Αλγόριθμοι Αναζήτησης

Δημήτρης Φωτάκης

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Εθνικό Μετσόβιο Πολυτεχνείο

Γραμμική Αναζήτηση

- Μοναδικός τρόπος όταν:
  - είτε όχι ταξινομημένος πίνακας,
  - είτε μόνο σειριακή προσπέλαση (π.χ. αρχεία).

```
int linearSearch(int A[], int n, int x) {
    for (int i = 0; i < n; i++)
        if (x == A[i]) return(i);
    return(-1); }
```

- Χρόνος χ.π. / αποτυχημένης αναζήτησης: $\Theta(n)$ (βέλτιστος).
  Χρόνος καλύτερης περίπτωσης: $\Theta(1)$.

Γραμμική Αναζήτηση (μ.π.)

- Πιθανότητα αναζήτησης στοιχείου *k* = 1 / *n*
  - Όλα τα στοιχεία αναζητούνται ισοπίθανα!
- Χρόνος μ.π. $= \sum_{i=k}^{n} \mathbb{P}[k] \cdot k = \frac{n(n+1)}{2n} = \frac{n+1}{2}$

Γραμμική Αναζήτηση (μ.π.)

- Όχι ισοπίθανη αναζήτηση:
  - Μπροστά στοιχεία με μεγαλύτερη συχνότητα αναζήτησης.
- Δεν γνωρίζουμε συχνότητες:
  - Σταδιακή αναδιοργάνωση : μπροστά στοιχεία που ζητούνται.
  - Move-to-Front : χρόνος ≤ 2×βέλτιστος, λίστα.
  - Move-Forward : πίνακας, όχι βελτίωση αν μόνο δύο στοιχεία.
- Self-adjusting DS: «προσαρμόζεται» ώστε να είναι ταχύτερη στο μέλλον (βλ. splay trees).

0.5	0.3	0.1	0.5
2	3	1	4

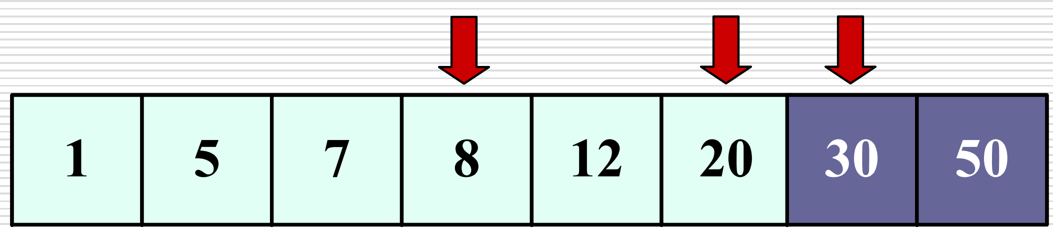
Μέσος #συγκρίσεων = 1.8

Δυαδική Αναζήτηση

- Ταξινόμηση και τυχαία προσπέλαση.
- Χρόνος O(log *n*) (βέλτιστος ως προς χ.π.).

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

Αναζήτηση **30**:

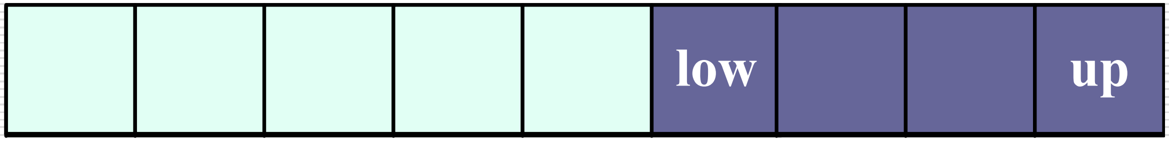


Ορθότητα

- Αν A[*mid*] > *k*, *k* μπορεί να βρίσκεται μόνο αριστερά.
- Αν A[*mid*] < *k*, *k* μπορεί να βρίσκεται μόνο δεξιά.
- Σε κάθε επανάληψη, πλήθος υποψήφιων στοιχείων μειώνεται (περίπου) στο μισό :

