



QUICKSORT

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare

Partiționare

- Partiționarea reprezintă mecanismul de bază pentru sortarea rapidă
- **Partiționarea** datelor = împărțirea acestora în două mulțimi, astfel încât toate elementele cu valori mai mari decât o valoare precizată se află într-un grup, iar cele cu valori mai mici decât aceeași valoare, în celălalt grup
- Vezi demonstrația Partition




Partiționare

- După efectuarea partiționării, datele nu sunt sortate, ci doar împărțite în două clase
- Gradul de sortare este oricum ceva mai mare decât înainte de partiționare



Partiționare

- Partiționarea nu este stabilă
- Niciunul dintre grupuri nu este în ordinea în care se găsea inițial
- Partiționarea tinde să inverseze ordinea unora din elementele fiecărui grup



```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int nElems;    //numarul de elemente din vector
int *theArray; //vectorul care trebuie partitionat
int partitie(int left, int right, int pivot);
void swap(int *index1, int *index2);
```


```
int main(void) {
    int pivot;
    printf("Introduceti dimensiunea vectorului nElems
                                                = ");
    scanf("%d", &nElems);
    theArray = (int*) malloc(nElems * sizeof(int));
                                                // creeaza vectorul
    for (int i = 0; i < nElems; i++) {
        printf("Introduceti elementul a[%d] = ", i);
        scanf("%d", &theArray[i]);
    }
    printf("Vectorul inainte de sortare este\n");
    for (int i = 0; i < nElems; i++)
        printf("a[%d] = %d\n", i, theArray[i]);
}
```

```
printf("Introduceti valoarea pivotului = ");
scanf("%d", &pivot);
printf("Pivotul este %d\n", pivot);
                                // partitioneaza vectorul
int partIndex = partitie(0, nElems - 1, pivot);
printf("Partitia este la indexul %d\n", partIndex);
printf("Vectorul partitionat este\n");
for (int i = 0; i < nElems; i++)
    printf("a[%d] = %d\n", i, theArray[i]);
getch();
} // end main()
```


```
int partitie(int left, int right, int pivot)
{
    int leftPtr = left - 1;
        // la stanga primului element
    int rightPtr = right + 1;
        // la dreapta ultimului element
    while(true)
    {
        while(leftPtr < right &&
            theArray[++leftPtr] < pivot)
            // gaseste element mai mare
        ;    // (nop)
```



```
while(rightPtr > left &&
        theArray[--rightPtr] > pivot)
    // gaseste element mai mic
    ; // (nop)
if(leftPtr >= rightPtr) // daca indicii se suprapun,
    break;             // partitia e gata
else                  // daca nu
    swap(&theArray[leftPtr], &theArray[rightPtr]);
    // interschimba doua elemente
} // end while(true)
return leftPtr;       // intoarce indexul leftPtr
} // end partitie()
```




```
void swap(int *index1, int *index2)
    // interschimba doua elemente
{
    int temp;
    temp = *index1;
    *index1 = *index2;
    *index2 = temp;
} // end swap()
```



Indexul din stânga, *leftPtr*, se deplasează spre dreapta, iar cel din dreapta, *rightPtr*, se deplasează spre stânga

Inițial, *leftPtr* indică poziția de la stânga primei celule, iar *rightPtr* poziția de la dreapta ultimei celule, din cauză că incrementarea și decrementarea celor doi indici se va efectua înainte de prima lor utilizare efectivă

Cât timp *leftPtr* indică numai elemente mai mici decât valoarea pivot, el își continuă deplasarea spre stânga, deoarece elementul se găsește în grupul potrivit



Când elementul întâlnit este mai mare decât valoarea pivot, *leftPtr* se oprește

Analog pentru *rightPtr*

Această deplasare este controlată de două bucle *while* interioare, una pentru *leftPtr* și cealaltă pentru *rightPtr*

După efectuarea permutării, cei doi indici își continuă deplasarea, oprindu-se din nou la elemente care nu aparțin grupului potrivit și permutându-le

Când cei doi indici se întâlnesc, partiționarea este completă și bucla exterioară se termină




