Αλγόριθμοι Ταξινόμησης

Στην ενότητα αυτή θα μελετηθούν τα εξής επιμέρους θέματα:

Οι αλγόριθμοι ταξινόμησης

SelectionSort, InsertionSort,

Mergesort, QuickSort,

BucketSort

Κάτω φράγμα της αποδοτικότητας Αλγορίθμων Ταξινόμησης και δένδρα αποφάσεων

Αλγόριθμοι ταξινόμησης

Δοθέντων μιας συνάρτησης f (ordering function) και ενός συνόλου στοιχείων

$$x_1, x_2, ..., x_n$$

η ταξινόμηση συνίσταται στη μετάθεση των στοιχείων ώστε να μπουν σε μια σειρά

$$x_{k_1}, x_{k_2}, ..., x_{k_n}$$

η οποία να ικανοποιεί

$$f(x_{k_1}) \le f(x_{k_2}) \le ... \le f(x_{k_n})$$

ή

$$f(x_{k_1}) \ge f(x_{k_2}) \ge ... \ge f(x_{k_n}).$$

Θα εξετάσουμε αλγόριθμους ταξινόμησης με κύριο γνώμονα την αποδοτικότητά τους (χρόνος εκτέλεσης, χρήση μνήμης).

Ταξινόμηση με Επιλογή (Selection Sort)

- Η *ταξινόμηση με επιλογή* βασίζεται στα ακόλουθα τρία βήματα:
  1. επιλογή του ελάχιστου στοιχείου
  2. ανταλλαγή με το πρώτο στοιχείο
  3. επανάληψη των βημάτων 1 και 2 για τα υπόλοιπα στοιχεία.

- Το ελάχιστο μεταξύ i στοιχείων μπορεί να βρεθεί με τη χρήση ενός while-loop, σε χρόνο O(i).

- Άρα ο χρόνος εκτέλεσης του Selection Sort είναι

$$\sum_{i=1}^{n-1} i = O(n^2)$$

Διαδικασία Selection Sort

```
void SelectionSort(int A[],int n){

int k;
int temp;

for (int i=1; i<n; i++){
   k=i;
   for (j = i+1; j <= n; j++)
       if A[j]<A[k] k=j;
   temp = A[i];
   A[i] = A[k];
   A[k] = temp;
}
}
```

Παράδειγμα Selection Sort

Θέση	**1**	**2**	**3**	**4**	**5**	**6**
Αρχική Τιμή	*34*	*8*	*64*	*51*	*32*	*33*
Με i=1	8	34	64	51	32	33
Με i=2	8	32	64	51	34	33
Με i=3	8	32	33	51	34	64
Με i=4	8	32	33	34	51	64
Με i=5	8	32	33	34	51	64

Ταξινόμηση με εισαγωγή (Insertion Sort)

- Η *ταξινόμηση με εισαγωγή* εισάγει ένα-ένα τα στοιχεία του συνόλου που εξετάζεται, στη σωστή τους θέση.

- Στη φάση i:
  1. υποθέτουμε πως ο πίνακας $A[1..(i\text{-}1)]$ είναι ταξινομημένος,
  2. εισάγουμε το στοιχείο $A[i]$ στην ακολουθία $A[1..(i\text{-}1)]$ στη σωστή θέση,
  3. μετακινώντας όλα τα στοιχεία που είναι μεγαλύτερα του $A[i]$ μια θέση δεξιά.

- Έστω μια ταξινομημένη ακολουθία από i στοιχεία. Ένα στοιχείο μπορεί να εισαχθεί στη σωστή του θέση μέσα στην ακολουθία σε χρόνο $O(i)$.

- Άρα ο χρόνος εκτέλεσης του Insertion Sort είναι

$$\sum_{i=1}^{n-1} i = O(n^2)$$

Διαδικασία Insertion Sort

```
void InsertionSort(int A[], int n){

  int temp;

  for (int i=2; i<=n; i++){
    temp=A[i];

    for (int j=i; (j > 1)&&(temp < A[j-1]); j--)
         A[j]=A[j-1];

     A[j]=temp;
  }
}
```

Παράδειγμα Insertion Sort

Θέση	**1**	**2**	**3**	**4**	**5**	**6**
Αρχική Τιμή	*34*	*8*	*64*	*51*	*32*	*21*
Με i=2	8	34	64	51	32	21
Με i=3	8	34	64	51	32	21
Με i=4	8	34	51	64	32	21
Με i=5	8	32	34	51	64	21
Με i=6	8	21	32	34	51	64

