Recursivitate

Recursivitate

- O **funcție recursivă** este o funcție care se **autoapelează**, direct sau indirect.

Recursivitate

- O **funcție recursivă** este o funcție care se **autoapelează**, direct sau indirect.
- **Aplicații**

  – reducerea unei probleme la **subprobleme mai mici de același tip**

  – implementare de **relații de recurență**

    - corespondentul din matematică al recursivității

Recursivitate

- O **funcție recursivă** este o funcție care se **autoapelează**, direct sau indirect.
- **Aplicații**

  – reducerea unei probleme la **subprobleme mai mici de același tip**

  – implementare de **relații de recurență**

    - corespondentul din matematică al recursivității
- Important – **condiția de oprire**
  - ce știu să rezolv direct
  - primii termeni din relația de recurență

Exemplul 1
Factorial

Exemplul 1 – factorial

$$n! = \begin{cases} 1, & \text{dacă } n = 0 \\ n*(n-1)!, & \text{dacă n } \geq 1 \end{cases}$$

4! = 4*3!

Exemplul 1 – factorial

$$n! = \begin{cases} 1, \text{ dacă } n = 0 \\ n*(n-1)!, \text{ dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!

Exemplul 1 – factorial

$$n! = \begin{cases} 1, \text{ dacă } n = 0 \\ n*(n-1)!, \text{ dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!
2! = 2*1!

Exemplul 1 – factorial

$$n! = \begin{cases} 1, & \text{dacă } n = 0 \\ n*(n-1)!, & \text{dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!

Exemplul 1 – factorial

$$n! = \begin{cases} 1, & \text{dacă } n = 0 \\ n*(n-1)!, & \text{dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!
0! = 1

Adâncimea recursivității

Exemplul 1 – factorial

$$n! = \begin{cases} 1, & \text{dacă } n = 0 \\ n*(n-1)!, & \text{dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!   = 1*1 = 1
0! = 1

Adâncimea recursivității

Exemplul 1 – factorial

$$n! = \begin{cases} 1, \text{ dacă } n = 0 \\ n*(n-1)!, \text{ dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2!
2! = 2*1! = 2*1 = 2
1! = 1*0! = 1*1 = 1
0! = 1

Adâncimea recursivității

Exemplul 1 – factorial

$$n! = \begin{cases} 1, \text{ dacă } n = 0 \\ n * (n-1)!, \text{ dacă n } \geq 1 \end{cases}$$

4! = 4*3!
3! = 3*2! = 3*2 = 6
2! = 2*1! = 2*1 = 2
1! = 1*0! = 1*1 = 1
0! = 1

Adâncimea recursivității

Exemplul 1 – factorial

$$n! = \begin{cases} 1, \text{ dacă } n = 0 \\ n*(n-1)!, \text{ dacă n } \geq 1 \end{cases}$$

4! = 4*3!	= 4*6 = 24
3! = 3*2!	= 3*2 = 6
2! = 2*1!	= 2*1 = 2
1! = 1*0!	= 1*1 = 1
0! = 1	

Adâncimea recursivității

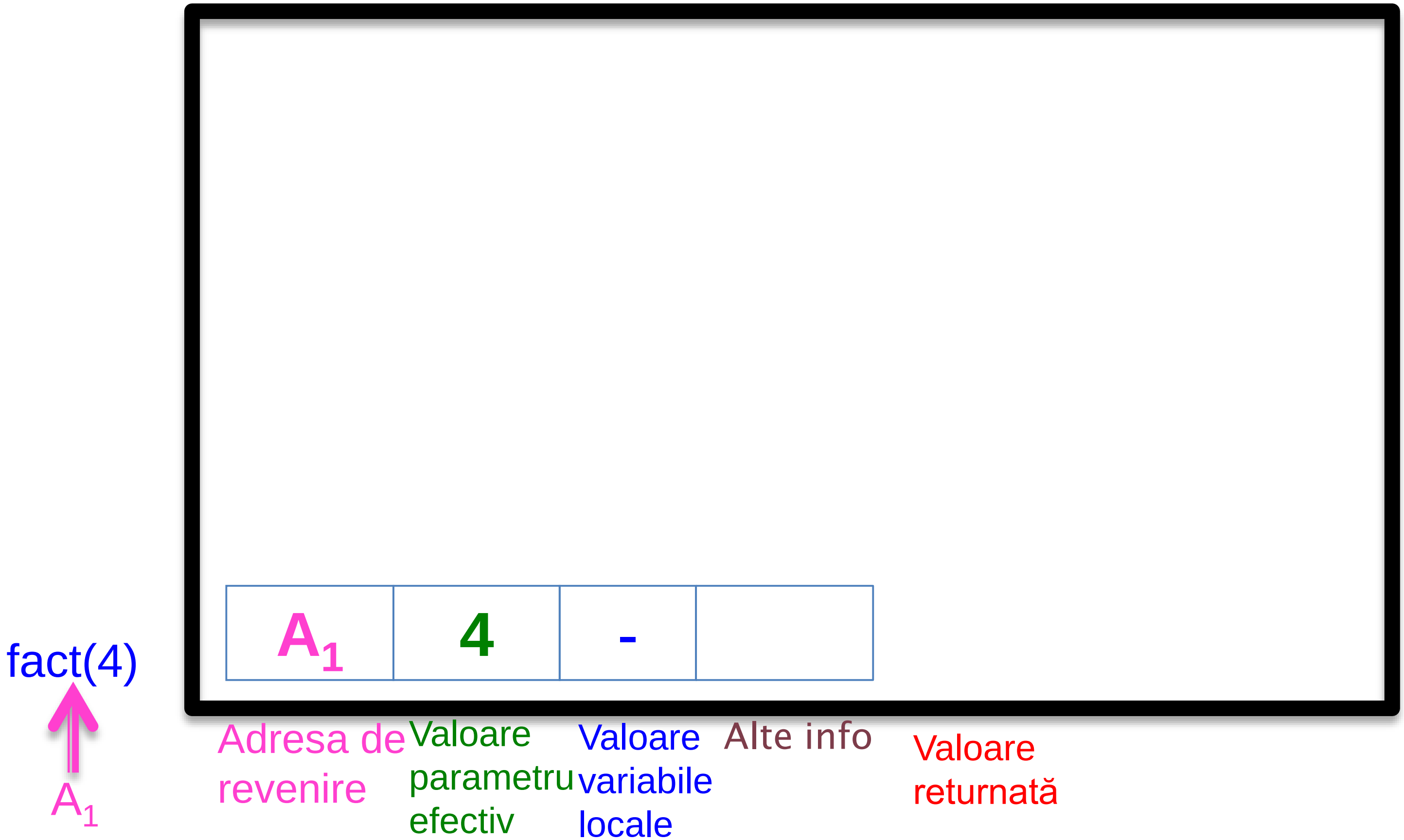
Exemplul 1 – factorial

```
def fact(n):
    if n==0: #conditia de oprire
        return 1
    else:
        return n * fact(n-1)
```