Ce se întâmplă dacă înlocuim linia:

```
while (leftPtr < right && theArray[++leftPtr] < pivot);
```

cu liniile:

```
while (leftPtr < right && theArray[leftPtr] < pivot)  
    ++leftPtr;
```

Valoarea inițială a lui *leftPtr* va fi chiar *left*, ceea ce este mai evident decât *left - 1*



În urma acestei schimbări, *leftPtr* se va incrementa numai când condiția este îndeplinită
leftPtr se deplasează însă și în caz contrar, deci va fi necesară o instrucțiune suplimentară în bucla *while* exterioară, pentru a modifica valoarea lui *leftPtr*

În concluzie, varianta cu instrucțiunea *vidă* reprezintă soluția cea mai eficientă

Aceeași discuție pentru *rightPtr*



Quicksort

- Este cel mai rapid dintre toți algoritmi de sortare, în majoritatea cazurilor
- A fost descoperit de Charles Antony Richard Hoare în 1962
- Se partiționează un vector în doi sub-vectori, apelându-se apoi, recursiv, fiecare dintre acești sub-vectori



Quicksort


- Există câteva artificii, care se referă la maniera de alegere a pivotului și la sortarea partițiilor de dimensiuni reduse



Algoritmul recursiv

- 1) Partiționarea vectorului în două grupuri:
 - Cel din stânga (cu valori mai mici)
 - Cel din dreapta (cu valori mai mari)
- 2) Apelul recursiv pentru sortarea grupului stâng
- 3) Apelul recursiv pentru sortarea grupului drept


```
void recQuickSort(int left, int right)
{
    if(right - left <= 0)    // daca dimensiunea < 1,
        return;            // partitia e deja sortata
    else    // dimensiunea e cel putin 2
    {        // stabileste domeniul partitiei
        int partition = partitie(left, right);
        recQuickSort(left, partition - 1);
                        // sorteaza partea stanga
        recQuickSort(partition + 1, right);
                        // sorteaza partea dreapta
    }
} // end recQuickSort()
```



Parametrii funcției *recQuickSort* reprezintă capetele din stânga și respectiv din dreapta pentru vectorul (sau sub-vectorul) care trebuie sortat

Dacă vectorul are două sau mai multe elemente, se apelează mai întâi funcția *partitie*, pentru a-l partiționa

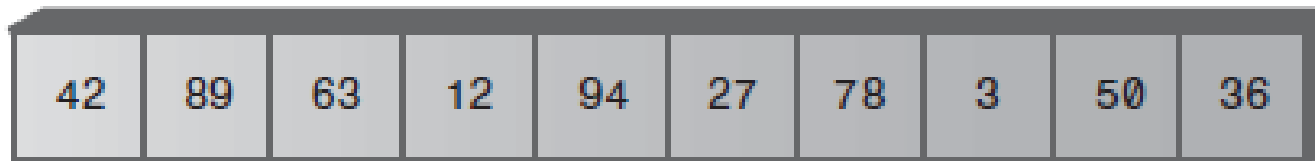
Funcția întoarce indicele de partiție – acesta este indicele primului element din sub-vectorul drept (care conține valorile mai mari)



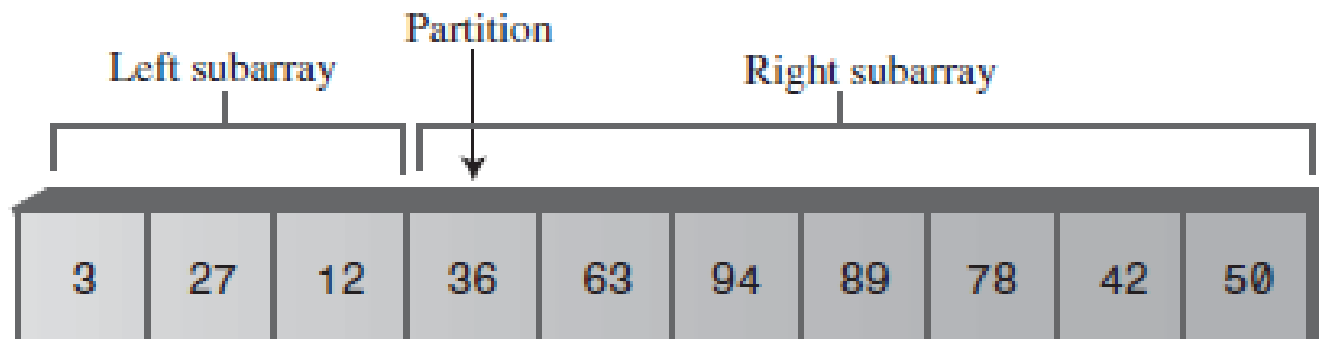
După partiționarea vectorului, funcția *recQuickSort* se apelează recursiv, mai întâi pentru partea stângă a vectorului, cuprinsă între *left* și *partition-1*, apoi și pentru partea dreaptă, dintre *partition+1* și *right*

