Quick Sort

2015

Algoritmul

QuickSort ca Divide and Conquer

```
void QS(int Left, int Right)
{ int q;  // (Right - Left + 1) reprezintă dimensiunea vectorului
  if (Right - Left + 1) == 1)
    return;  // nu facem nimic
  else  // Urmăatorul apel corespunde pasului Divide.
    {
      Partition(Left,Right,q);  // Returnează pivotul in q.
       // Pivotul este pus la locul său final.
      QS(Left, q-1);  // 1 apel recursiv
      QS(Q+1, Right);  // 2 apel recursiv
       // Pentru pasul Combine nu trebuie să facem nimic.
     }
}
```

Procedura Partition

Pentru a sorta vectorul $A[1..n]$ apelul principal va fi QS(1,n);

Despre partiții la Quick Sort

Partiția Hoare

Procedura Partition

Procedura de partiționare care implementează algoritmul descris mai sus pe subvectorul $A[Left..Right]$ este următoarea:

Partiția Hoare

Procedura Partition

Procedura de partiționare care implementează algoritmul descris mai sus pe subvectorul $A[Left..Right]$ este următoarea:

```
void Partition(int Left, int Right)
{ int iLeft;  // iLeft = indicele curent pentru parcurgerea (2a)
   int iRight;  // iRight = indicele curent pentru parcurgerea (2b)
   iLeft = Left; iRight = Right;          // inițializarea indicilor curenți pentru parcurgeri
   int x = A[(Left + Right)/2];                        // algerea pivotului în poziție mediană
   do {                                                                              // partiția
      while (A[iLeft] < x)                                                              // (2a)
         iLeft++;
      while (A[iRight] > x)                                                             // (2b)
         iRight--;
      if (iLeft < iRight)                                                                // (3)
         { Interschimbă(A[iLeft], A[iRight]);
         iLeft++;
         iRight--;
          }
   } while (iLeft < iRight);
}
```

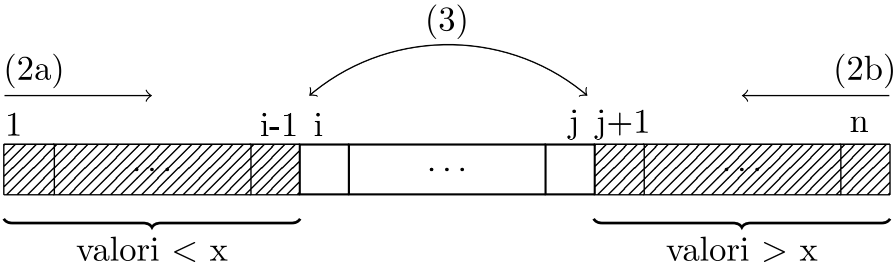


Figure : Procedura de partiționare.

Exemplu:

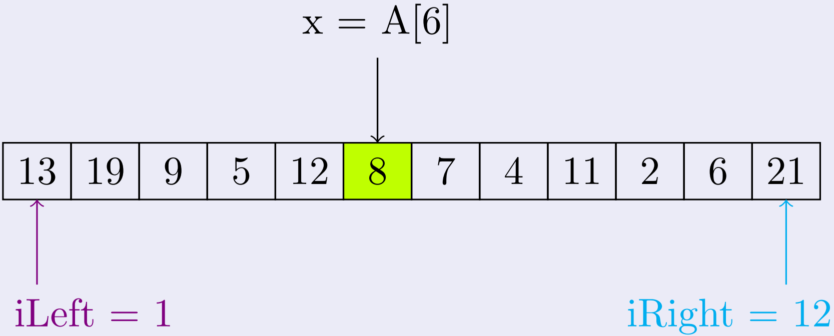


Figure : vectorul A inițial

```
x = A[(Left+Right)/2];
iLeft = 1; iRight = 12;
```

Exemplu:

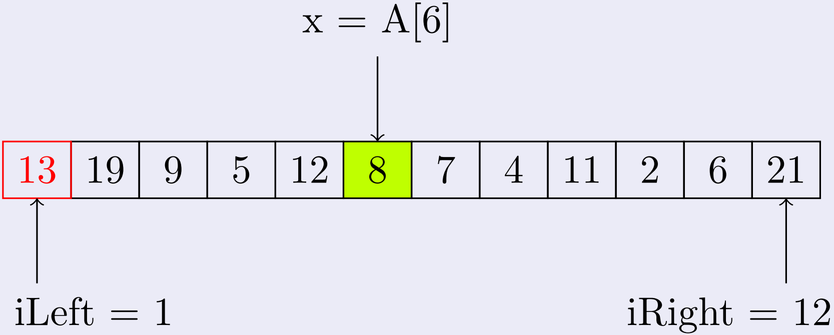


Figure : (2a)

`A[1]` $\geq$ `x`, deci nu putem avansa cu `iLeft`.

Exemplu:

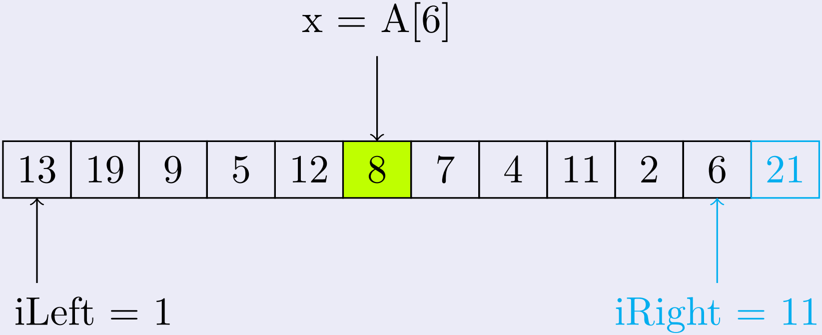


Figure : (2b)

`A[12] > x`, deci `iRight = iRight-1;`.

Exemplu:

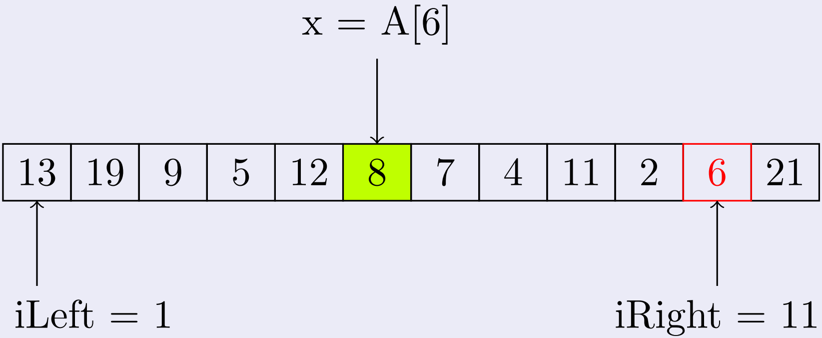


Figure : (2b)

`A[11]` $\leq$ `x`, deci `iRight` nu mai avansează.

Exemplu:

Figure : (3)

Trebuie să interschimbăm `A[1]` cu `A[11]`.

Exemplu:

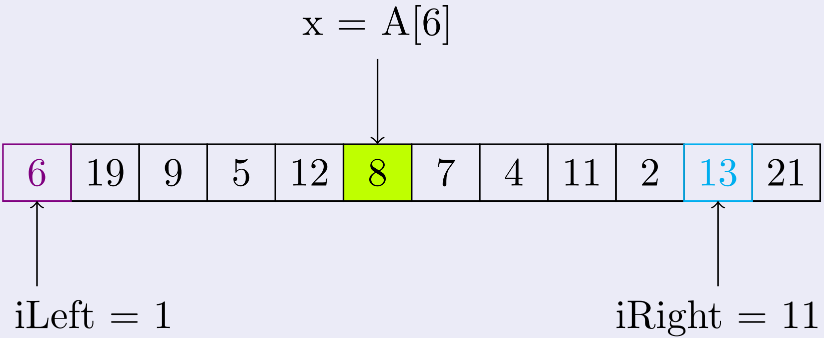


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

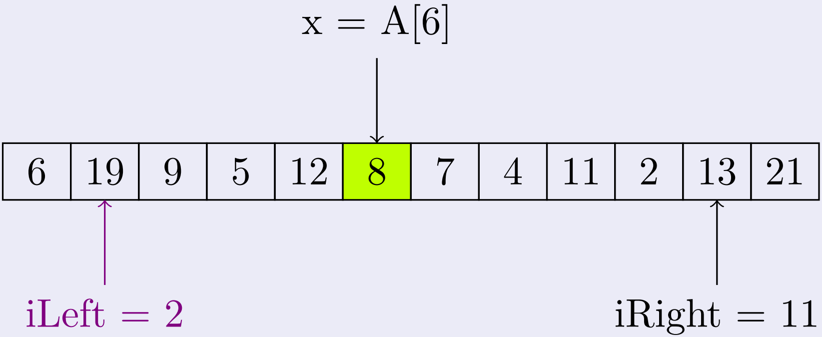


Figure : (3)

```
iLeft = iLeft+1;
```

Exemplu:

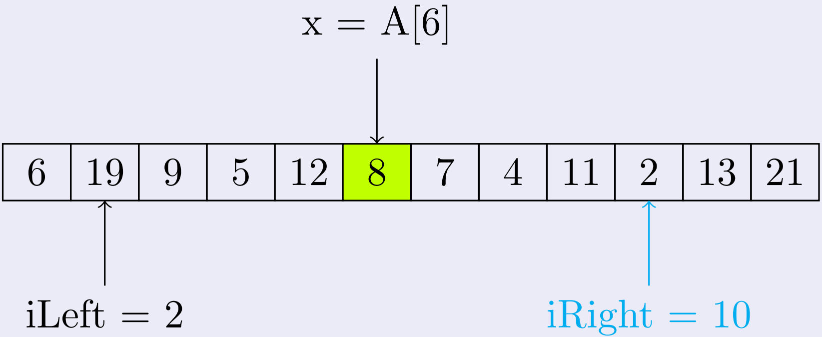


Figure : (3)

```
iRight = iRight-1;
```

Exemplu:

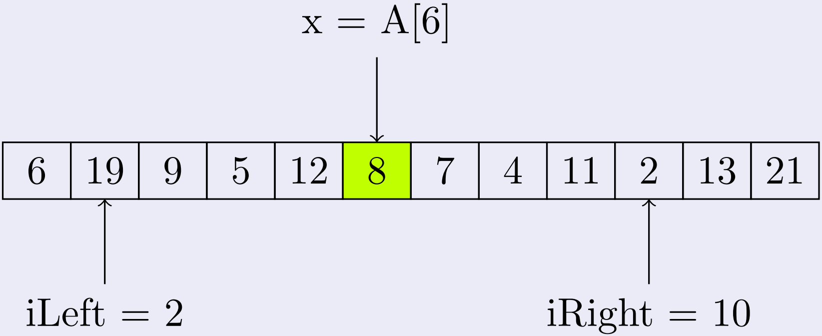


Figure : După prima iterație a ciclului `repeat`

Avem `2 = iLeft < iRight = 10`, deci reluăm procedura de partiție.