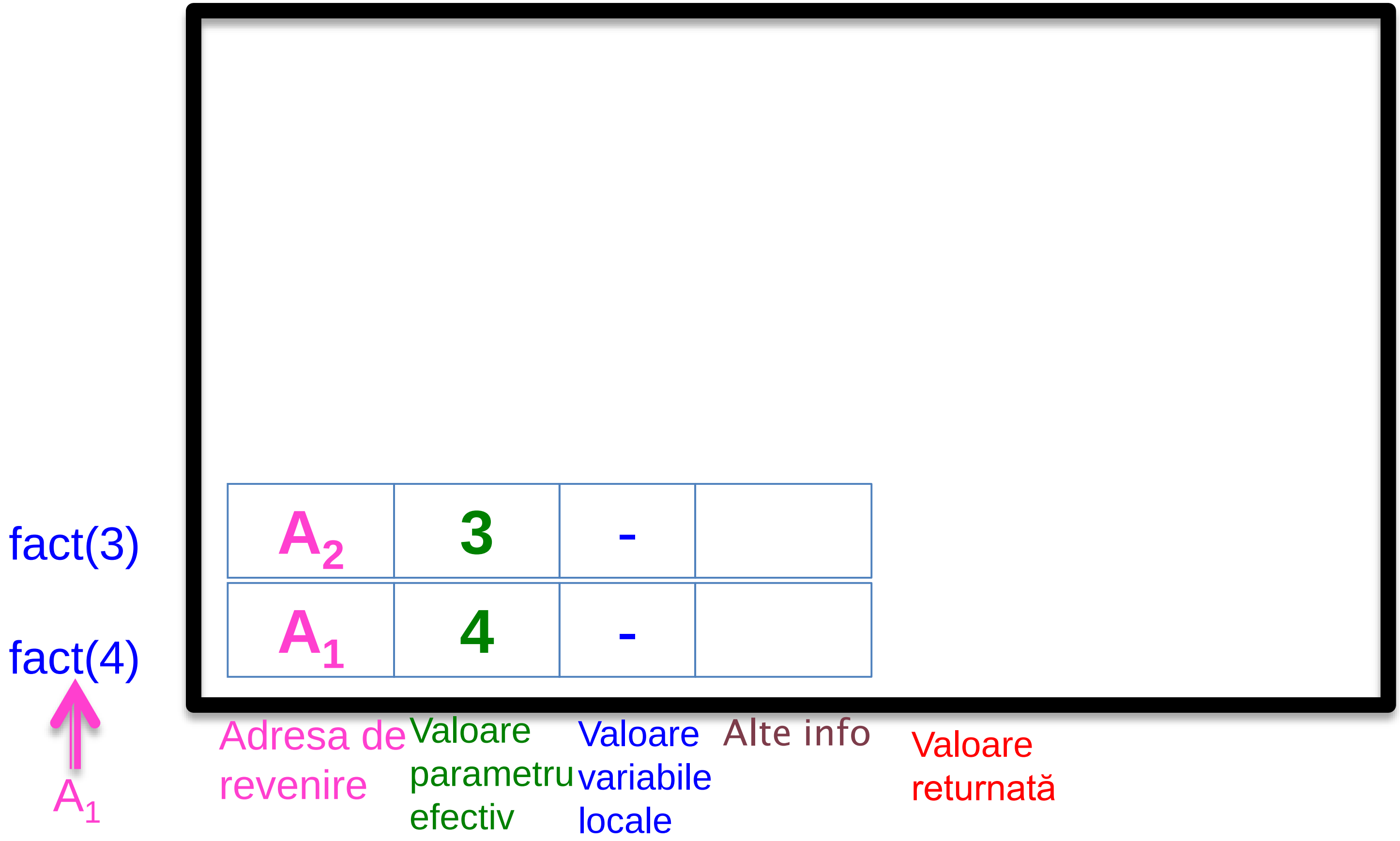
Recursivitate – stiva

- La un apel de funcție => **context de apel** asociat
  - Conține **numele, variabilele locale, parametrii**, adresa de revenire etc.

Recursivitate – stiva



Recursivitate - stiva



Recursivitate – stiva

	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info	Valoare returnată
fact(2)	A_3	2	-		
fact(3)	A_2	3	-		
fact(4)	A_1	4	-		

A_1

Recursivitate – stiva

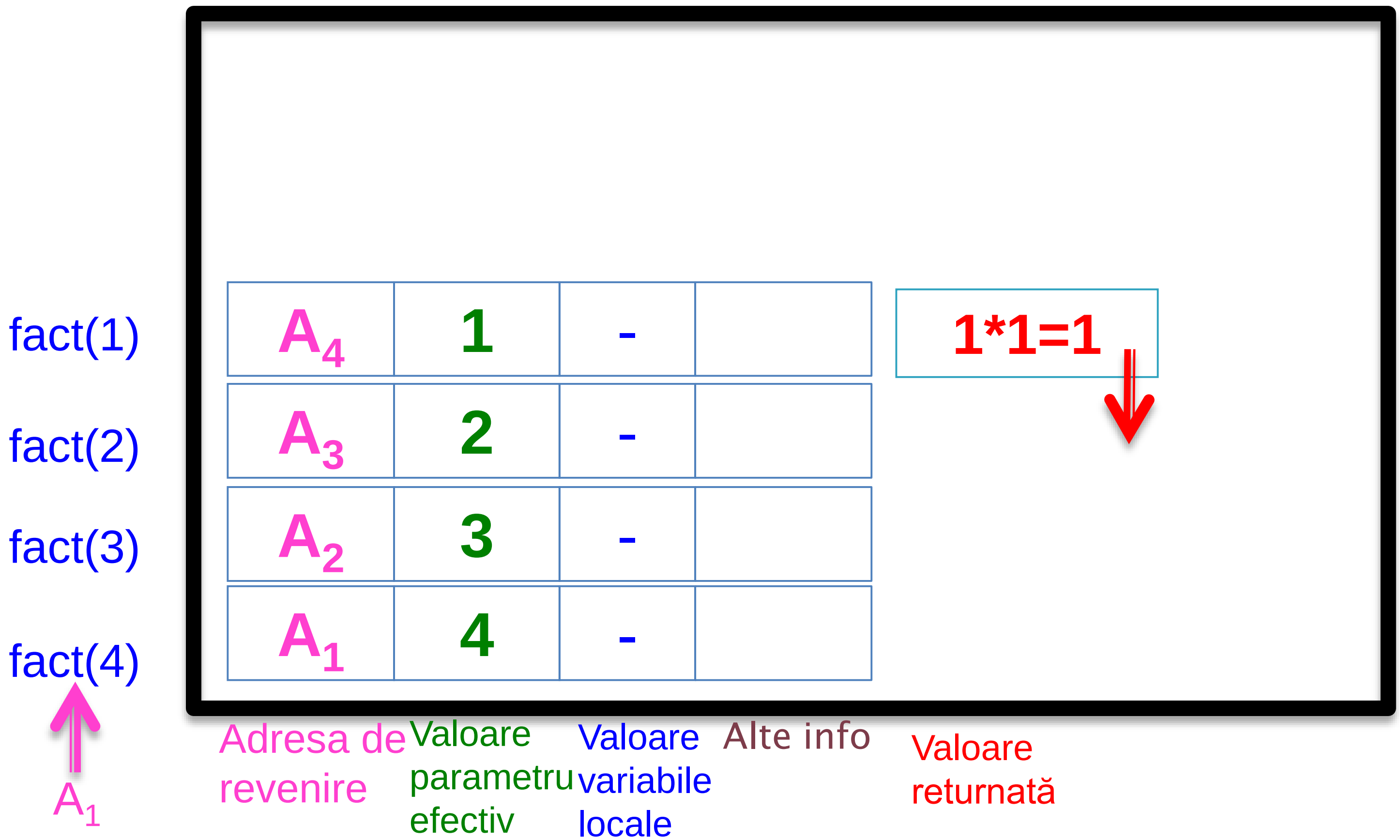
fact(1)	A_4	1	-		
fact(2)	A_3	2	-		
fact(3)	A_2	3	-		
fact(4)	A_1	4	-		
A_1	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info	Valoare returnată