Elementul cu indicele *partition* nu apare în niciunul din apelurile recursive. De ce ?
Răspunsul este dat de modalitatea de alegere a pivotului

Unpartitioned array



↑
Pivot



↑
Left

Will be sorted
by first recursive
call to recQuickSort()

↑
Already
Sorted

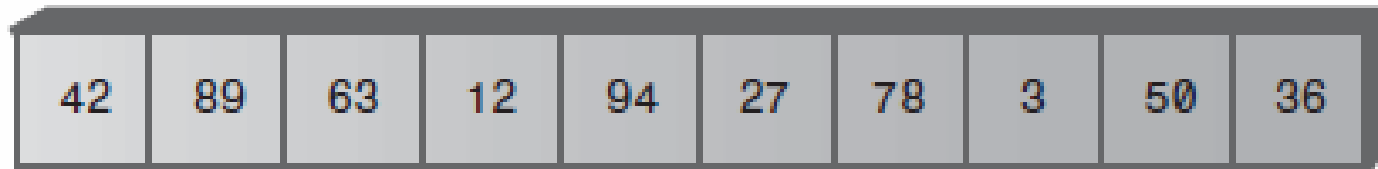


Will be sorted
by second recursive
call to recQuickSort()

Alegerea unei valori pivot

- Valoarea pivot trebuie să reprezinte valoarea unui element din vector – acest element se numește **pivot**
- Se alege ca pivot ultimul element al vectorului
- După partiționare, dacă pivotul va fi inserat între sub-vectorul din stânga și cel din dreapta, acesta se va afla în poziția în care se va găsi și după sortarea vectorului

Unpartitioned array




Pivot item



An upward-pointing arrow from the text "Pivot item" to the last element of the array, 36.

Correct place
for pivot

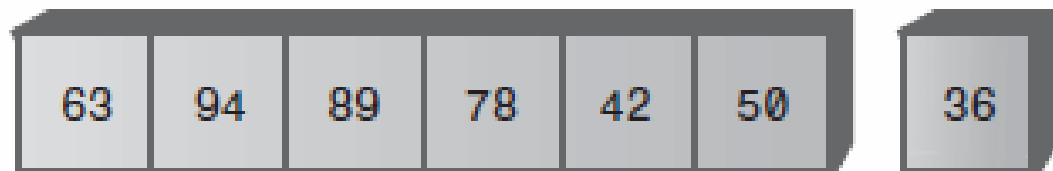


A downward-pointing arrow from the text "Correct place for pivot" to the space between the left and right subarrays.