Exemplu:

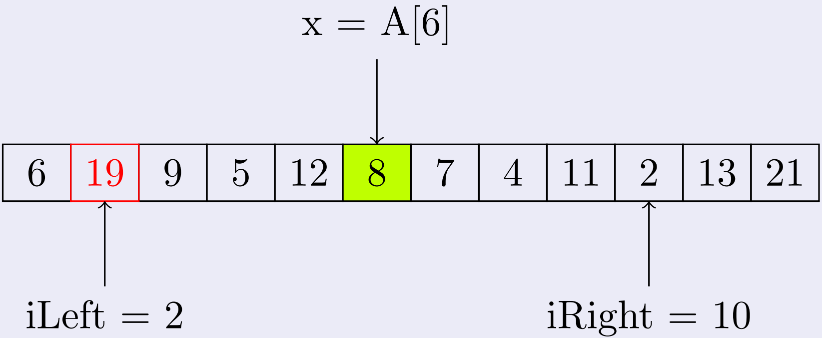


Figure : (2a)

`A[2]` $\geq$ `x`, deci nu putem avansa cu `iLeft`.

Exemplu:

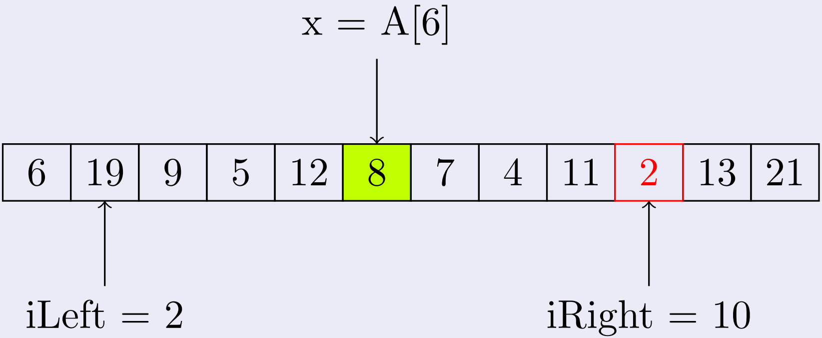


Figure : (2b)

`A[10]` $\leq$ `x`, deci nu putem avansa cu `iRight`.

Exemplu:

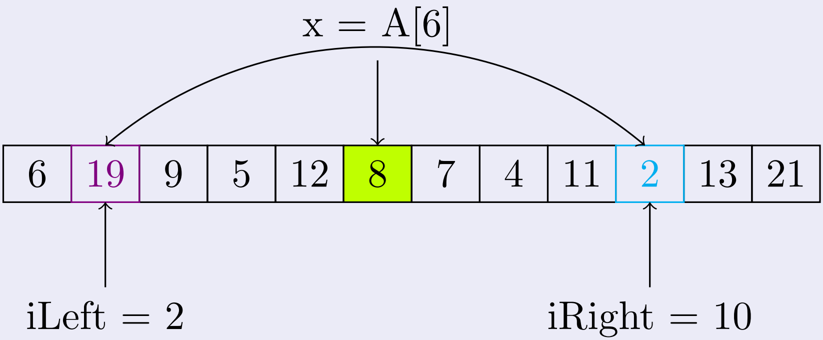


Figure : (3)

Trebuie să interschimbăm `A[2]` cu `A[10]`.

Exemplu:

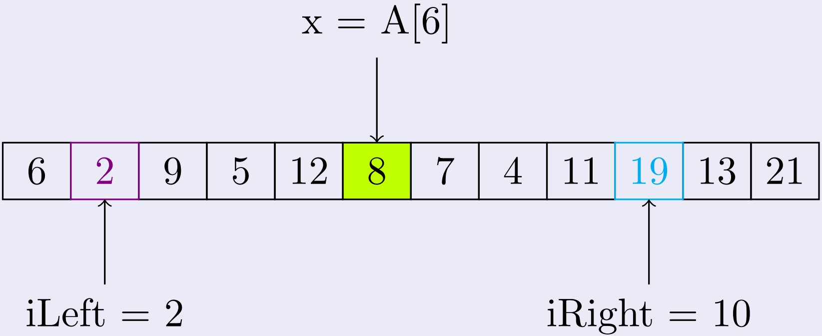


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

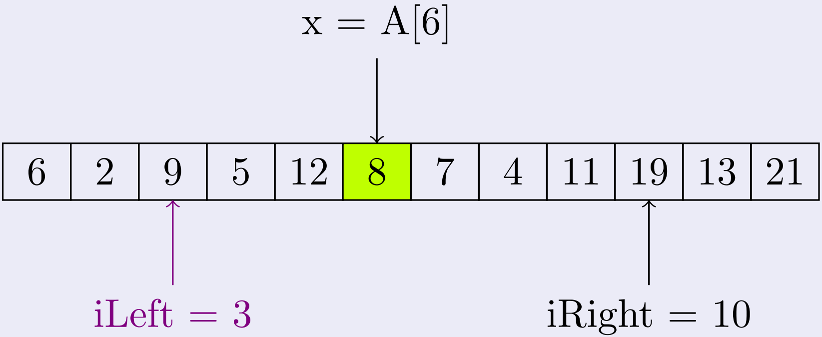


Figure : (3)

```
iLeft = iLeft+1;
```

Exemplu:

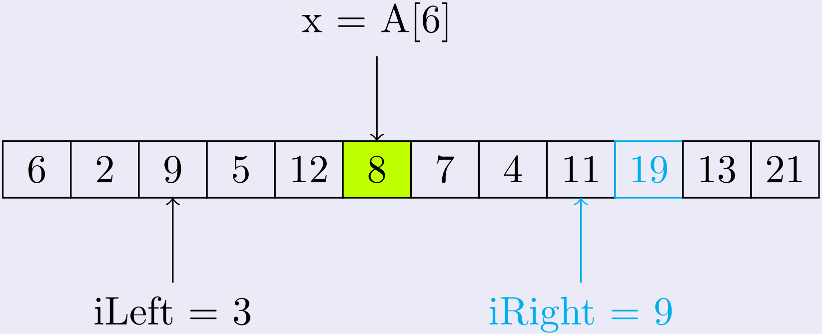


Figure : (3)

```
iRight = iRight-1;
```

Exemplu:

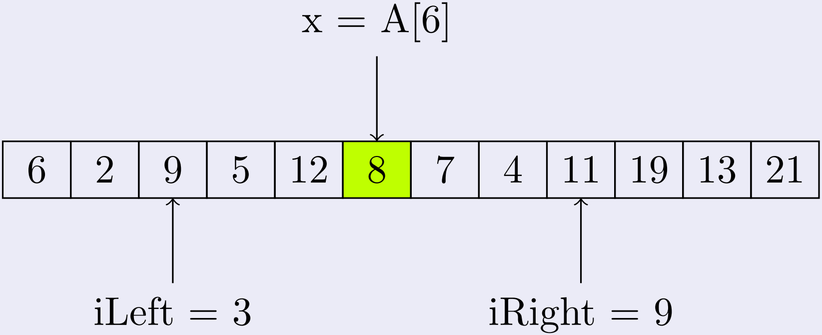


Figure : (Vectorul după a doua iterație a ciclului `repeat`)

Avem 3 `= iLeft < iRight =` 9, deci reluăm procedura de partiție.

Exemplu:

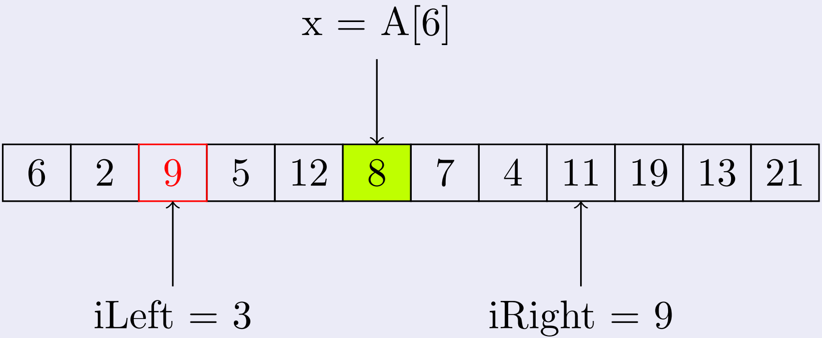


Figure : (2a)

`A[3]` $\geq$ `x`, deci nu putem avansa cu `iLeft`.

Exemplu:

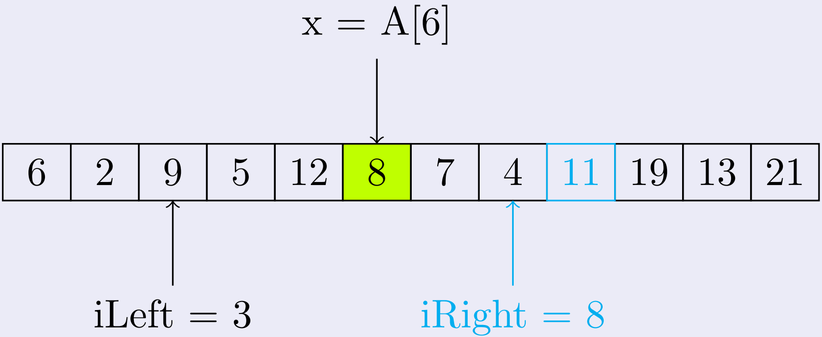


Figure : (2b)

`A[9] > x`, deci `iRight = iRight - 1;`.