Σύγκριση Insertion Sort και Selection Sort

- Ο αλγόριθμος Selection sort απαιτεί $O(n^2)$ βήματα (δεν είναι δυνατή η γρήγορη έξοδος από τους βρόχους), έτσι η βέλτιστη περίπτωση είναι η ίδια με τη χείριστη περίπτωση.
- Στον αλγόριθμο Insertion Sort, είναι δυνατό να βγούμε από το δεύτερο βρόχο γρήγορα. Στη βέλτιστη περίπτωση (ο πίνακας είναι ήδη ταξινομημένος), ο χρόνος εκτέλεσης είναι της τάξης $O(n)$.
- Παρά τούτου, το Selection Sort είναι πιο αποδοτικό αν κρίνουμε τους αλγόριθμους με βάση τον αριθμό των μετακινήσεων (swaps) που απαιτούν:
  - το selection sort απαιτεί $O(n)$, μετακινήσεις,
  - το insertion sort, απαιτεί $O(n^2)$ μετακινήσεις (στη χείριστη περίπτωση όπου ο αρχικός πίνακας είναι ταξινομημένος σε φθίνουσα σειρά).

Ταξινόμηση με Συγχώνευση (Merge sort)

- Η *ταξινόμηση με συγχώνευση* είναι διαδικασία *διαίρει και βασίλευε* (divide and conquer, δηλ. αναδρομική διαδικασία όπου το πρόβλημα μοιράζεται σε μέρη τα οποία λύνονται ξεχωριστά, και μετά οι λύσεις συνδυάζονται.).
- Χρόνος Εκτέλεσης: O(n log n).
- Απαιτεί τη χρήση βοηθητικού πίνακα, αλλά είναι εύκολα κατανοητή και υλοποιήσιμη διαδικασία.
- Περιγραφή του Mergesort
  1. Μοιράζουμε τον πίνακα στα δύο.
  2. Αναδρομικά ταξινομούμε τα δύο μέρη.
  3. Συγχωνεύουμε τα αποτελέσματα των πιο πάνω αναδρομικών ταξινομήσεων.
- Παρατηρούμε ότι κατά την i-οστή κλήση της αναδρομικής διαδικασίας, κομμάτια μεγέθους 2^i είναι ταξινομημένα. Άρα το Merge Sort χρειάζεται να κληθεί log n φορές μέχρις ότου να ταξινομηθεί ολόκληρος ο πίνακας.

Παραδείγματα διαδοχικών κλήσεων

Δεδομένο Εισόδου:

34 57 28 3 15 8 26 73

Μετά την πρώτη εκτέλεση του Mergesort

34 57 3 28 8 15 26 73

Μετά τη δεύτερη εκτέλεση του Mergesort

3 28 34 57 8 15 26 73

Μετά την τρίτη εκτέλεση του Mergesort

3 8 15 26 28 34 57 73

Η διαδικασία συγχώνευσης

- Δύο ταξινομημένες λίστες μπορούν να συγχωνευθούν σε μία ταξινομημένη λίστα σε χρόνο γραμμικό ως προς το άθροισμα των μεγεθών των δύο λιστών.
- Η διαδικασία απαιτεί τη χρήση βοηθητικού πίνακα.
- Μπορούμε να χρησιμοποιούμε τον ίδιο βοηθητικό πίνακα temp για όλες τις κλήσεις του MergeSort.
- Μετά από τις δύο αναδρομικές κλήσεις του Mergesort στον αρχικό πίνακα A, τα δύο μισά του πίνακα είναι ταξινομημένα. Αντιγράφουμε τα δύο μισά στο βοηθητικό πίνακα temp, και τα συγχωνεύουμε πίσω στον A.

Παράδειγμα εκτέλεσης Merge (Συγχώνευσης)

- *Οι αρχικοί πίνακες και οι αρχικοί "δείκτες":*

  *13 17 20 36 *14 16 28 33

- Συγκρίνουμε τα στοιχεία που δείχνονται από τους δείκτες, διαλέγουμε και τυπώνουμε το μικρότερο και προχωρούμε το δείκτη αυτού:

  13 *17 20 36 *14 16 28 33

  Output:**13**

- Επαναλαμβάνουμε μέχρις ότου τα στοιχεία του ενός πίνακα εξαντληθούν οπότε τυπώνουμε τα υπόλοιπα στοιχεία του άλλου πίνακα:

  13 *17 20 36 14 *16 28 33

  Output: **13, 14**

  13 *17 20 36 14 16 *28 33

  Output: **13, 14, 16**

  13 17 *20 36 14 16 *28 33

  Output: **13, 14, 16, 17**

  13 17 20 *36 14 16 28 *33

  Output: **13, 14, 16, 17, 20, 28**

  Output: **13, 14, 16, 17, 20, 28, 33, 36**

Διαδικασία Merge Sort

```
void MergeSort(int A[], int temp[],int l, int r){

  if (l==r) return;
  int mid = (l+r)/2;
  Mergesort(A, temp, l, mid);
  Mergesort(A, temp, mid+1, r);

  int i, j, k;
  for (i=l; i<= r; i++)
       temp[i] = A[i];
  for ( i=l,j=mid+1,k=l;  i<=mid && j<=r && k<=r;   k++)
       if (temp[i]<temp[j])        A[k]= temp[i]; i++;
       else                        A[k]= temp[j]; j++;
  for ( ; i <= mid ; i++, k++ )
       A[k] = temp[i];
  for ( ; j <= r ; j++, k++ )
       A[k] = temp[j];
}
```

Παράδειγμα Merge Sort

Δεδομένο Εισόδου:

1	***2***	***3***	***4***	***5***	***6***	***7***	***8***
36	*20*	*17*	*13*	*28*	*14*	*23*	*15*

l=1, r=8, mid = 4

Μετά από την εκτέλεση των Mergesort(A,temp,1,4) και Mergesort(A, temp, 5, 8)

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
13	17	20	36	14	15	23	28

Μετά από την αντιγραφή ο πίνακας temp περιέχει τα ίδια στοιχεία όπως ο Α

Μετά από την εκτέλεση των τελευταίων τριών for-βρόχων ο Α περιέχει

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
13	14	15	17	20	23	28	36

Μερικά Σχόλια

- Η αντιγραφή και η συγχώνευση παίρνουν χρόνο O(n).
- Στη χείριστη περίπτωση ο χρόνος εκτέλεσης ικανοποιεί:

  1. Βάση της αναδρομής
     $$T(0) = T(1) = 1$$
  2. Αναδρομική περίπτωση
     $$T(n) = 2 \cdot T(n/2) + n$$

- Άρα ο χρόνος εκτέλεσης είναι $\Theta(n \log n)$.
- Μπορούμε να μειώσουμε την ανάγκη αντιγραφής (από τον A στον temp) με την ανταλλαγή των ρόλων των δύο πινάκων σε διαδοχικές κλήσεις της διαδικασίας Mergesort.

Κάτω φράγμα για αλγόριθμους ταξινόμησης

- Ξέρουμε πως το πρόβλημα ταξινόμησης μπορεί να λυθεί σε χρόνο O(n log n) (Heap Sort και Merge Sort).
- Υπάρχει πιο αποδοτικός αλγόριθμος ταξινόμησης;
- Θα δείξουμε πως κάθε αλγόριθμος ταξινόμησης είναι Ω(n log n).
- Ως μονάδα μέτρησης αποδοτικότητας θα χρησιμοποιήσουμε τον *αριθμό συγκρίσεων* που απαιτεί κάποιος αλγόριθμος.
- Υποθέτουμε ότι κάθε στοιχείο του πίνακα που θέλουμε να ταξινομήσουμε είναι διαφορετικό από όλα τα άλλα.

Σειριακή διάταξη στοιχείων και ταξινόμηση

- Η συμπεριφορά ενός αλγόριθμου ταξινόμησης εξαρτάται μόνο από τη σχετική σειρά μεταξύ των στοιχείων που ταξινομούμε και όχι από τα συγκεκριμένα στοιχεία.

- Δηλαδή: αν A και B είναι δύο πίνακες τέτοιοι ώστε για κάθε i και j,

  $A[i] < A[j]$ αν και μόνο αν $B[i] < B[j]$,

  τότε ο αριθμός των βημάτων (όπως και ο αριθμός των συγκρίσεων) που θα εκτελέσει κάποιος αλγόριθμος με δεδομένο εισόδου A θα είναι ο ίδιος με τον ανάλογο αριθμό που θα εκτελέσει με δεδομένο εισόδου B.

