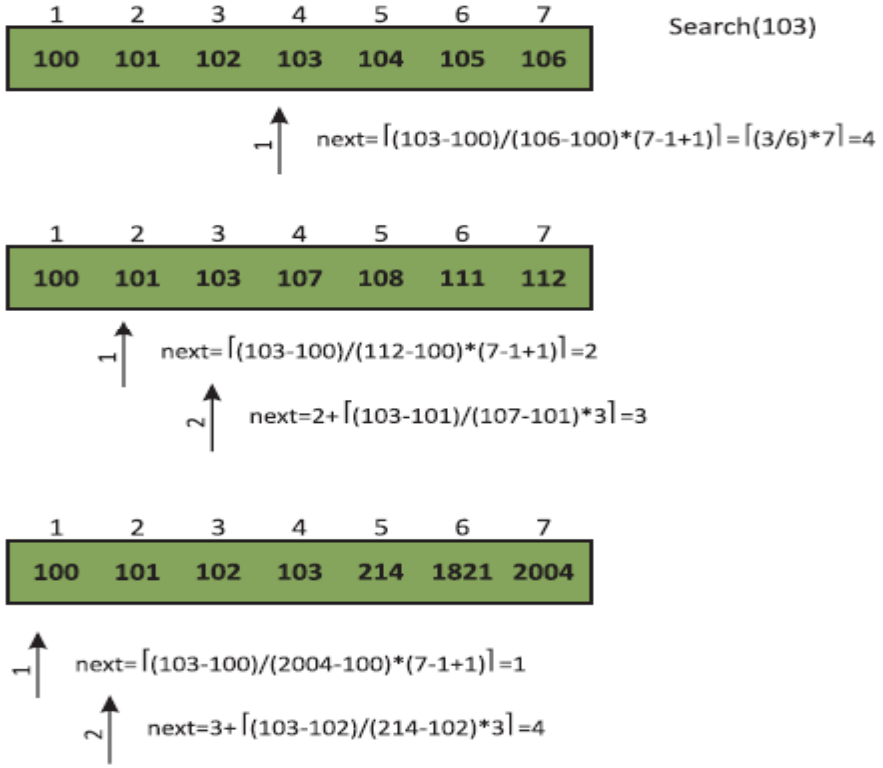


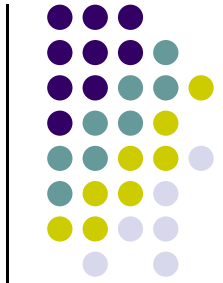
Binary Interpolation Searching Example



Εικόνα 3.2

Αναζήτηση δυαδικής παρεμβολής





Expected Time Complexity : O(loglogn)

Worst-case Time Complexity : $O(\sqrt{n})$

• Για την πολυπλοκότητα της μέσης περίπτωσης, υποθέτουμε ότι όλα τα στοιχεία του πίνακα Α επιλέγονται ισοπίθανα από ένα διάστημα τιμών. Θεωρούμε ότι το πλήθος των συγκρίσεων που χρειάζονται μέχρι να φτάσουμε στο υποδιάστημα μεγέθους $\sqrt{n}$ που θα περιέχει το στοιχείο key, είναι C.

• Συμβολίζουμε με p_i την πιθανότητα να πραγματοποιηθούν i συγκρίσεις, μέχρι να υπολογίσουμε το ζητούμενο διάστημα.

• Με βάση όσα αναφέραμε το πλήθος των συγκρίσεων θα είναι:

$$C = \sum_{i \geq 1} i(P_i - P_{i+1}) = \sum_{i \geq 1} P_i$$

Αν ο πίνακας είχε μόνο ένα στοιχείο, δε θα είχε νόημα να κάνουμε αναζήτηση γιατί η μία θέση του πίνακα, θα είχε ή όχι το κλειδί key. Συνεπώς θεωρούμε ότι θα χρειαστούν τουλάχιστον 2 συγκρίσεις και άρα $P_1 = P_2 = 1$. Για το P_i θα ισχύει:

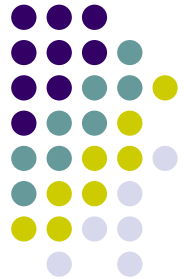
$$P_i \leq Prob\left[|keyposition - next| \geq (i-2)\sqrt{n}\right], \quad i \geq 3$$

Όπου ως keyposition θεωρούμε τη θέση του κλειδιού στον πίνακα. Η προηγούμενη σχέση, θυμίζει την ανισότητα Chebyshev:

$$Prob[|X - \mu| \geq \varepsilon] \leq \frac{\sigma^2}{\varepsilon^2}$$

Όπου X μία τυχαία μεταβλητή, μ η μέση τιμή, ε κάποια σταθερά και σ^2 η διασπορά.