Exemplu:

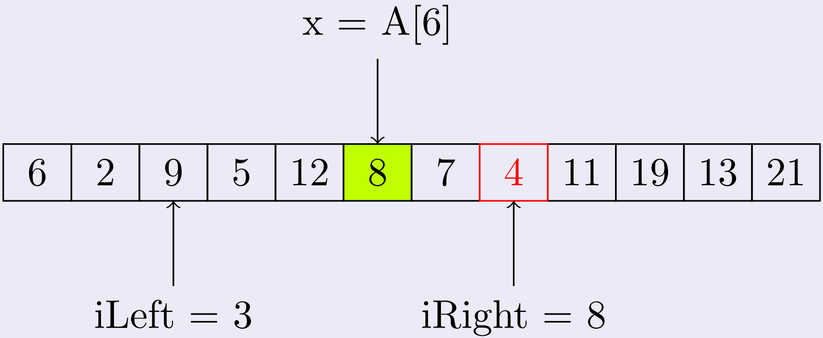


Figure : (2b)

`A[8]` $\leq$ `x`, deci nu putem avansa cu `iRight`.

Exemplu:

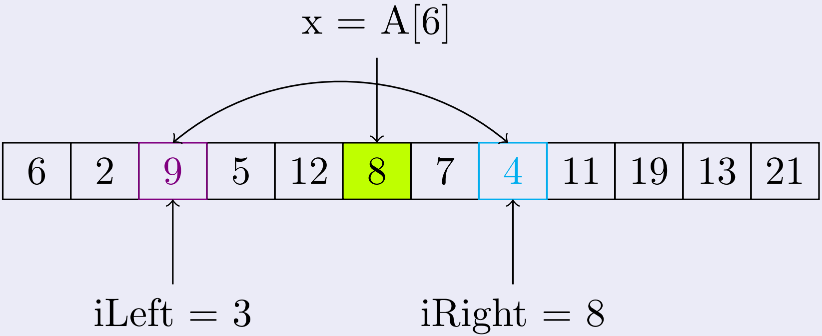


Figure : (2b)

Trebuie să interschimbăm `A[3]` cu `A[8]`.

Exemplu:

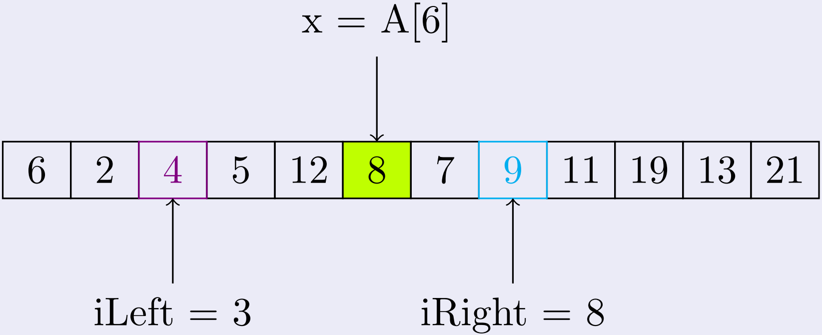


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

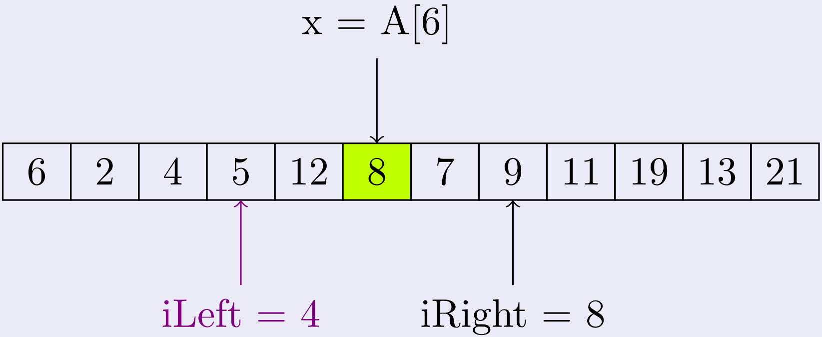


Figure : (3)

```
iLeft = iLeft+1;
```

Exemplu:

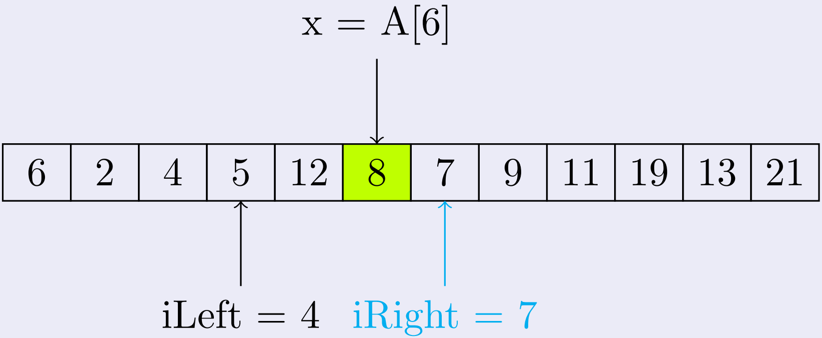


Figure : (3)

```
iRight = iRight-1;
```

Exemplu:

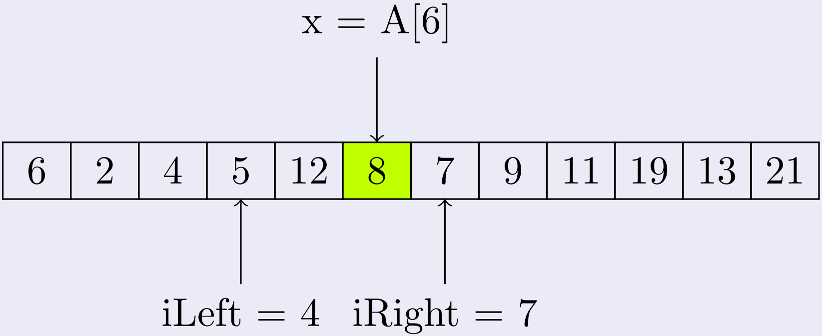


Figure : (Vectorul după a treia iterație a ciclului `repeat`)

Avem `4 = iLeft < iRight = 7`, deci reluăm procedura de partiție.

Exemplu:

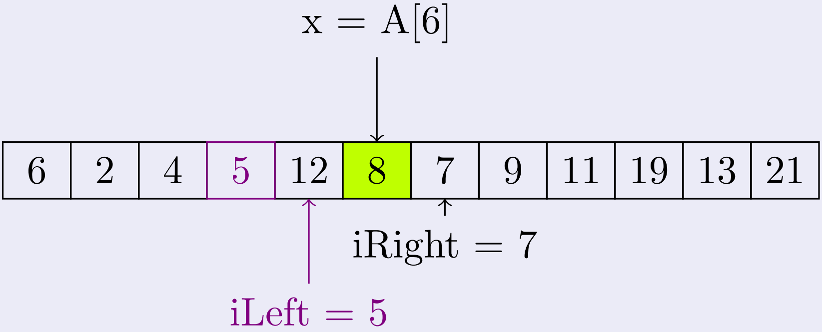


Figure : (2a)

`A[4] < x`, deci `iLeft = iLeft + 1;`.

Exemplu:

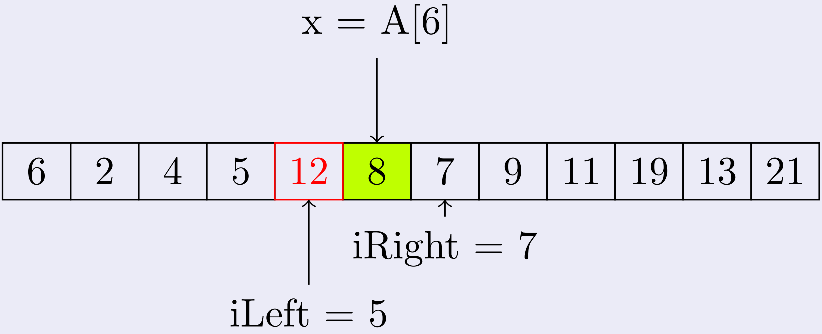


Figure : (2a)

`A[5]` $\geq$ `x`, deci nu putem avansa cu `iLeft`.

Exemplu:

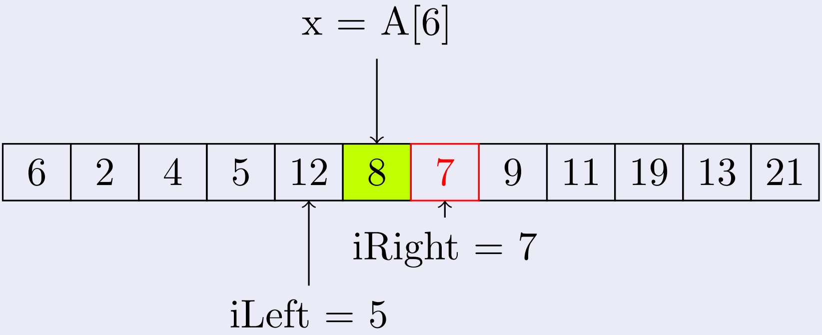


Figure : (2b)

`A[7] < x`, deci nu putem avansa cu `iRight`.

Exemplu:

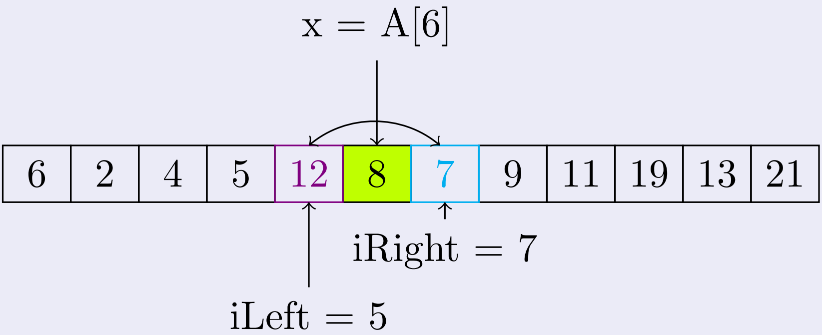


Figure : (3)

Trebuie să interschimbăm `A[5]` cu `A[7]`.