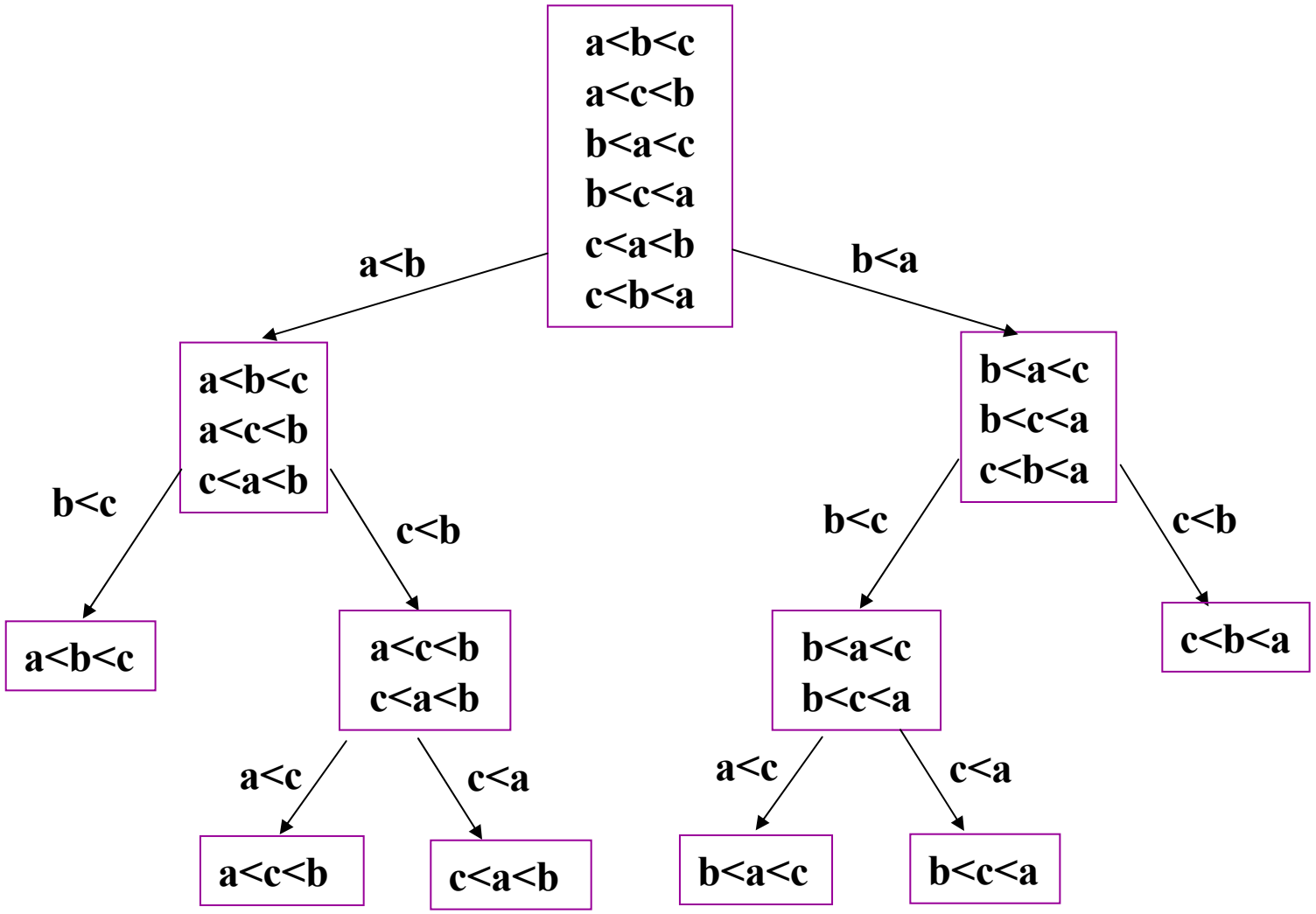
- Άρα, η σειριακή διάταξη των στοιχείων του δεδομένου εισόδου A, $A[1],A[2],\ldots,A[n]$, έχει κύρια σημασία.

- Υπάρχουν n! 'διαφορετικές' τοποθετήσεις n ξεχωριστών στοιχείων. Άρα υπάρχουν n! διαφορετικά δεδομένα εισόδου.

Ανάλυση Αλγόριθμων Ταξινόμησης

- Θα περιγράψουμε τη συμπεριφορά ενός αλγόριθμου ως ένα *δένδρο αποφάσεων* (decision tree).
- Στη ρίζα επιτρέπονται όλες οι διαφορετικές ‘σειρές’ των στοιχείων.
- Ας υποθέσουμε πως ο αλγόριθμος συγκρίνει τα δύο πρώτα στοιχεία $A[1]$ και $A[2]$. Τότε το αριστερό παιδί του δένδρου αντιστοιχεί στην περίπτωση $A[1] < A[2]$ και το δεξί παιδί της ρίζας στην περίπτωση $A[2] < A[1]$.
- Σε κάθε κόμβο, μια σειρά στοιχείων είναι νόμιμη αν ικανοποιεί όλες τις συγκρίσεις στο μονοπάτι από τη ρίζα στον κόμβο.
- Τα φύλλα αντιστοιχούν στον τερματισμό του αλγόριθμου και κάθε φύλλο περιέχει το πολύ μια νόμιμη σειρά στοιχείων.