Partitioned
left subarray



Partitioned
right subarray

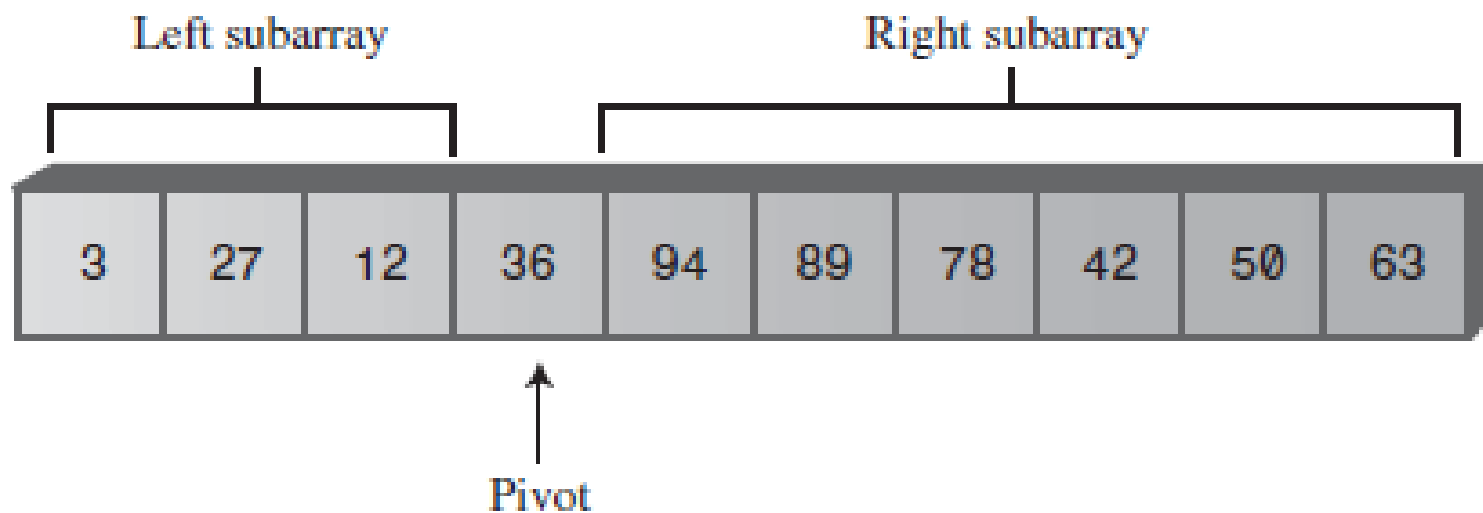
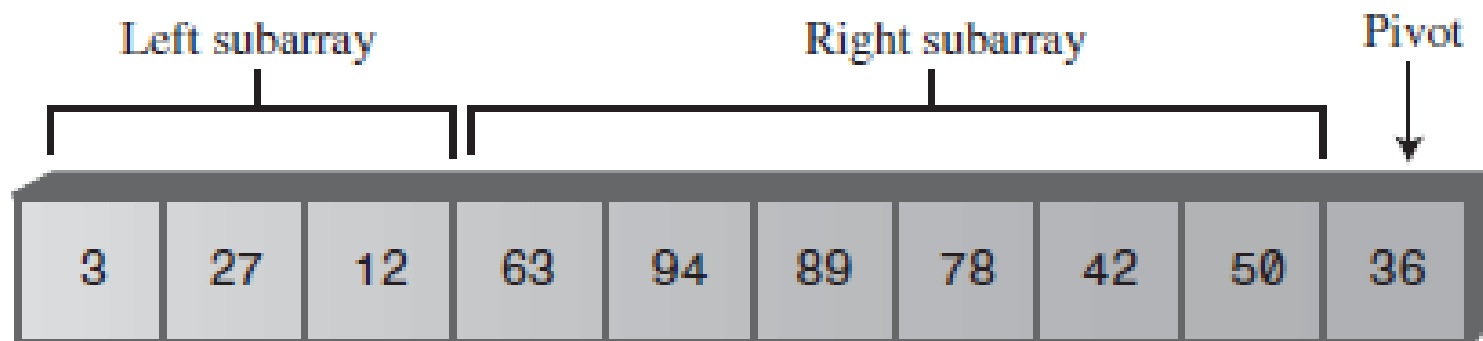



Cum deplasăm pivotul ?

- O variantă este de a deplasa toate elementele din sub-vectorul drept cu o celulă spre dreapta, pentru a crea spațiu pivotului
- Această variantă este ineficientă și inutilă
- Elementele din sub-vectorul drept, deși sunt toate mai mari decât pivotul, nu sunt încă sortate, deci pot fi deplasate în orice mod în cadrul sub-vectorului, fără a afecta ordinea finală

Cum deplasăm pivotul ?