Exemplu:

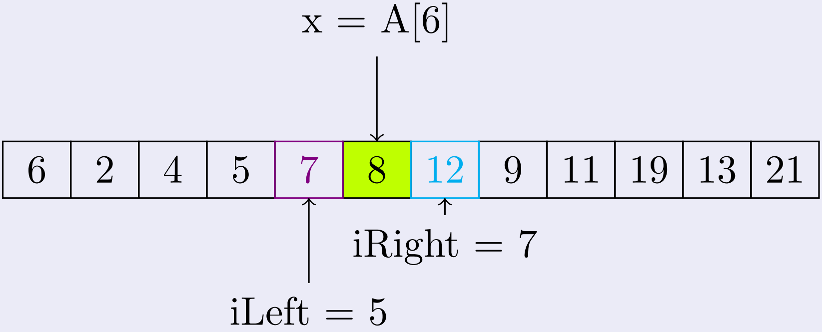


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

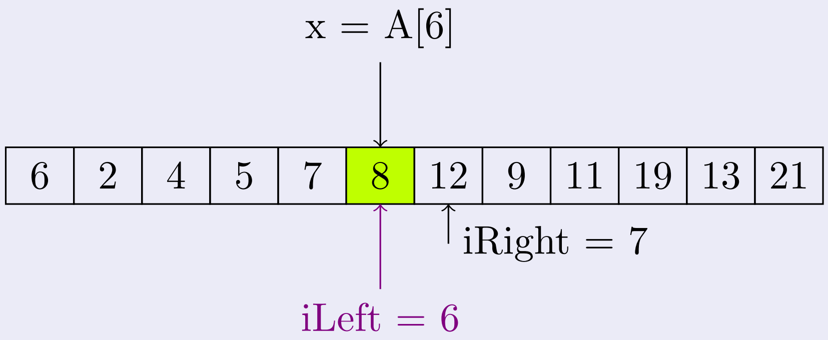


Figure : (3)

```
iLeft = iLeft+1;
```

Exemplu:

$$x = A[6]$$

6	2	4	5	7	8	12	9	11	19	13	21

$$iRight = 6$$

$$iLeft = 6$$

Figure : (3)

```
iRight = iRight-1;
```

Exemplu:

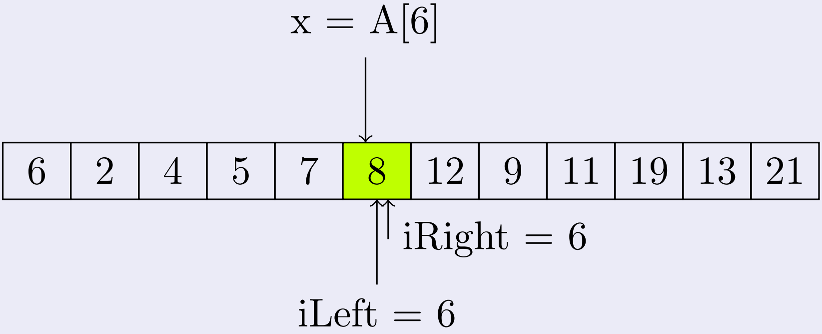


Figure : Vectorul după a patra iterație a ciclului `repeat`

Avem 6 = `iLeft` < `iRight` = 6, deci am terminat procedura de partiție.

Exemplu:

6	2	4	5	7	8	12	9	11	19	13	21

Figure : Final

Am obținut partiția: `A[1..6]`, `A[7..12]`.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partiționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partiționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partiționare (parcurgerea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partiționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partiționare (parcurgerea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.
- Executăm (2b), adică parcurgem de la dreapta la stânga vectorul până la primul indice j pentru care $A[j] \leq x$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partiționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partiționare (parcurgerea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.
- Executăm (2b), adică parcurgem de la dreapta la stânga vectorul până la primul indice j pentru care $A[j] \leq x$.
- Pasul (3) devine:

  `if` $Left < j$ `then`

  `interschimbă (A[Left], A[j])`

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atâta timp cât mai există componente în vector care nu au fost comparate cu pivotul.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atâta timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atâta timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.
- Fie *loc* indicele lui A ce va conține pe x. Avem atunci:
  $A[k] \leq x \; \forall k \in [1..loc-1]$
  $A[k] \geq x \; \forall k \in [loc+1..n]$

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atâta timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.
- Fie *loc* indicele lui A ce va conține pe x. Avem atunci:
  $A[k] \leq x \; \forall k \in [1..loc - 1]$
  $A[k] \geq x \; \forall k \in [loc + 1..n]$
- Deci subintervalele ce trebuie procesate în continuare sunt $A[1..loc - 1]$ și $A[loc + 1..n]$.

Procedura `Partition2`

Dăm mai jos noua procedură de partiționare.

- La parcurgeri ea va lăsa pe loc valorile egale cu pivotul.
- Indicele *loc* ține minte tot timpul componenta pe care se află valoarea pivot.
- La sfârșit o transmite procedurii apelante pentru a putea fi calculate noile capete ale subvectorilor rezultați.

```
void Partition2 (int Left, int Right, int loc)
{  // loc este indicele pe care se va plasa în final valoarea x = A[Left]
    // inițializarea indicilor i și j pentru parcurgerile de la stânga la dreapta, respectiv de la
    //  dreapta la stânga
   int i = Left, j= Right;
   loc = Left;                                           // inițializarea indicelui loc
   while (i < j)
      {  // parcurgerea de la dreapta la stânga, urmată de interschimbare
      while ((A[loc] <= A[j]) && (j != loc))
         j--;
      if (A[loc] > A[j])
         { Interschimbă(A[loc], A[j])
         loc = j;
         }
       // parcurgerea de la stânga la dreapta, urmată de interschimbare
      while ((A[i] <= A[loc]) && (i != loc))
         i++;
      if (A[i] > A[loc])
         { Interschimbă (A[loc], A[i])
         loc = i;
         }
      }
}
```

Partiția Lomuto

Avantaje față de `Partition2`

- Pivot într-o extremitate, dar parcurge restul, făcând **separarea valorilor, într-un singur sens**.

Partiția Lomuto

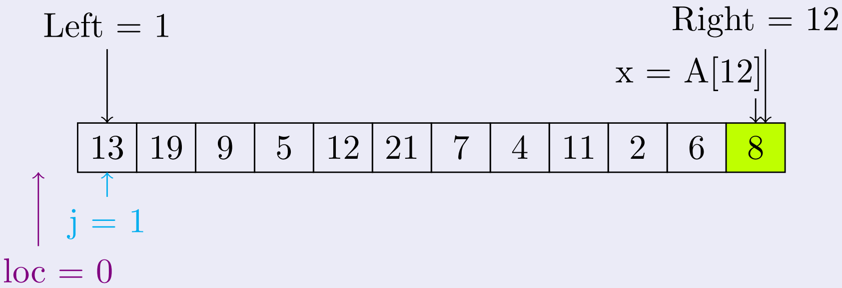
Avantaje față de `Partition2`

- Pivot într-o extremitate, dar parcurge restul, făcând **separarea valorilor, într-un singur sens**.
- Reducerea numărului de interschimbări: Lomuto face $m + 1$, față de $2m$ făcute de `Partition2`.

Partiția Lomuto

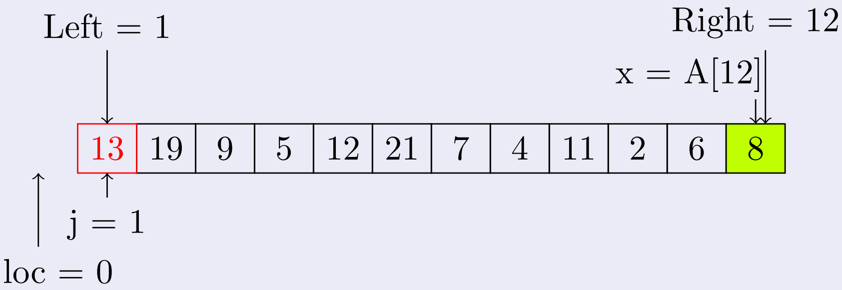
```
void PartitionLomuto(int Left, int Right, int loc)
{ int x = A[Right];                    // algerea pivotului în într-o extremitate
   // loc este indicele pe care se va plasa în final valoarea x = A[Right]
  loc = Left-1;
// inițializare indice j pentru parcurgerea de la stânga la dreapta (un singur sens)
  int j = Left;
  while (j <= Right)
    { if (A[j] <= x)
      { loc++;
      Interschimbă(A[loc],A[j]);
      }
    j++;
    }
  if (loc >= Right)
    loc--;
}
```

Exemplu:




```
x = A[Right];
loc = Left-1; j = Left
```

Exemplu:



`A[1] > x.`

Exemplu:

Left $= 1$

Right $= 12$

$x = A[12]$

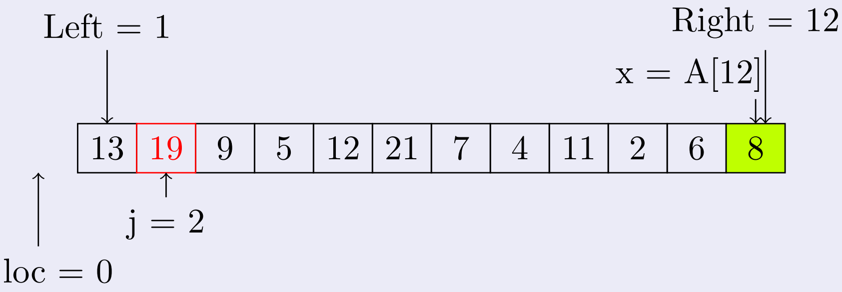
13	19	9	5	12	21	7	4	11	2	6	8

$j = 2$

loc $= 0$

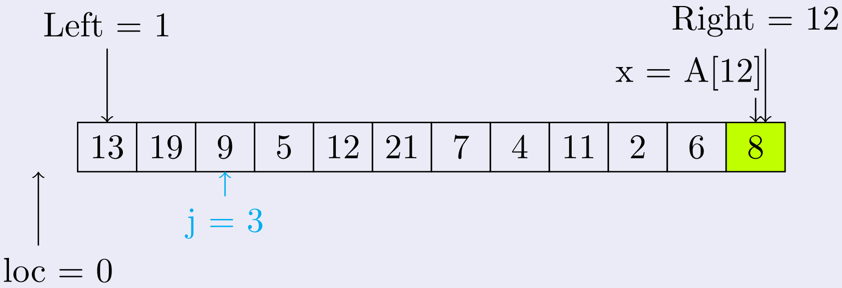
Avansam cu `j`.