Recursivitate – stiva

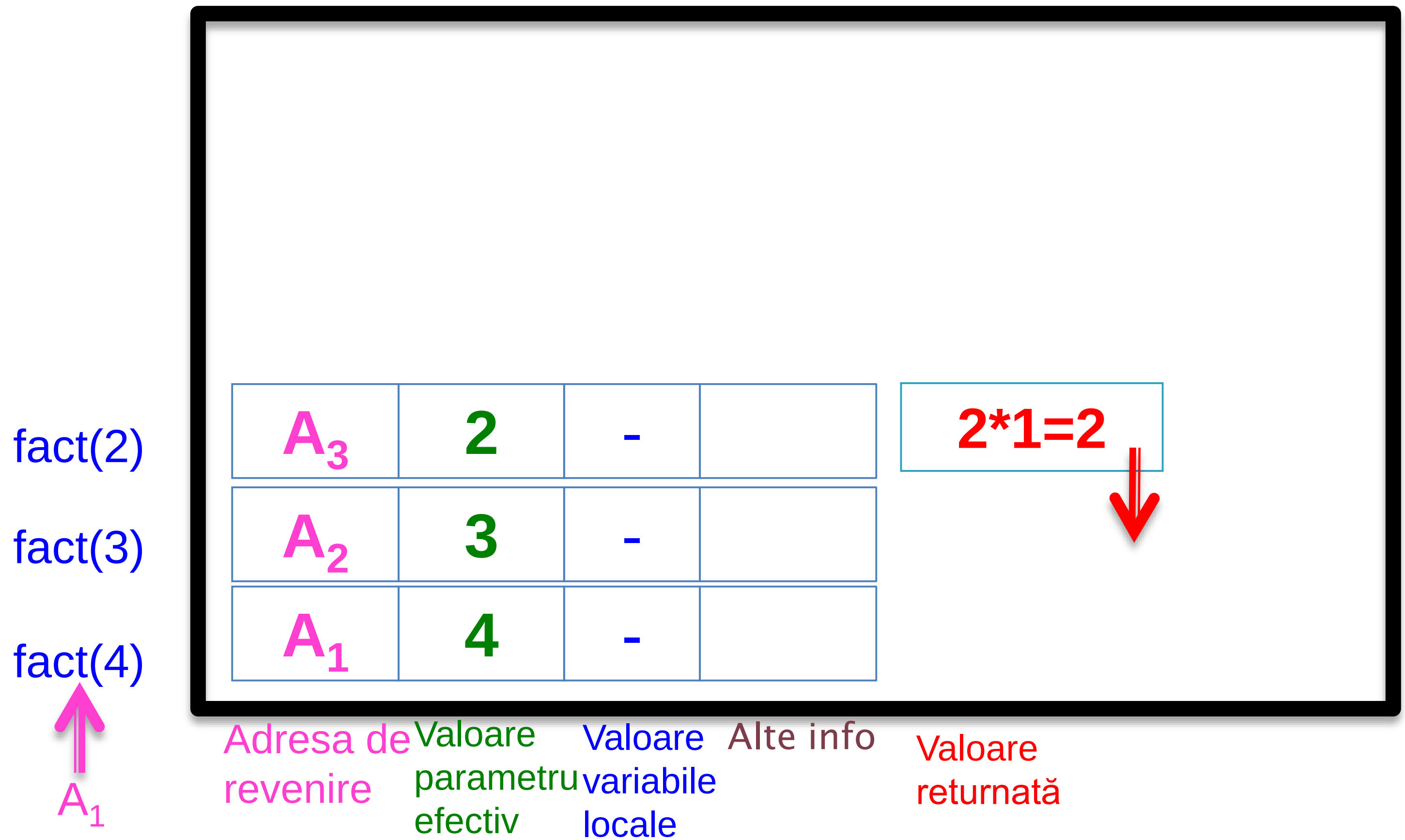
	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info	Valoare returnată
fact(0)	A_5	0	-		1
fact(1)	A_4	1	-		
fact(2)	A_3	2	-		
fact(3)	A_2	3	-		
fact(4)	A_1	4	-		