- Pentru a simplifica inserția pivotului (36) în locul potrivit, acesta se permută cu elementul din stânga sub-vectorului (63)
- Pivotul ajunge în locul potrivit
- Elementul 63 ajunge în capătul din dreapta al vectorului, partiționarea nefiind afectată de această permutare







După ce este permutat între cele două
partiții, pivotul își ocupă locul final în vector
Pașii următori ai algoritmului vor modifica
ordinea elementelor din stânga sau din
dreapta sa

Includem și operația de selecție a pivotului în
funcția *recQuickSort*


Valoarea pivotului se transmite, ca al treilea
parametru, pentru funcția *partitie*



```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int nElems; //numarul de elemente din vector
int *theArray; //vectorul care trebuie sortat crescator
void quickSort();
void recQuickSort(int left, int right);
int partitie(int left, int right, double pivot);
void swap(int *index1, int *index2);
```




```
int main(void)
{
    int pivot;
    printf("Introduceti dimensiunea vectorului nElems
                                                = ");
    scanf("%d", &nElems);
    theArray = (int*) malloc(nElems * sizeof(int));
                                                // creeaza vectorul
    for (int i = 0; i < nElems; i++) {
        printf("Introduceti elementul a[%d] = ", i);
        scanf("%d", &theArray[i]);
    }
}
```




```
printf("Vectorul inainte de sortare este\n");
    for (int i = 0; i < nElems; i++)
        printf("a[%d] = %d\n", i, theArray[i]);
    quickSort();           // quicksort
    printf("Vectorul sortat este\n");
    for (int i = 0; i < nElems; i++)
        printf("a[%d] = %d\n", i, theArray[i]);
    getch();
} // end main()
```

```
void quickSort()
{
    recQuickSort(0, nElems - 1);
}
```




```
void recQuickSort(int left, int right)
{
    if (right - left <= 0)        // if size < 1,
        return;                  // already sorted
    else                          // size is 2 or larger
    {
        double pivot = theArray[right]; // rightmost item
                                         // partition range
        int partition = partitie(left, right, pivot);
        recQuickSort(left, partition - 1); // sort left side
        recQuickSort(partition + 1, right); // sort right side
    }
} // end recQuickSort()
```




```
int partitie(int left, int right, double pivot)
{
    int leftPtr = left-1;           // left  (after ++)
    int rightPtr = right;           // right-1 (after --)
    while (true)
    {
        // find bigger item
        while (theArray[++leftPtr] < pivot)
            ; // (nop)

        // find smaller item
        while (rightPtr > 0 && theArray[--rightPtr] > pivot)
            ; // (nop)
```

```
if (leftPtr >= rightPtr)           // if pointers cross,  
    break;                         // partition done  
else                               // not crossed, so  
    swap(&theArray[leftPtr], &theArray[rightPtr]);  
                                   // swap elements  
    } // end while(true)  
swap(&theArray[leftPtr], &theArray[right]);  
                                   // restore pivot  
return leftPtr;                   // return pivot location  
} // end partitie()
```



```
void swap(int *index1, int *index2)
    // swap two elements
{
    int temp = *index1;
    *index1 = *index2;
    *index2 = temp;
} // end swap()
```



Este posibil să renunțăm la testul de sfârșit de vector în prima buclă *while* interioară

Acest test era *leftPtr < right*

Testul împiedica indexul *leftPtr* să treacă de capătul din dreapta al vectorului, în cazul când nu exista niciun element mai mare decât pivotul

Se poate renunța la acest test pentru că, prin selectarea ultimului element ca pivot, *leftPtr* se va opri întotdeauna la el

Testul similar pentru *rightPtr* din cel de-a doua buclă *while* rămâne necesar

Vezi demonstrația Quicksort

0	1	2	3	4	5	6	7	8	9	10	11
90	100	20	60	80	110	120	40	10	30	50	70

1

30	10	20	60	40	50	70	120	80	100	90	110
----	----	----	----	----	----	----	-----	----	-----	----	-----

2

30	10	20	40	50	60	70	120	80	100	90	110
----	----	----	----	----	----	----	-----	----	-----	----	-----

3

30	10	20	40	50	60	70	120	80	100	90	110
----	----	----	----	----	----	----	-----	----	-----	----	-----

4