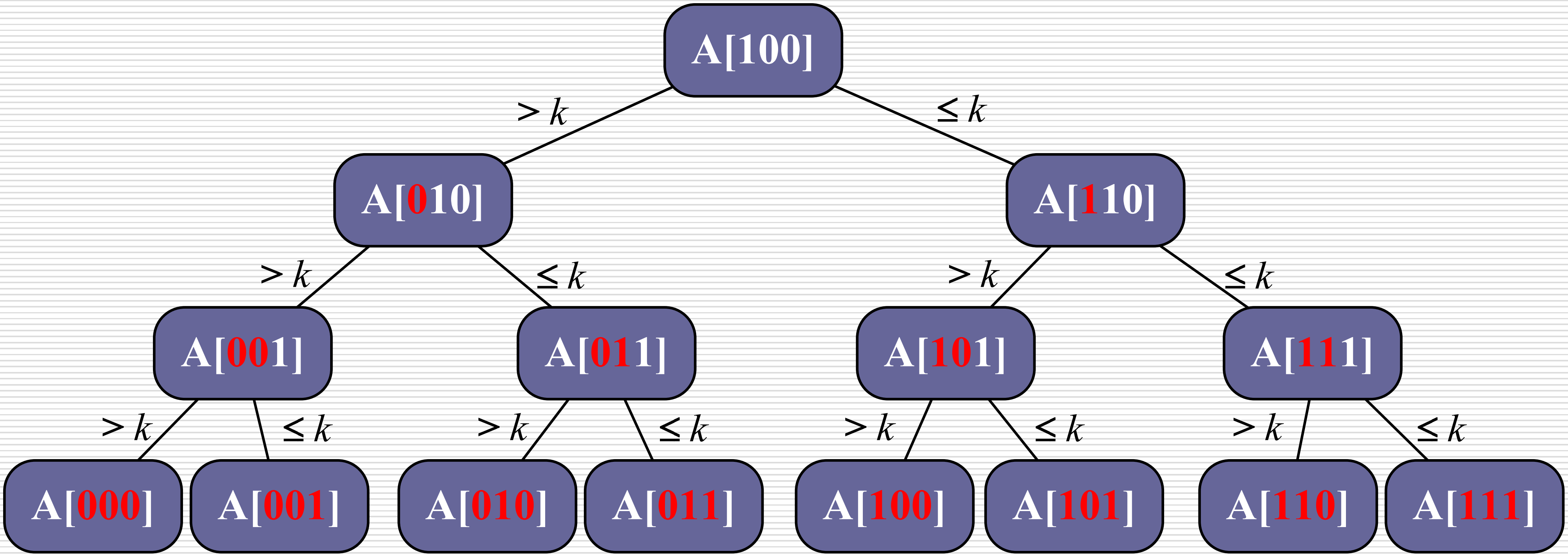
$$n, \frac{n}{2}, \frac{n}{4}, \cdots, \frac{n}{2^{\ell}}, \cdots, 1 \Rightarrow \ell \approx \log_2 n$$

- Χρόνος O(log *n*): βέλτιστος (χειρότερη περίπτωση).