A_1

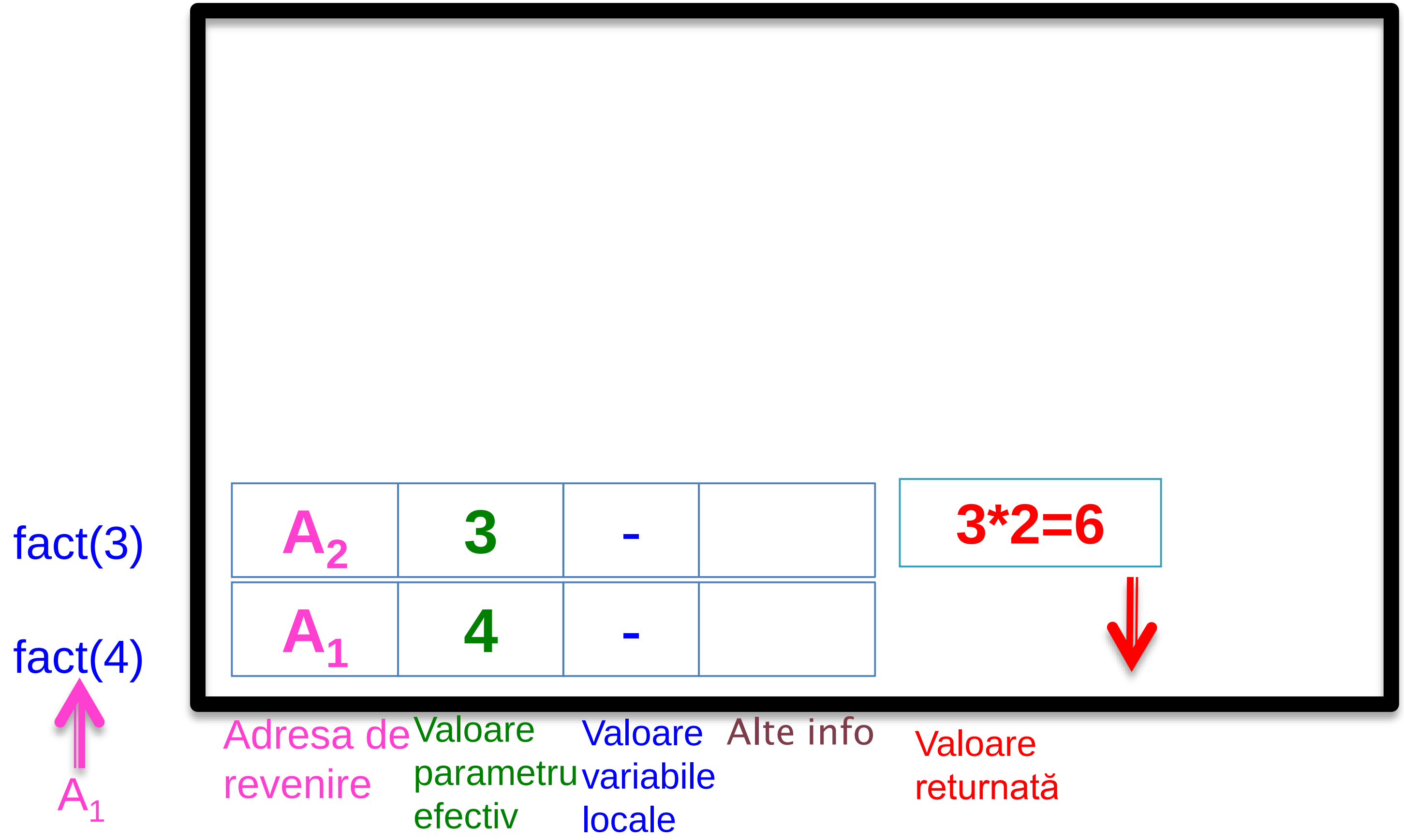
Recursivitate – stiva



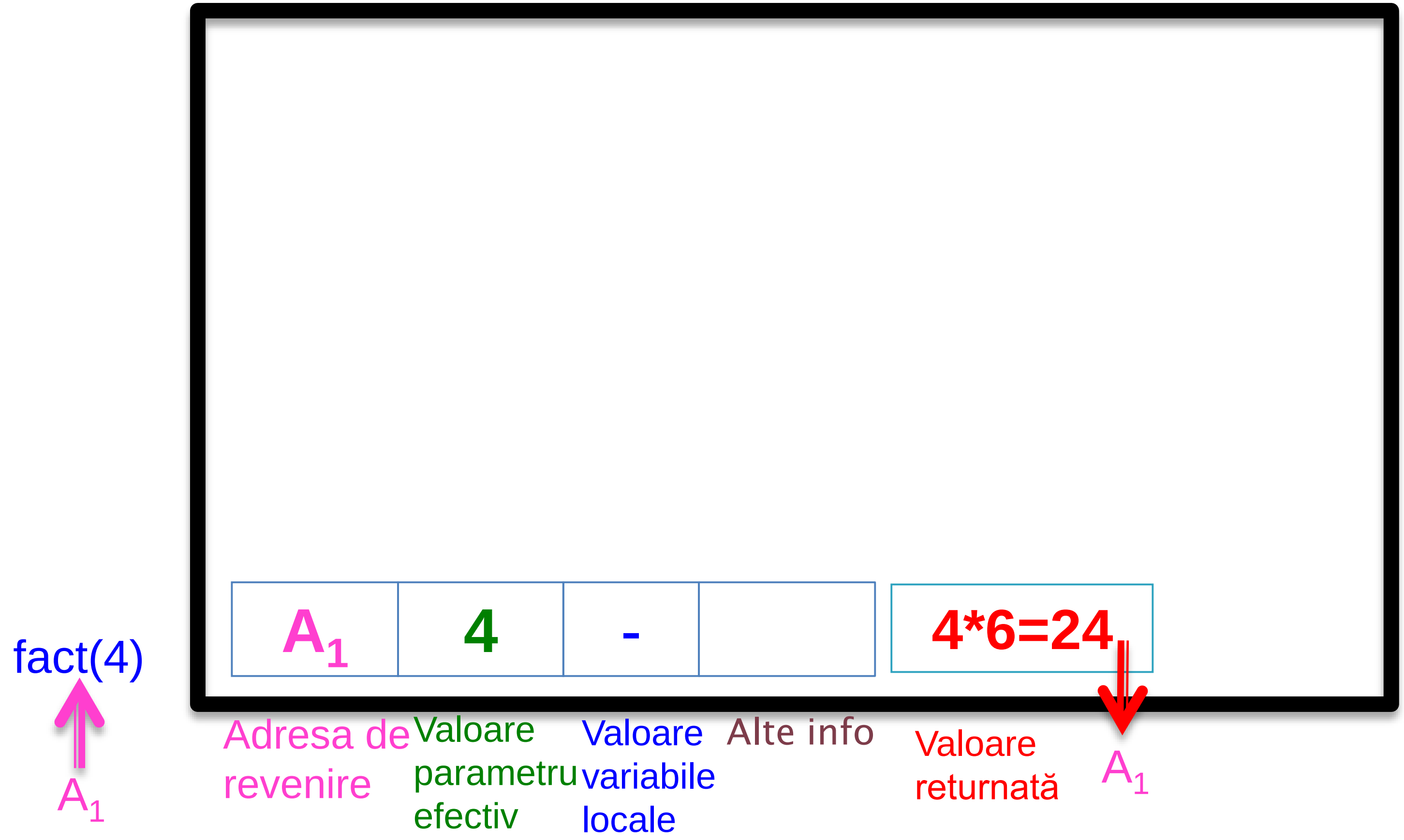
Recursivitate – stiva



Recursivitate - stiva



Recursivitate - stiva



Exemplul 2
Suma cifrelor lui n

Exemplul 2 – suma cifrelor

- suma cifrelor(n) = suma cifrelor(n//10) +ultima cifra(n)
- suma cifrelor(0) = 0

Exemplul 2 – suma cifrelor

```
def suma_cifre(n):
    if n==0:
        return n
    return n%10 + suma_cifre(n//10)
```

Exemplul 3
Afisare numere 1..n crescator/descrescător

Exemplul 3 – afișare 1..n

Reducem problema la o problemă mai mică.

Varianta 1:

- afișăm **1** ... **n-1** => problemă mai mică de același fel
- scriem **n**

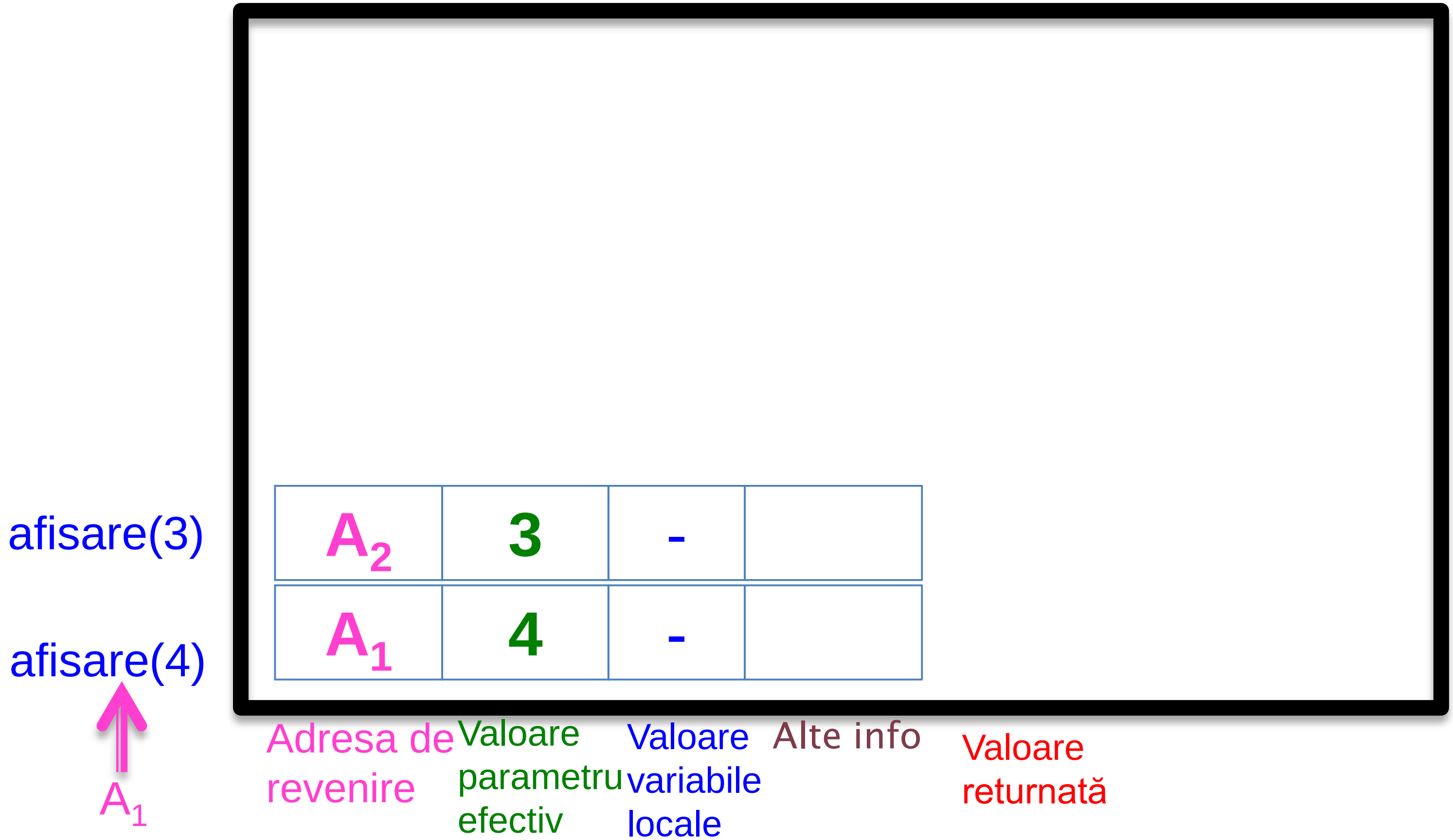
Exemplul 3 – afișare 1..n

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")