Exemplu:



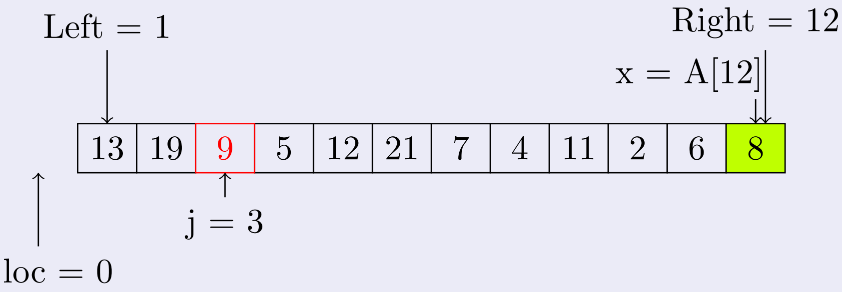
`A[2] > x.`

Exemplu:



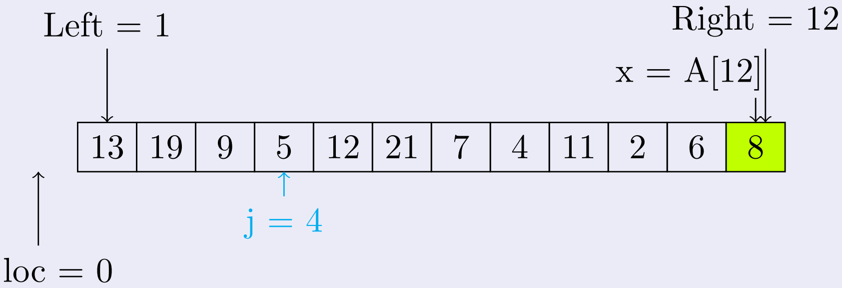
Avansam cu j.

Exemplu:



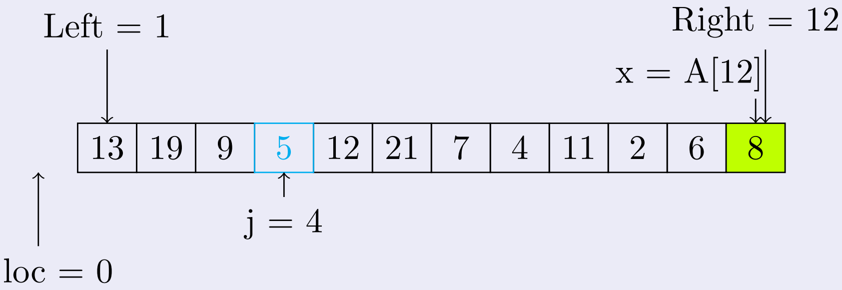
`A[3] > x.`

Exemplu:



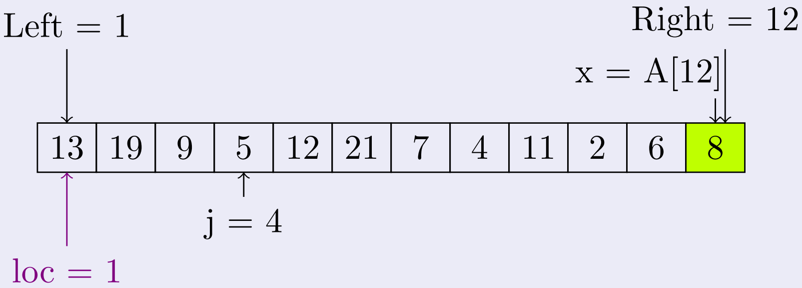
Avansam cu j.

Exemplu:



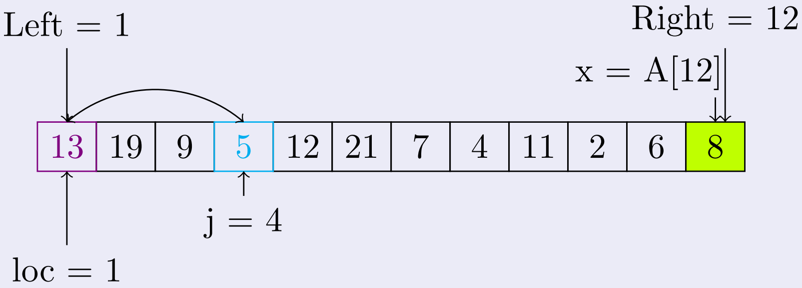
$\texttt{A[4]} \leq \texttt{x}$.

Exemplu:



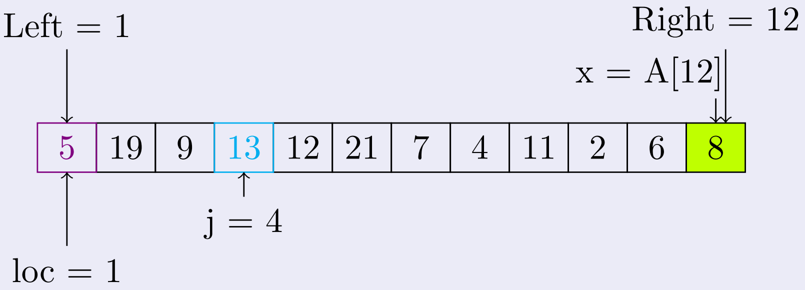
Avansăm cu `loc`.

Exemplu:



Trebuie să interschimbăm `A[1]` cu `A[4]`.

Exemplu:



Vectorul după interschimbare.

Exemplu:

$\text{Left} = 1$

$\text{Right} = 12$

$x = A[12]$

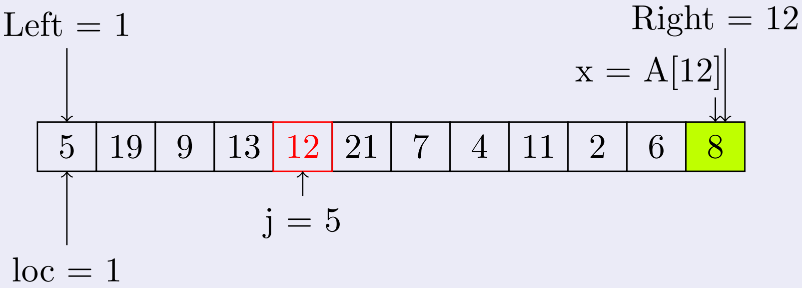
5	19	9	13	12	21	7	4	11	2	6	8

$j = 5$

$\text{loc} = 1$

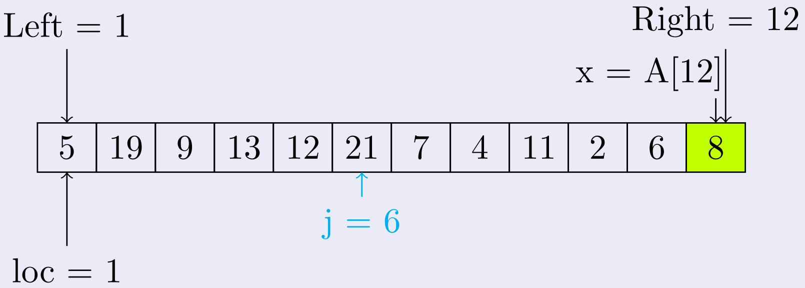
Avansăm cu j.

Exemplu:



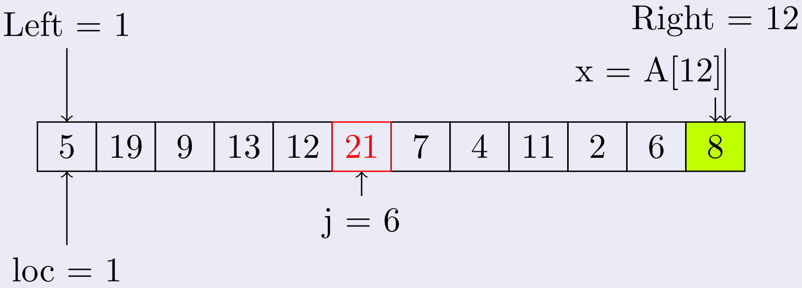
`A[5] > x.`

Exemplu:



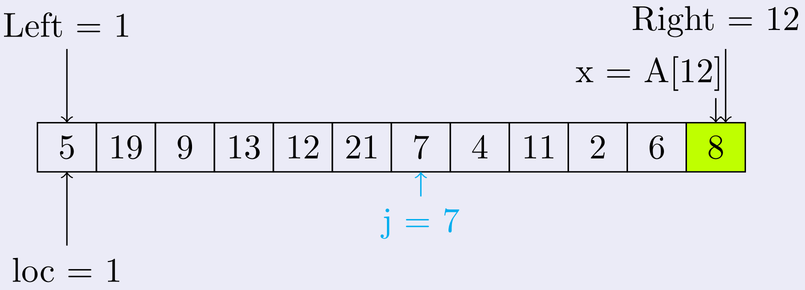
Avansăm cu j.