Δένδρο αποφάσεων για 3 στοιχεία



Κάτω φράγμα

- Έστω P ένας αλγόριθμός ταξινόμησης, και έστω T το δένδρο αποφάσεων του P με δεδομένο εισόδου μεγέθους n.
- Ο αριθμός των φύλλων του T είναι n!
- Το ύψος του T είναι ένα κάτω φράγμα του χείριστου χρόνου εκτέλεσης του αλγόριθμου P.
- Ένα δυαδικό δένδρο ύψους d έχει το πολύ 2^d φύλλα.
- Άρα το T έχει ύψος το λιγότερο log n!
- Συμπέρασμα: $P \in \Omega(\log n!) = \Omega(n \log n)$.
- Η μέση περίπτωση είναι επίσης n log n.

Αλγόριθμος Quick Sort

- Πρακτικά, ο πιο γρήγορος αλγόριθμος.
- Στη χείριστη περίπτωση ο αλγόριθμος Quick Sort είναι $O(n^2)$. Τέτοιες περιπτώσεις όμως έχουν μικρή πιθανότητα. Έχουμε μάλιστα ότι:
- *Στη μέση περίπτωση ο αλγόριθμος είναι $O(n \log n)$.*
- Τα περισσότερα συστήματα χρησιμοποιούν το Quick Sort (π.χ. Unix).
- Όπως και το Merge Sort, παράδειγμα αλγόριθμου διαίρει και βασίλευε.

Περιγραφή του Quick Sort

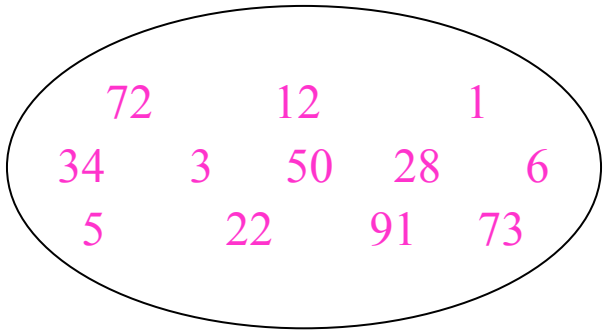
- Αν το δεδομένο εισόδου περιέχει 0 ή 1 στοιχεία δεν κάνουμε τίποτα.
- Διαφορετικά, αναδρομικά:
  1. διαλέγουμε ένα στοιχείο p, το οποίο ονομάζουμε το *άξον* στοιχείο (the pivot) και το αφαιρούμε από το δεδομένο εισόδου.
  2. χωρίζουμε τον πίνακα σε δύο μέρη S1 και S2, όπου το S1 περιέχει όλα τα στοιχεία του πίνακα που είναι μικρότερα από το p, και το S2 περιέχει τα υπόλοιπα στοιχεία (όλα τα στοιχεία που είναι μεγαλύτερα από το p).
  3. Καλούμε αναδρομικά τον αλγόριθμο

     στο S1, και παίρνουμε απάντηση το T1,

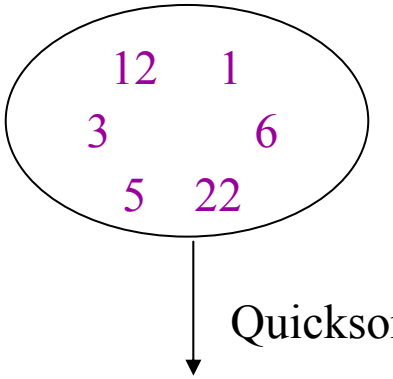
     και

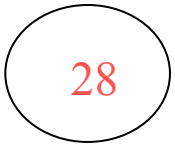
     στο S2, και παίρνουμε απάντηση το T2.
  4. Επιστρέφουμε τον πίνακα T1, p, T2.