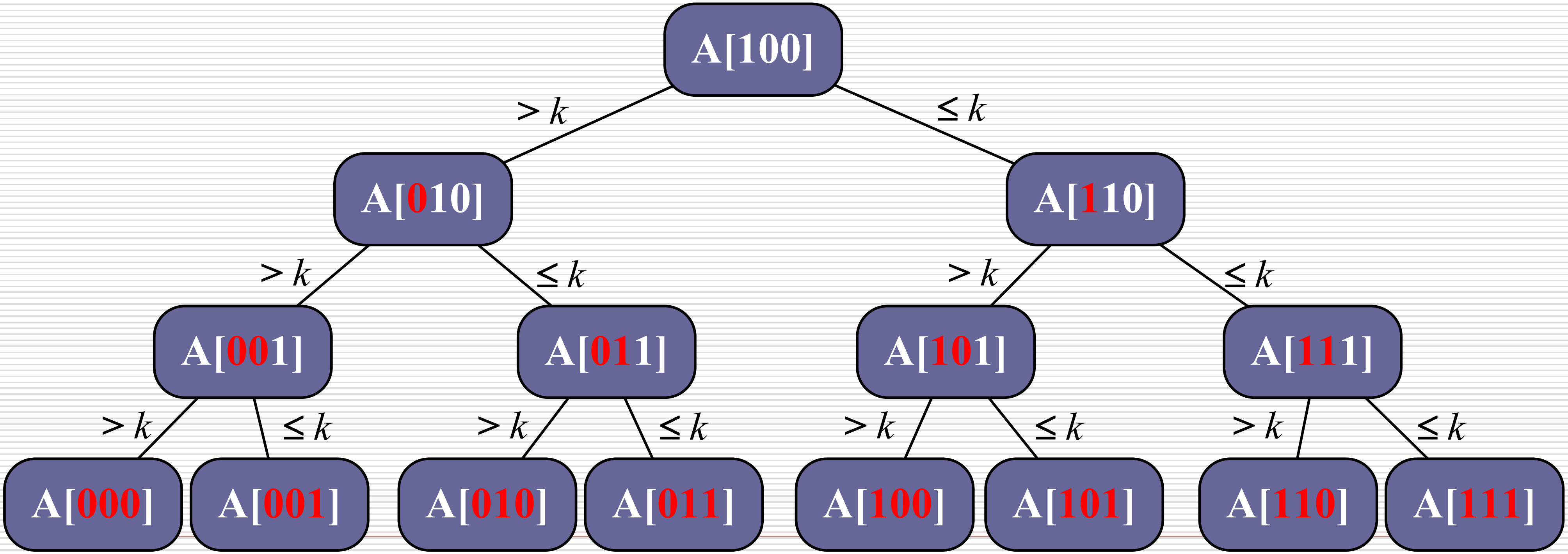
Χρόνος Εκτέλεσης

- Δυαδική αναπαράσταση θέσεων: κάθε σύγκριση προσδιορίζει ένα bit της θέσης του στοιχείου!
- Δυαδικό δέντρο συγκρίσεων έχει n φύλλα και $\log_2 n$ ύψος.



Χρόνος Εκτέλεσης

- Για κάθε αλγόριθμο, χωρίς άλλη πληροφορία (π.χ. κατανομή στοιχείων), μία σύγκριση προσδιορίζει ένα bit θέσης στοιχείου.
- Αν δεν γνωρίζουμε κατανομή στοιχείων, κάθε αλγόριθμος αναζήτησης χρειάζεται χρόνο $\Omega(\log n)$ στη χειρότερη περίπτωση.



Αναζήτηση με Παρεμβολή

- Συνήθως έχουμε κάποια πληροφορία σχετικά με την κατανομή των στοιχείων (π.χ. όταν ψάχνουμε τον τηλεφωνικό κατάλογο δεν ανοίγουμε στη μέση).
- Παρεμβολή αξιοποιεί την πληροφορία κατανομής. Ταχύτερη (στη μέση περ.) από δυαδική.
  - Αν ξέρουμε κατανομή, μια σύγκριση μπορεί να δίνει περισσότερο από ένα bit πληροφορίας για θέση στοιχείου.
  - Αναμενόμενη θέση *k* στο A[*low* … *up*]:

$$pos = low + (k - A[low]) \,/\, \text{μέση αύξηση} \,/\, \text{ανά θέση}$$

  - Κατά τα άλλα ίδια με δυαδική.

Αναζήτηση με Παρεμβολή

- Εκδοχή για ομοιόμορφη κατανομή σε διάστημα.
  - Προσαρμόζεται σε οποιαδήποτε άλλη κατανομή.

$$pos = low + (k - A[low]) \times \frac{up - low}{A[up] - A[low]}$$

```
int interpolationSearch(int A[], int n, int k) {
    int low = 0, up = n-1, pos;
    while (low <= up)  {
     if ((k < A[low]) || (k > A[up])) return(-1);
     pos = low + (int) ((double) (up - low))*
        (((double) (k - A[low])) / ((double) (A[up] - A[low])));
        if (A[pos] == k) return(pos);
        else if (A[pos] > k) up = pos - 1;
            else                  low = pos + 1;    }
    return(-1);    }
```

Αναζήτηση με Παρεμβολή

$$pos = low + (k - A[low]) \times \frac{up - low}{A[up] - A[low]}$$

Αναζήτηση **30**:

1	5	7	8	12	20	30	50

Αναζήτηση **30**:

1	8	13	15	30	38	45	50

Αναζήτηση με Παρεμβολή

- Χρόνος μέσης περίπτωσης : O(loglog n)
  - Αναζήτηση σε 1 τρισεκατ. στοιχεία με 6 συγκρίσεις!
- Χρόνος χειρότερης περίπτωσης : O(n)
- Χρόνος χειρότερης περίπτωσης βελτιώνεται με Δυαδική Αναζήτηση με Παρεμβολή.
- Πρώτη επανάληψη βρίσκει περιοχή με ακρίβεια. Επόμενες προσπελάσεις στην ίδια περιοχή.
  - Καλή αξιοποίηση της cache memory.