Exemplu:



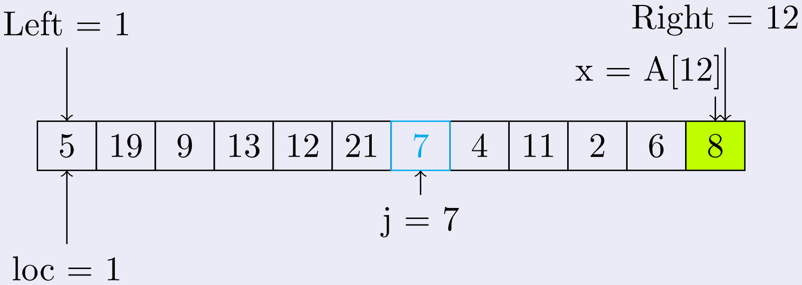
`A[6] > x.`

Exemplu:



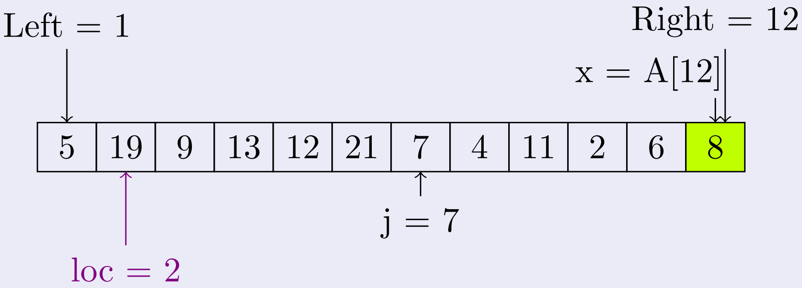
Avansăm cu j.

Exemplu:



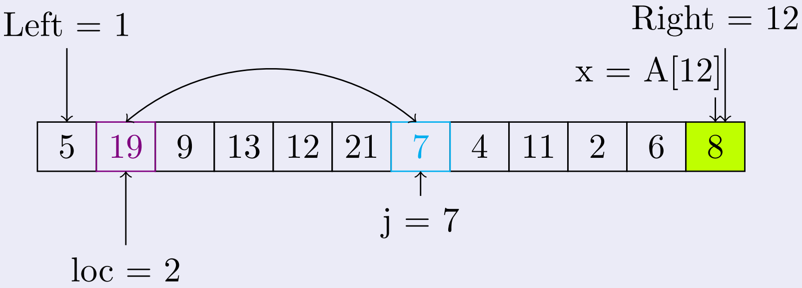
`A[7]` $\leq$ `x`.

Exemplu:



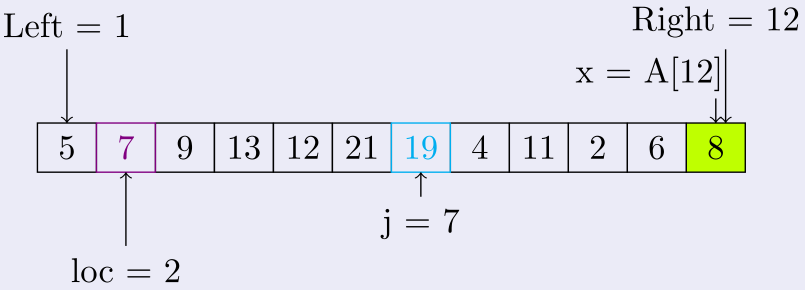
Avansăm cu `loc`.

Exemplu:



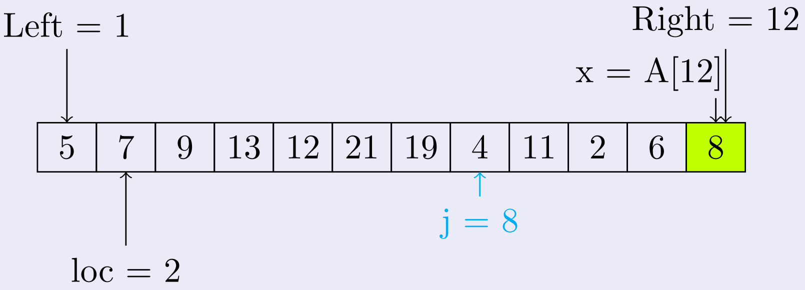
Trebuie să interschimbăm `A[2]` cu `A[7]`.

Exemplu:



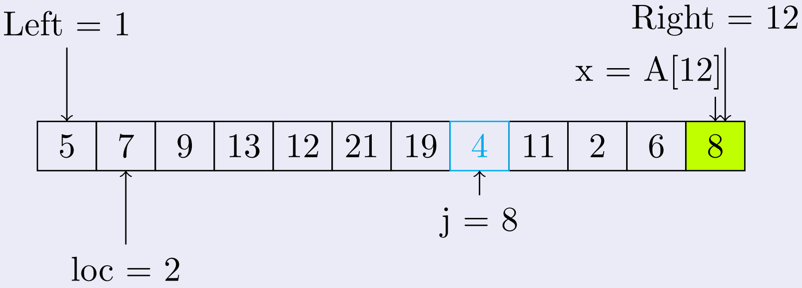
Vectorul după interschimbare.

Exemplu:



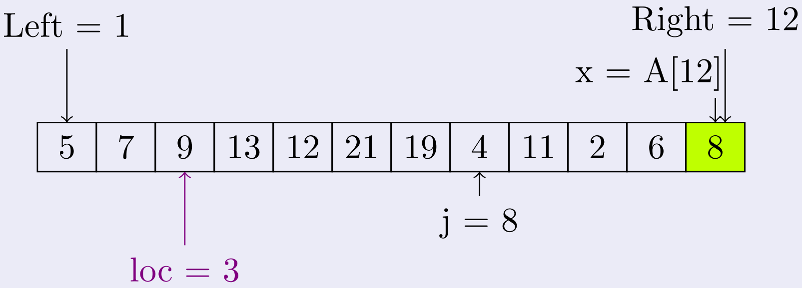
Avansăm cu j.

Exemplu:



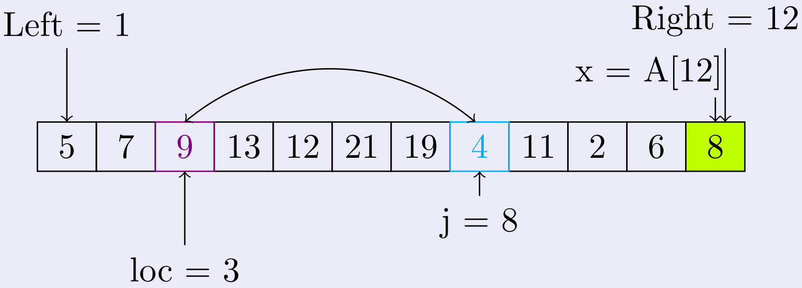
`A[8]` $\leq$ `x`.

Exemplu:



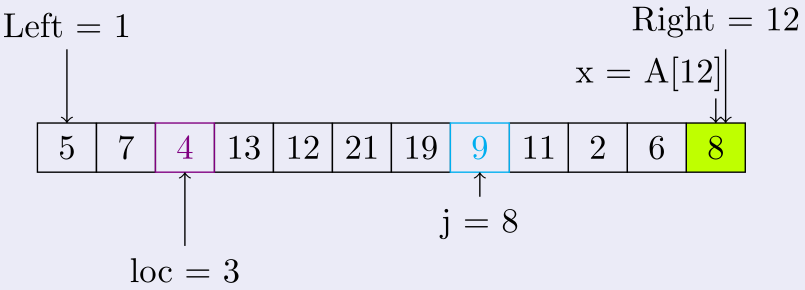
Avansăm cu `loc`.

Exemplu:



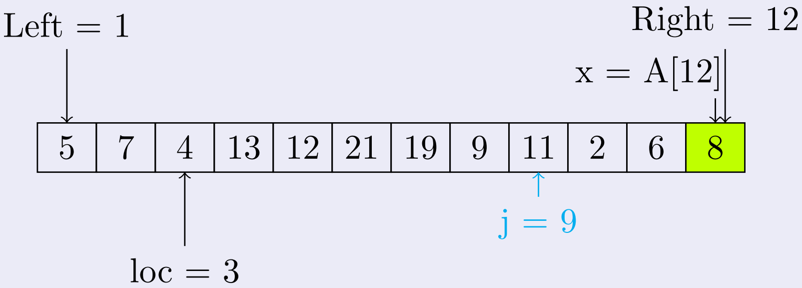
Trebuie să interschimbăm `A[3]` cu `A[8]`.

Exemplu:



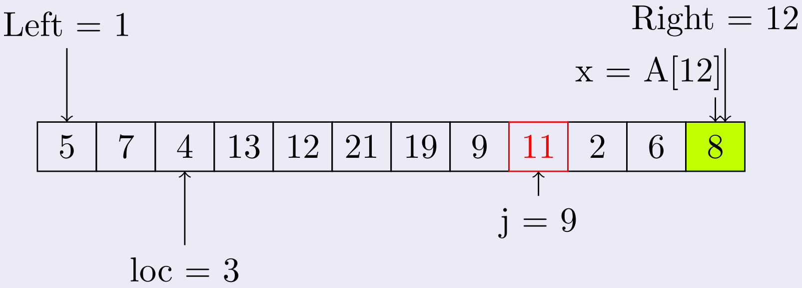
Vectorul după interschimbare.

Exemplu:



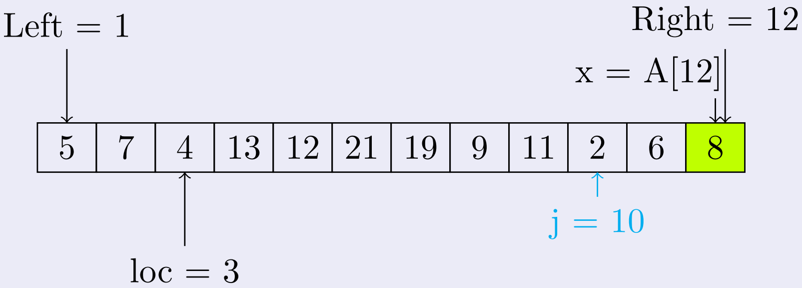
Avansăm cu j.