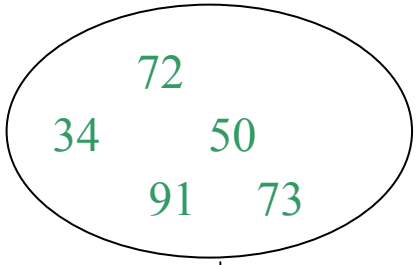
Παράδειγμα εφαρμογής του Quicksort



χωρίζουμε με pivot το 28







Quicksort

Quicksort

1, 3, 5, 6, 12, 22

34, 50, 72, 73, 91

Αποτέλεσμα: 1, 3, 5, 6, 12, 22, 28, 34, 50, 72, 73, 91

Ο Αλγόριθμος Quicksort

```
void Quicksort(int A[], int i, int j){
if (j-i<=1) return;
int pivotIndex = FindPivot(A, i, j);
int pivot = A[pivotIndex];
swap(A, pivotIndex, j);
int k = partition (A, i, j-1, pivot);
swap(A, k, j);
Quicksort(A, i, k-1);
QuickSort(A, k+1, j);
}
```

- όπου η διαδικασία `FindPivot(A,i,j)` βρίσκει και επιστρέφει τη θέση του άξονα του A[i…j]
- και η διαδικασία `partition (A,i,j-1,pivot),` χωρίζει τον πίνακα A[i…j-1] έτσι ώστε A[i..k-1] να περιέχει στοιχεία < pivot, A[k…j-1] να περιέχει στοιχεία >pivot, και επιστρέφει την τιμή k.

Διαδικασία FindPivot(A, i, j)

- Η αποδοτικότητα του αλγόριθμου βασίζεται στην επιλογή του 'μεσαίου' στοιχείου (άξονα), pivot. Το pivot θα πρέπει να χωρίσει τον πίνακα Α, περίπου, σε δύο μισά.
- Ιδανικά, το pivot θα θέλαμε να είναι ακριβώς το μεσαίο των στοιχείων.
- Από την άλλη, πρέπει να μπορούμε να βρίσκουμε το pivot γρήγορα, σε χρόνο O(1).
- Πρώτη προσπάθεια: διαλέγουμε το πρώτο στοιχείο. Κακή επιλογή γιατί συχνά το δεδομένο εισόδου είναι σχεδόν ταξινομημένο. Κατά συνέπεια, τα δύο μέρη θα διαφέρουν κατά πολύ ως προς τον αριθμό των στοιχείων τους, και ο αλγόριθμος θα είναι συχνά της τάξης $O(n^2)$.
- Καλύτερη επιλογή, το στοιχείο που βρίσκεται στη μέση του πίνακα:

```
int FindPivot(int A[],int l,int r){
  return ((l+r)/2);
}
```

Διαδικασία Partition(A, l, r, p)

- Με δεδομένο εισόδου τον πίνακα A[l…r], και pivot p, θέλουμε να χωρίσουμε τον πίνακα σε δύο μέρη ως προς το p.
- Το πιο πάνω πρέπει να επιτευχθεί χωρίς τη χρήση δεύτερου πίνακα.
- Βασική Ιδέα:
  1. Επαναλαμβάνουμε τα εξής μέχρις ότου τα l και r να διασταυρωθούν.
  2. Προχώρα το l προς τα δεξιά όσο τα στοιχεία που βρίσκεις είναι μικρότερα του p,
  3. προχώρα το r προς τα αριστερά όσο τα στοιχεία που βρίσκεις είναι μεγαλύτερα του p,
  4. αντάλλαξε τα στοιχεία που δείχνονται από τα l και r.

Η διαδικασία Partition

```
int partition(int A[], int l, int r, int p)}

while (r>l) {
   while (A[l]< p) l++;
   while (A[r]> p && r>= l) r--;
   if (l<r)
       swap (A, l, r);
   else
       break;
}
return l;
}
```

Παράδειγμα Εκτέλεσης Partition

Δεδομένο Εισόδου:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
72	*6*	*37*	*48*	*30*	*42*	*83*	*75*

pivot = 48, μετακίνηση του pivot στο τέλος (swap(4, 8)):

72	6	37	75	30	42	83	48
l						**r**	

εκτέλεση του Partition(A, l, r, 48):

72	6	37	75	30	42	83	48
l					**r**		
42	6	37	75	30	72	83	48
			l	**r**			
42	6	37	30	75	72	83	48
			l	**r**			
42	6	37	30	75	72	83	48
			r	**l**			

Παράδειγμα Εκτέλεσης Partition

- Η διαδικασία Partition(A, 1, 7, 48) επιστρέφει την τιμή l=5 και ακολουθείται από τη μετακίνηση swap(5, 8):

  42 6 37 30 48 72 83 75

- Στη συνέχεια εκτελούνται αναδρομικά οι διαδικασίες Quicksort(A, 1,4), Quicksort(A, 6, 8), και έτσι επιστρέφεται το επιθυμητό αποτέλεσμα.