Expected Time Complexity : O(loglogn)
Worst-case Time Complexity : $O(\sqrt{n})$



Η τυχαία μεταβλητή X παίρνει ως τιμές, το πλήθος των στοιχείων A_i που είναι μικρότερα του κλειδιού key που αναζητούμε (A_i < key).
Έστω q_j η πιθανότητα το στοιχείο key να ισούται με j .
Αν τα στοιχεία A_i επιλέγονται ισοπίθανα, η πιθανότητα ένα στοιχείο A_i να είναι μικρότερο του κλειδιού key , θα είναι:

$$p = \frac{key - Ao}{A_{n+1} - A_1}$$

Και η πιθανότητα q_j θα είναι: $$q_j = \binom{n}{j} p^j (1-p)^{n-j}$$

Παρατηρούμε ότι έχουμε μία ΔΥΩΝΥΜΙΚΗ ΚΑΤΑΝΟΜΗ, όπου: $$\mu = pn \qquad \sigma^2 = p(1-p)n \le \frac{n}{4}$$

Δηλαδή μ=next. Επομένως θα ισχύει:

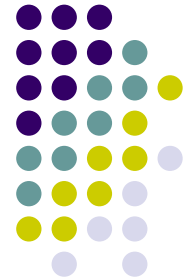
$$P_i \le Prob\left[|keyposition - next| \ge (i-2)\sqrt{n}\right] \le \frac{p(1-p)n}{\left((i-2)\sqrt{n}\right)^2} = \frac{p(1-p)}{(i-2)^2}$$

$$\le \frac{1}{4(i-2)^2}$$

Οπότε το πλήθος των συγκρίσεων φράσσεται από: $$C \le 2 + \sum_{i \ge 3} \frac{1}{4(i-2)^2} = 2 + \frac{\pi^2}{24} \approx 2.4$$

Άρα, στη μέση περίπτωση: T=2.4loglogn → που σημαίνει 3 το πολύ ΑΛΜΑΤΑ μήκους $\sqrt{n}$

Expected Time Complexity : O(loglogn)
Worst-case Time Complexity : $O(\sqrt{n})$



Αν συμβολίσουμε με $T(n)$, το μέσο πλήθος συγκρίσεων, τότε προκύπτει η εξίσωση:

$T(n) \leq C + T(\sqrt{n})$ για $n \geq 3$ και $T(1) \leq 1, T(2) \leq 2$. Συνεπώς:

$$T(n) \leq 2 + 2.4 \log \log n$$

Και άρα η πολυπλοκότητα στη μέση περίπτωση θα είναι $O(\log \log n)$.

Ας εξετάσουμε τώρα την πολυπλοκότητα για τη χειρότερη περίπτωση. Στον ψευδοκώδικα για τη δυαδική αναζήτηση παρεμβολής που παρουσιάσαμε προηγουμένως, πραγματοποιούμε γραμμικά αναζήτηση μεγέθους $\sqrt{size}$ (εντολή 12). Την πρώτη φορά, θα εκτελεστούν επομένως $\sqrt{n}$ συγκρίσεις. Τη δεύτερη φορά, θα εκτελεστούν $\sqrt{\sqrt{n}}$ συγκρίσεις κοκ. Δηλαδή, θα ισχύει:

$$n^{\frac{1}{2}} + n^{\frac{1}{4}} + n^{\frac{1}{8}} + \cdots = O(\sqrt{n})$$

Το φράγμα της ασυμπτωτικής πολυπλοκότητας προκύπτει κρατώντας τον μεγαλύτερο όρο της ακολουθίας $n^{\frac{1}{2}}$.

Βελτίωση του χειρότερου χρόνου

4.4.1 Βελτίωση του Χρόνου Χειρότερης Περίπτωσης

Ο χρόνος χειρότερης περίπτωσης μπορεί να βελτιωθεί από $O(\sqrt{n})$ σε $O(\log n)$ χωρίς να χειροτερεύει ο χρόνος μέσης περίπτωσης. Αντί, μέσα στο *while loop*, το i να αυξάνεται γραμμικά ($i \leftarrow i + 1$) αυτό να αυξάνεται εκθετικά ($i \leftarrow 2 * i$). Με αυτόν το τρόπο γίνονται ολοένα μεγαλύτερα άλματα, κρατώντας το αριστερό άκρο σταθερό (βλ. σχήμα 4.5), με αποτέλεσμα το τελευταίο υποδιάστημα να είναι πολύ πιο μεγάλο από $\sqrt{n}$. Μόλις βρεθεί αυτό το υποδιάστημα εφαρμόζουμε δυική αναζήτηση στα στοιχεία μέσα σε αυτό που απέχουν κατά $\sqrt{n}$ και έτσι προκύπτει το ζητούμενο υποδιάστημα μεγέθους $\sqrt{n}$. Πιο συγκεκριμένα:

Παραλλαγή του BIS

i) Κάνε εκθετικά μεγάλα βήματα, εξετάζοντας τα διαστήματα : $[next, next+\sqrt{n}], [next, next+2\sqrt{n}], [next, next+2^2\sqrt{n}], [next, next+2^3\sqrt{n}], ..., [next, next+2^i\sqrt{n}]$ μέχρι να βρεθεί j:

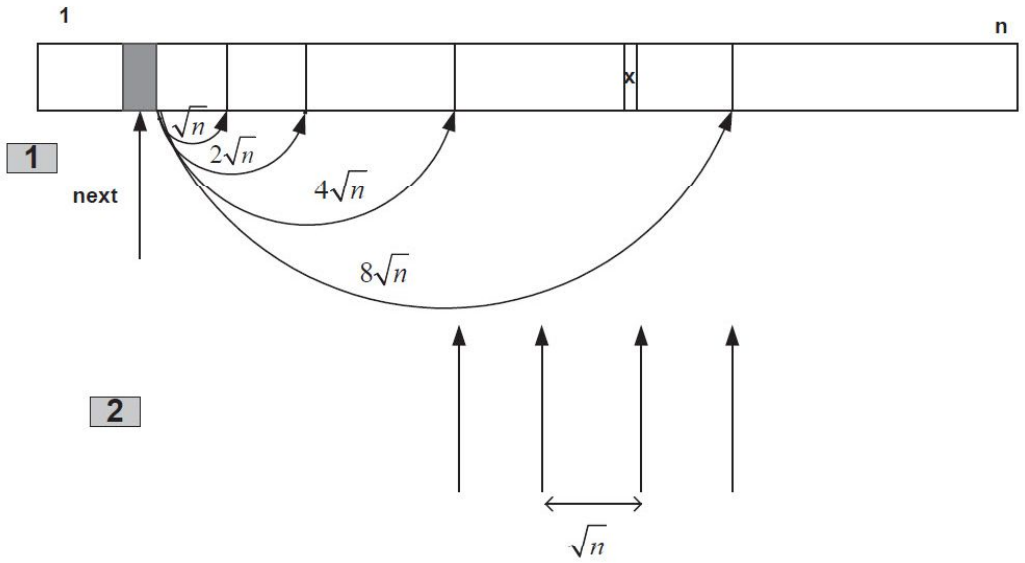
$$S[next + 2^{j-1}\sqrt{n}] < x \leq S[next + 2^j\sqrt{n}]$$

ii) Ψάξε δυικά στα στοιχεία των θέσεων

$$next + 2^{j-1}\sqrt{n}, next + [2^{j-1}+1]\sqrt{n}, next + [2^{j-1}+2]\sqrt{n}, ..., next + 2^j\sqrt{n}$$

Δηλαδή ψάξε δυικά στα στοιχεία που απέχουν απόσταση $\sqrt{n}$ στο διάστημα που βρέθηκε από το ερώτημα 1. Έτσι προκύπτει τελικά ένα διάστημα μεγέθους $\sqrt{n}$.

Παραλλαγή του BIS



Σχήμα 4.5: Βελτίωση BIS

$STEP1: 2^j \sqrt{n} = n \Leftrightarrow j = \log(\sqrt{n})$

$STEP2: \log(2^j - 2^{j-1}) = \log(2^{j-1}) = j - 1 = \log(\sqrt{n}) - 1$

Δηλαδή, j-1 βήματα για να εντοπίσω τη σωστή ρίζα-ν-άδα!!!

$Totally: 2\log(\sqrt{n}) - 1$

Παραλλαγή του BIS

Έστω ότι για το πρώτο βήμα εκτελούμε στην χειρότερη περίπτωση i συγκρίσεις. Αυτό σημαίνει ότι:

$$2^i\sqrt{n} = n \Rightarrow$$

$$2^i = \sqrt{n} \Rightarrow 2^i = \sqrt{n} \Rightarrow i = \log\sqrt{n}$$

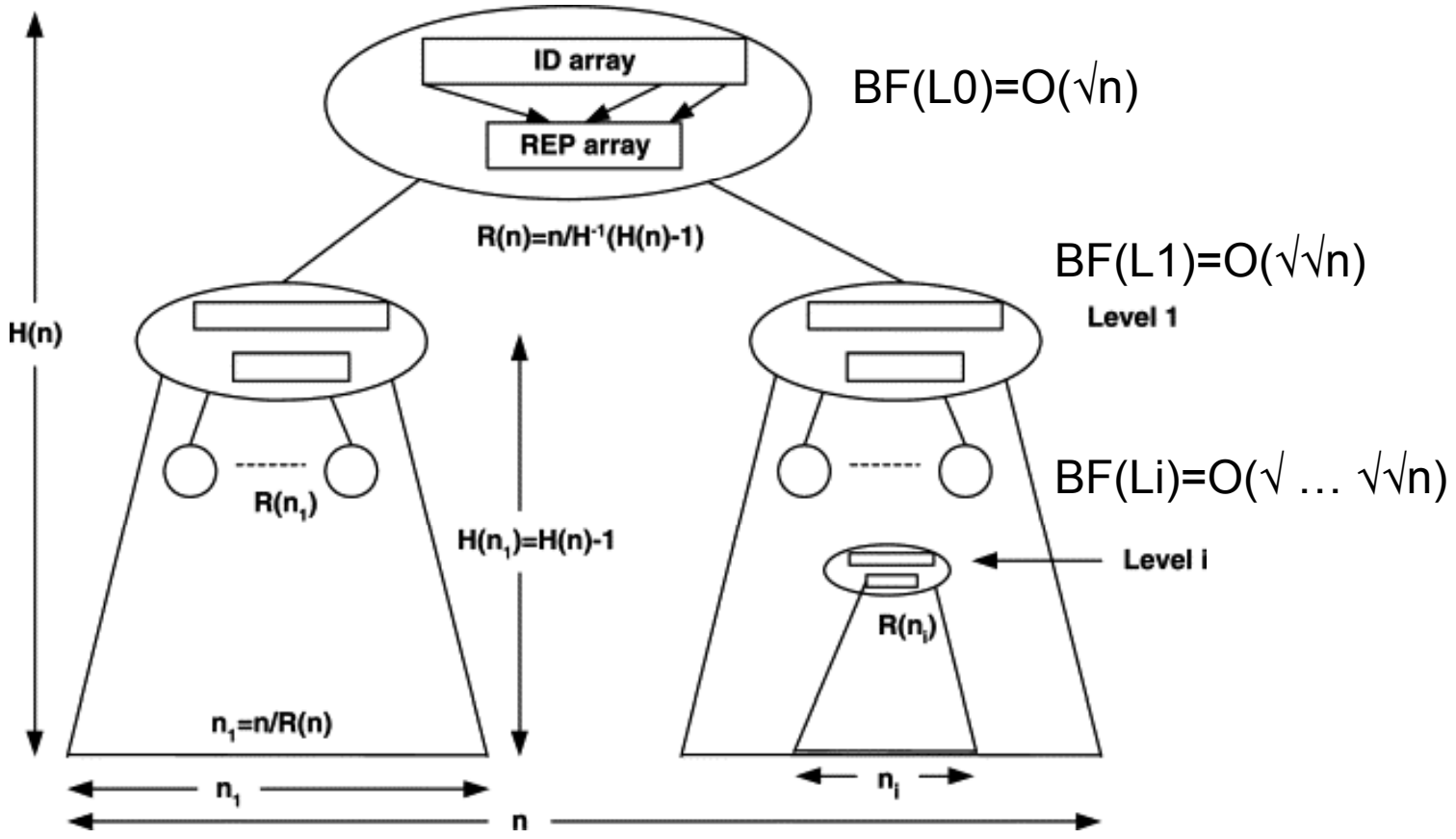
Συνεπώς για το πρώτο βήμα χρειάζεται χρόνο $O(\log\sqrt{n})$. Για το δεύτερο βήμα τότε χρειάζεται χρόνος $\log 2^{i-1} = O(i-1) = O(\log i) = O(\log\sqrt{n})$. Έτσι προκύπτει ότι κάθε επανάληψη του *while* θα χρειάζεται $O(\log i)$ χρόνο στην χειρότερη περίπτωση.

Έτσι συνολικά θα χρειασθεί χρόνος:

$$\begin{aligned}
\log\sqrt{n} + \log\sqrt{\sqrt{n}} + \log\sqrt{\sqrt{\sqrt{n}}} + \ldots &= \log n^{\frac{1}{2}} + \log n^{\frac{1}{4}} + \log n^{\frac{1}{8}} + \ldots \\
&= \frac{1}{2}\log n + \frac{1}{4}\log n + \frac{1}{8}\log n + \ldots \\
&= (\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots)\log n \\
&= O(\log n)
\end{aligned}$$

IST exponential tree

T=O(loglogn)



END