Exemplu:



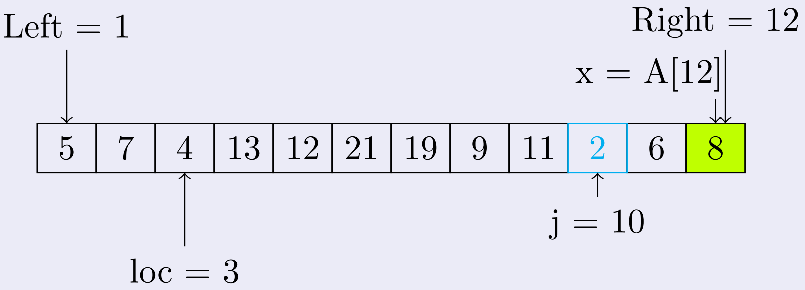
`A[9] > x.`

Exemplu:



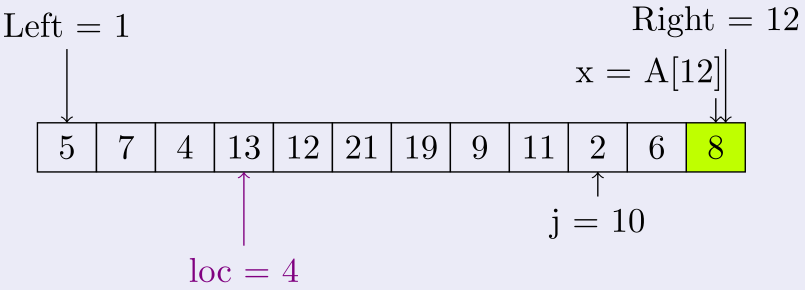
Avansăm cu j.

Exemplu:



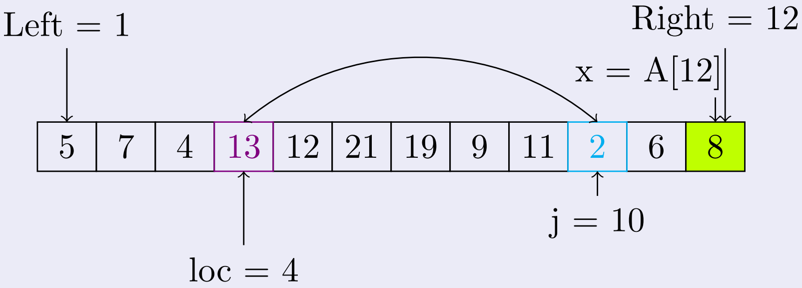
`A[10]` $\leq$ `x`.

Exemplu:



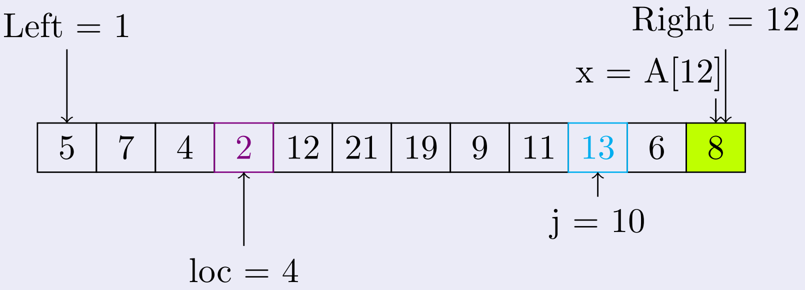
Avansăm cu `loc`.

Exemplu:



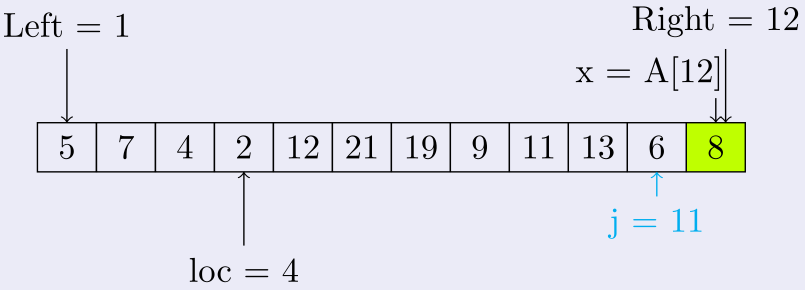
Trebuie să interschimbăm `A[4]` cu `A[10]`.

Exemplu:



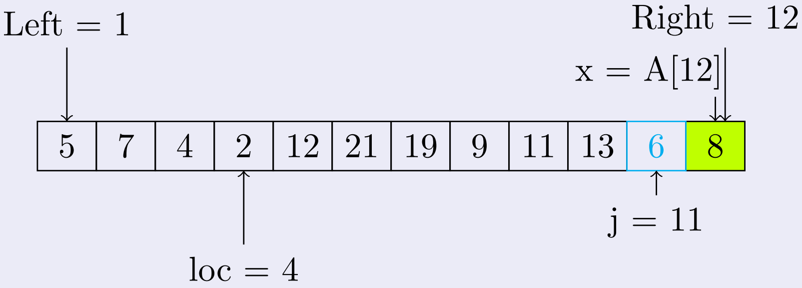
Vectorul după interschimbare.

Exemplu:



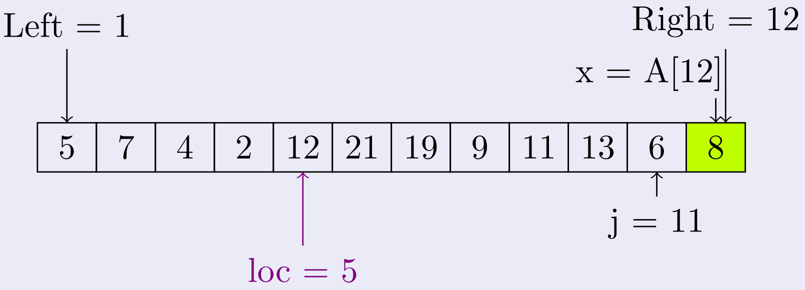
Avansăm cu j.

Exemplu:



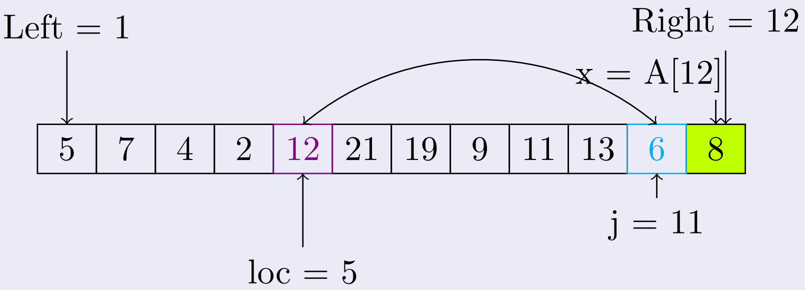
`A[11]` $\leq$ `x`.

Exemplu:



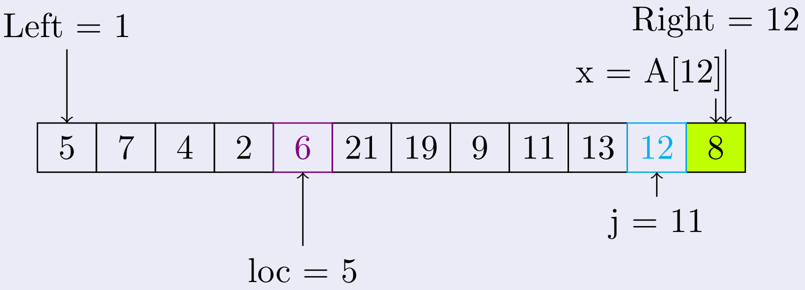
Avansăm cu `loc`.

Exemplu:



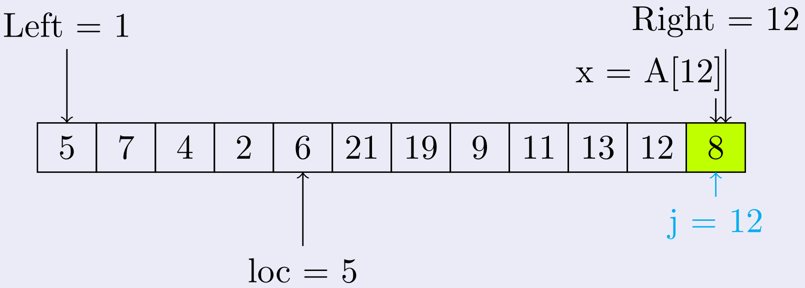
Trebuie să interschimbăm `A[5]` cu `A[11]`.

Exemplu:



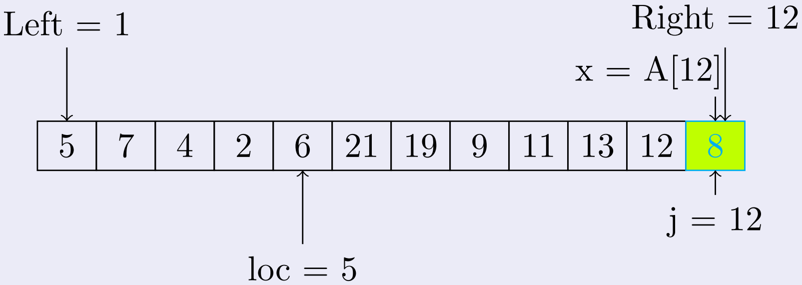
Vectorul după interschimbare.

Exemplu:



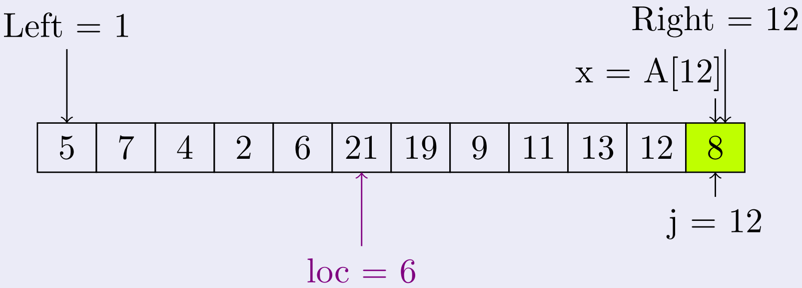
Avansăm cu j.

Exemplu:



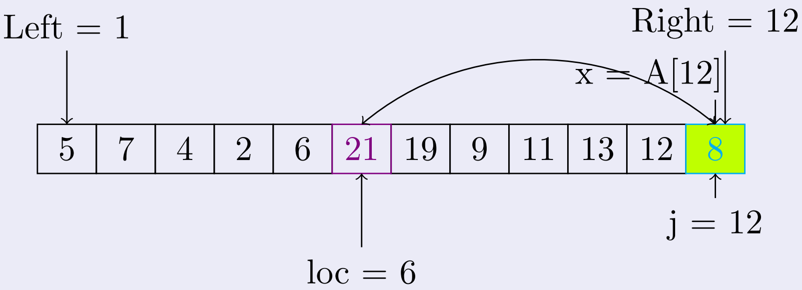
`A[8]` $\leq$ `x`.