- Σημειώστε πως μετά την εκτέλεση της Partition το στοιχείο 48 (και σε κάθε κλήση της αναδρομής το στοιχείο pivot) αποκτά τη σωστή του θέση στην ταξινομημένη μορφή του πίνακα.

Ανάλυση του Χρόνου Εκτέλεσης

- Η εύρεση του pivot απαιτεί χρόνο O(1) και η διαδικασία Partition(A, l, r, p) εκτελείται σε χρόνο $O(r-l) \in O(n)$.
- Έστω T(n) ο χρόνος εκτέλεσης του Quicksort σε δεδομένο εισόδου μεγέθους n. Για τη βάση της αναδρομής έχουμε:

  $T(0) = T(1) = c$

  και για $n > 1$

  $T(n) = T(i) + T(n-i-1) + cn$

  όπου i είναι το μέγεθος είναι το μέγεθος του αριστερού κομματιού μετά από την Partition.
- *Χείριστη περίπτωση*: $i = 0$ ή $n-1$. Τότε

  $T(n) = T(n-1) + cn$

  και $T(n) \in O(n^2)$.
- *Βέλτιστη περίπτωση*: $i = n/2$

  $T(n) = 2\,T(n/2) + cn$

  και $T(n) \in O(n \log n)$.

Ανάλυση Μέσης Περίπτωσης

- Θεωρούμε όλες τις πιθανές περιπτώσεις της συμπεριφοράς της διαδικασίας Partition(A, i, j, p), όπου j-i = n-1. Υπάρχουν n τέτοιες περιπτώσεις: το αριστερό κομμάτι του partition μπορεί να έχει από 0 μέχρι n-1 στοιχεία.
- Ας υποθέσουμε πως οι n αυτές περιπτώσεις είναι ισοπίθανες, δηλαδή η κάθε μια έχει πιθανότητα 1/n.
- Τότε η μέση περίπτωση του T(n) δίνεται ως

$$T(n) = c \cdot n + \frac{1}{n} \cdot \sum_{i=0}^{n-1} [T(i) + T(n-1-i)]$$

$$= c \cdot n + \frac{2}{n} \cdot \sum_{i=0}^{n-1} T(i)$$

- Επίλυση της αναδρομικής σχέσης δίνει $T(n) \in O(n \log n)$.

Παρατηρήσεις

- Πως δουλεύει η διαδικασία Partition με δεδομένο εισόδου πίνακα με πολλά στοιχεία ίσα με το pivot;
- Υπάρχουν άλλες στρατηγικές για επιλογή του pivot;
  1. pivot = mid(A[1], A[n], A[n/2])
  2.
- Στην πράξη, για δεδομένα εισόδου μικρού μεγέθους το InsertionSort δουλεύει πιο αποδοτικά. Επομένως μια καλή στρατηγική θα ήταν να συνδυάσουμε τους δύο αλγόριθμους ώστε σε μικρούς πίνακες (π.χ. $n \leq 10$) να χρησιμοποιείται το InsertionSort και σε μεγάλους το Quicksort: στη διαδικασία Quicksort ανταλλάξτε την πρώτη γραμμή με την εξής:

```
if (j-i)<= 10 InsertionSort(A[i…j], j-i);
```

- Ακόμα ένας πιθανός τρόπος βελτίωσης του χρόνου εκτέλεσης είναι η χρήση στοίβας αντί αναδρομής.

Αλγόριθμος BucketSort

- Έστω ότι ο πίνακας Α περιέχει στοιχεία που ανήκουν στο διάστημα [1..m].
- Ο *αλγόριθμος BucketSort* πετυχαίνει ταξινόμηση του Α σε χρόνο O(n+m):
  1. Δημιουργούμε ένα πίνακα count μήκους m και θέτουμε count[i]=0, για όλα τα i.
  2. Διαβάζουμε τον πίνακα Α ξεκινώντας από το πρώτο στοιχείο. Αν διαβάσουμε το στοιχείο α, τότε αυξάνουμε την τιμή του count[α] κατά ένα.
  3. Διαβάζουμε τον πίνακα count, ο οποίος περιέχει αναπαράσταση του ταξινομημένου πίνακα, και μεταβάλλουμε ανάλογα τον πίνακα Α.