Exemplu:



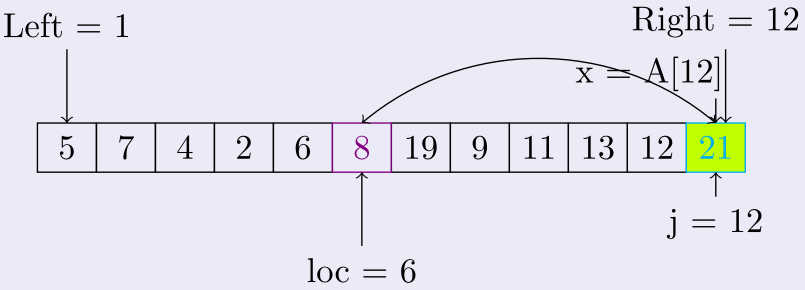
Avansăm cu `loc`.

Exemplu:



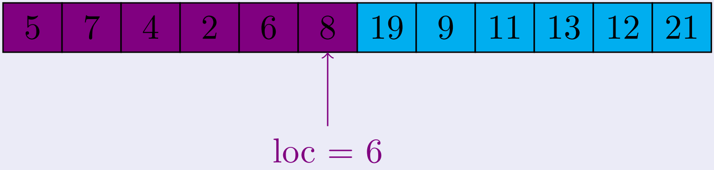
Trebuie să interschimbăm `A[6]` cu `A[12]`.

Exemplu:



Vectorul după interschimbare. Algoritmul se termină.

Exemplu:



A 6-a valoare din `A` este pe poziția `loc = 6`.Am obținut partiția: `A[1..6]`, `A[7..12]`.

Aplicație a procedurii de partiționare la găsirea medianei

Problema găsirii medianei

Mediana unui vector este acea valoare dintre componentele sale care este mai mare decât jumătate dintre componente și mai mică decât cealaltă jumătate.
Dacă am sorta întâi vectorul, atunci mediana s-ar găsi la mijlocul vectorului. Vrem să găsim însă această valoare **fără a sorta întregul vector**.

Aplicație a procedurii de partiționare la găsirea medianei

Problema găsirii medianei

Mediana unui vector este acea valoare dintre componentele sale care este mai mare decât jumătate dintre componente și mai mică decât cealaltă jumătate.
Dacă am sorta întâi vectorul, atunci mediana s-ar găsi la mijlocul vectorului. Vrem să găsim însă această valoare **fără a sorta întregul vector**.

Problema generalizată

Problema se generalizează la găsirea valorii a k-a dintr-un vector, cea care este mai mare decât k-1 componente și mai mică decât $n - k$ componente, pe scurt, cea care ar ocupa locația $A[k]$ în vectorul sortat, fără a sorta întregul vector.

Aplicație a procedurii de partiționare la găsirea medianei

C. A. R. Hoare a propus un algoritm care se bazează pe procedura de partiționare a sortării rapide.
Se aplică procedura de partiționare pe întregul vector, deci $Left = 1$ și $Right = n$, cu valoarea pivot $x = A[k]$.
Indicii i și j cu care se termină partiția au proprietățile:

- $i > j$ (cele două parcurgeri s-au întâlnit)
- $A[s] \leq x, \ \forall s < i$
- $A[s] \geq x, \ \forall s > j$

Avem trei cazuri posibile pentru indicele k în raport cu indicii i și j:

Avem trei cazuri posibile pentru indicele k în raport cu indicii i și j:

(1) $j < i < k$

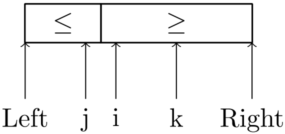


Figure : Limita unde se termină cele două partiții este în stânga lui k (din cauză că valoarea pivot x a fost mai mică decât valoarea căutată). Se reia partiționarea pe subvectorul $A[i..Right]$.

Avem trei cazuri posibile pentru indicele k în raport cu indicii i și j:

(2) $k < j < i$

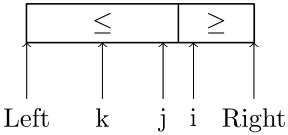


Figure : Cazul simetric lui (1). Limita unde se termină partițiile este în dreapta lui k. Se reia partiționarea pe subvectorul $A[Left..j]$.

Avem trei cazuri posibile pentru indicele k în raport cu indicii i și j:

(3) $j < k < i$

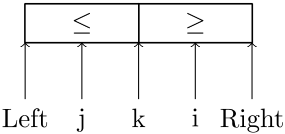


Figure : Valoarea de pe componenta k este cea căutată, deoarece acum în stânga avem $k-1$ valori mai mici decât ea, iar în dreapta $n-k$ valori mai mari.

Procedura `Find`

Procedura de partiționare se reia pe subintervalele ce conțin indicele k până când se ajunge în cazul (3).

Procedura `Find` implementează algoritmul descris mai sus, presupunând că procedura `Partition (Left, Right, i, j)` este complet modularizată și funcționează ca cea descrisă la sortarea rapidă.

Procedura Find

Procedura de partiționare se reia pe subintervalele ce conțin indicele k până când se ajunge în cazul (3).
Procedura `Find` implementează algoritmul descris mai sus, presupunând că procedura `Partition (Left, Right, i, j)` este complet modularizată și funcționează ca cea descrisă la sortarea rapidă.

```
void Find (int A[], int n, int k)
{ int x, i,j, Left = 1, Right = n;
     while (Left < Right)
        { x = A[k];
        Partition (Left, Right, i, j);
        if (j < k)
           Left = i;                    // se continuă pe subvectorul din dreapta
        if (k < i)
           Right = j;                   // se continuă pe subvectorul din stânga
        }
}
```

Procedura `Find2`

O modificare a algoritmului de mai sus se poate face în felul următor: în loc să așteptăm ca cele două parcurgeri să se întâlnească ($i > j$), să vedem când una dintre parcurgeri depășește indicele k și să reluăm partiționarea pe subintervalul corespunzător cu noua valoare pivot $x = A[k]$:

1. dacă $k \leq i < j$ reluăm partiționarea pe $A[Left..j]$ cu noul pivot;
2. dacă $i < j \leq k$ reluăm partiționarea pe $A[i..Right]$ cu noul pivot;

Procedura `Find2` implementează această idee.

Procedura Find2

O modificare a algoritmului de mai sus se poate face în felul următor: în loc să așteptăm ca cele două parcurgeri să se întâlnească ($i > j$), să vedem când una dintre parcurgeri depășește indicele k și să reluăm partiționarea pe subintervalul corespunzător cu noua valoare pivot $x = A[k]$:

1. dacă $k \leq i < j$ reluăm partiționarea pe $A[Left..j]$ cu noul pivot;
2. dacă $i < j \leq k$ reluăm partiționarea pe $A[i..Right]$ cu noul pivot;

Procedura `Find2` implementează această idee.

```
procedure void Find2 (int A[], int n, int k)
{ int x, i, j, Left = 1, Right = n;
   while (Left < Right)
      { x = A[k];
      i = Left; j = Right;
      while ((i <= k) && (k <= j))
         {
            while (A[i] < x) i++;
            while (x < A[j]) j--;
            if (i <= j)
               { Interschimbă (A[i], A[j]);
               i++; j--; }
         }
      if ((k < i) && (k != j)) Right = j;
      if ((j < k) && (k != i)) Left = i;
      }
}
```

Discuție finală

- Este adevărat că doar partiția 2 și Lomuto pun pivotul la locul lui final în vector.

Discuție finală

- Este adevărat că doar partiția 2 și Lomuto pun pivotul la locul lui final în vector.
- Dar, dacă vrem să aplicăm o partiție la problema găsirii medianei (generalizată la k), atunci:
  - Începem cu un pivot $x = A[k] =$ valoarea inițială de pe componenta k
  - Ne interesează, nu atât ca acest x să ajungă la locul lui final, ci ca in locația $A[k]$ să se succeadă, cât mai repede, valori cât mai bune, adică mai apropiate de valoarea $y =$ a k-a componenta a lui A sortat.

Discuție finală

- Este adevărat că doar partiția 2 și Lomuto pun pivotul la locul lui final în vector.
- Dar, dacă vrem să aplicăm o partiție la problema găsirii medianei (generalizată la k), atunci:
    - Începem cu un pivot $x = A[k] =$ valoarea inițială de pe componenta k
    - Ne interesează, nu atât ca acest x să ajungă la locul lui final, ci ca in locația $A[k]$ să se succeadă, cât mai repede, valori cât mai bune, adică mai apropiate de valoarea $y =$ a k-a componenta a lui A sortat.
- De aceea, mai sus:
    - se aplică Hoare, și nu una din celelalte 2 partiții.

Discuție finală

- Este adevărat că doar partiția 2 și Lomuto pun pivotul la locul lui final în vector.
- Dar, dacă vrem să aplicăm o partiție la problema găsirii medianei (generalizată la k), atunci:
  - Începem cu un pivot $x = A[k] =$ valoarea inițială de pe componenta k
  - Ne interesează, nu atât ca acest x să ajungă la locul lui final, ci ca in locația $A[k]$ să se succeadă, cât mai repede, valori cât mai bune, adică mai apropiate de valoarea $y =$ a k-a componenta a lui A sortat.
- De aceea, mai sus:
  - se aplică Hoare, și nu una din celelalte 2 partiții.
  - este propusă `Find2` ca o îmbunătățire a lui `Find` (relativ la problema găsirii medianei). De îndată ce una din parcurgeri trece de indicele k, pe acea pozitie sunt șanse să fi apărut o nouă valoare, mai apropiată de cea finală, y, și se reia partiția cu acest nou pivot.

Problemă

De demonstrat formal corectitudinea și eficiența lui `Find2` (cel putin față de `Find`).

Problemă

De demonstrat formal corectitudinea și eficiența lui `Find2` (cel putin față de `Find`).

Altă problemă

De adaptat partiția 2 și/sau 3 la problema găsirii medianei, respectând idea de mai sus, de a ,,plimba" prin locația $A[k]$ valori cât mai ,,bune", cât mai ,,repede".