π.χ. με δεδομένο εισόδου

A = [3, 5, 7, 2, 1, 6, 7, 4, 5, 6, 1, 3, 7]

εφαρμογή του αλγόριθμου δίνει

count = [2, 1, 2, 1, 2, 2, 3]

το οποίο μεταφράζεται ως

A = [1, 1, 2, 3, 3, 4, 5, 5, 6, 6, 7, 7, 7]

Χρόνος Εκτέλεσης

- Αν $m \in O(n)$, τότε ο χρόνος εκτέλεσης του αλγόριθμου είναι $O(n)$.
- Αυτό διαψεύδει το κάτω φράγμα $\Omega(n \log n)$ της διαφάνειας 22;
- ΟΧΙ, γιατί το μοντέλο είναι διαφορετικό: στην ανάλυση κάτω φράγματος υποθέσαμε ότι η μόνη πράξη που μπορούμε να εφαρμόσουμε στα δεδομένα είναι η δυαδική σύγκριση στοιχείων. Ο αλγόριθμος BucketSort όμως στο Βήμα 2 ουσιαστικά εφαρμόζει *m-αδική σύγκριση*, σε χρόνο $O(1)$.
- Αυτό μας υπενθυμίζει πως σχεδιάζοντας ένα αλγόριθμο και λαμβάνοντας υπόψη κάποια αποδεδειγμένα κάτω φράγματα πρέπει πάντα να αναλύουμε το μοντέλο στο οποίο δουλεύουμε: η ύπαρξη και αξιοποίηση περισσότερων πληροφοριών πιθανόν να επιτρέπουν τη δημιουργία αποδοτικότερων αλγορίθμων.

Ταξινόμηση πολύπλοκων αρχείων

- Οι αλγόριθμοι που έχουμε παρουσιάσει υποθέτουν πως οι πίνακες-δεδομένα εισόδου περιέχουν ακέραιους αριθμούς και προϋποθέτουν μετακίνηση των στοιχείων μέσα στον πίνακα.
- Συχνά θέλουμε να ταξινομήσουμε αρχεία που περιέχουν πολύπλοκα αντικείμενα ως προς διάφορες σχέσεις σειράς (π.χ. ως προς αλφαβητική σειρά του τελευταίου πεδίου των αντικειμένων).
- Σε τέτοιες περιπτώσεις η μετακίνηση (swapping) στοιχείων είναι δαπανηρή.
- Αυτό μπορεί να αποφευχθεί με τη *χρήση δεικτών*: ως δεδομένο εισόδου χρησιμοποιούμε πίνακα που περιέχει δείκτες στα στοιχεία που θέλουμε να ταξινομήσουμε. Σύγκριση γίνεται με έλεγχο των σχετικών πεδίων των αντικειμένων που δείχνονται από τους δείκτες και μετακίνηση γίνεται στο επίπεδο των δεικτών.
- Οι αλγόριθμοι παραμένουν οι ίδιοι.
- Η μέθοδος ονομάζεται *έμμεση ταξινόμηση* (indirect sorting).

Εξωτερική ταξινόμηση

- Έχουμε υποθέσει πως για την ταξινόμηση ενός αρχείου μπορούμε να μεταφέρουμε τις εγγραφές του αρχείο σε ένα πίνακα και να εφαρμόσουμε ένα αλγόριθμο ταξινόμησης.
- Αυτό είναι ρεαλιστικό για μικρού μεγέθους αρχεία (που μπορούν να χωρέσουν σε ένα πίνακα κύριας μνήμης). Αυτή η μέθοδος ονομάζεται εσωτερική ταξινόμηση (internal sorting).
- Σε αντίθεση, όταν θέλουμε να ταξινομήσουμε μεγάλα αρχεία επιβάλλεται η χρήση βοηθητικής μνήμης (external sorting).
- Η ταξινόμηση γίνεται κατά τμήματα: ένα μέρος του αρχείου μεταφέρεται στην κύρια μνήμη, ταξινομείται και αποθηκεύεται σε ένα προσωρινό αρχείο. Το επόμενο τμήμα μεταφέρεται στην κύρια μνήμη και ταξινομείται και μετά συγχωνεύεται με το προσωρινό αρχείο. Η διαδικασία επαναλαμβάνεται μέχρι εξάντλησης του αρχικού αρχείου.
- Με βάση αυτή την κύρια ιδέα υπάρχουν διάφοροι αλγόριθμοι εξωτερικής ταξινόμησης. Κύριος στόχος τους είναι η αποδοτική επεξεργασία της βοηθητικής μνήμης.