afisare(4)
```

Recursivitate – stiva


```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Recursivitate – stiva




```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Recursivitate – stiva

	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info	Valoare returnată
afisare(2)	A_3	2	-		
afisare(3)	A_2	3	-		
afisare(4)	A_1	4	-		

A_1

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Recursivitate – stiva

	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info
afisare(1)	A_4	1	-	
afisare(2)	A_3	2	-	
afisare(3)	A_2	3	-	
afisare(4)	A_1	4	-	

A_1

Valoare returnată

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Recursivitate – stiva

afisare(1)	A_4	1	-	
afisare(2)	A_3	2	-	
afisare(3)	A_2	3	-	
afisare(4)	A_1	4	-	
A_1	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info

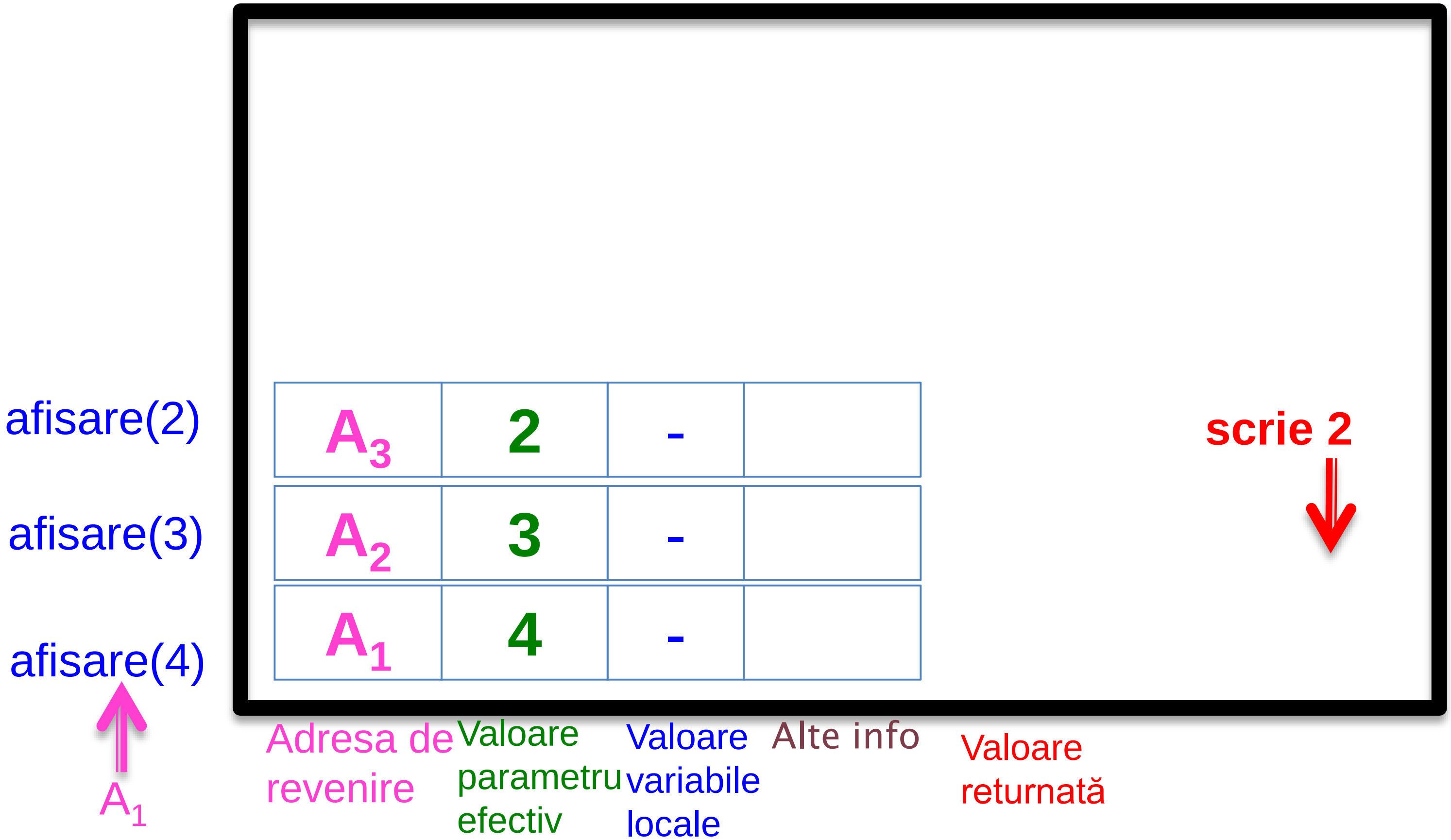
scrie 1

Valoare returnată

1

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

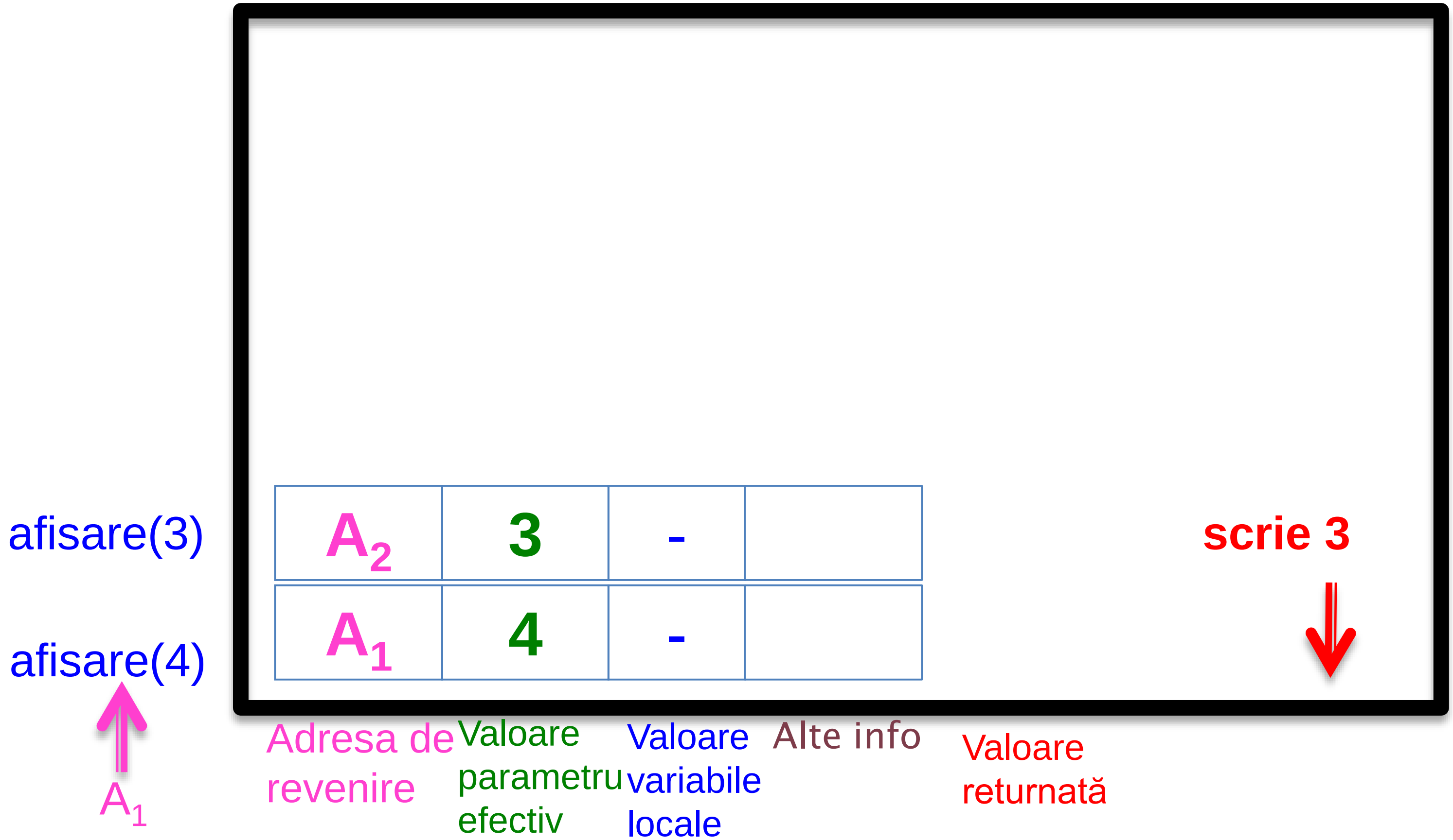
Recursivitate – stiva



1 2

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

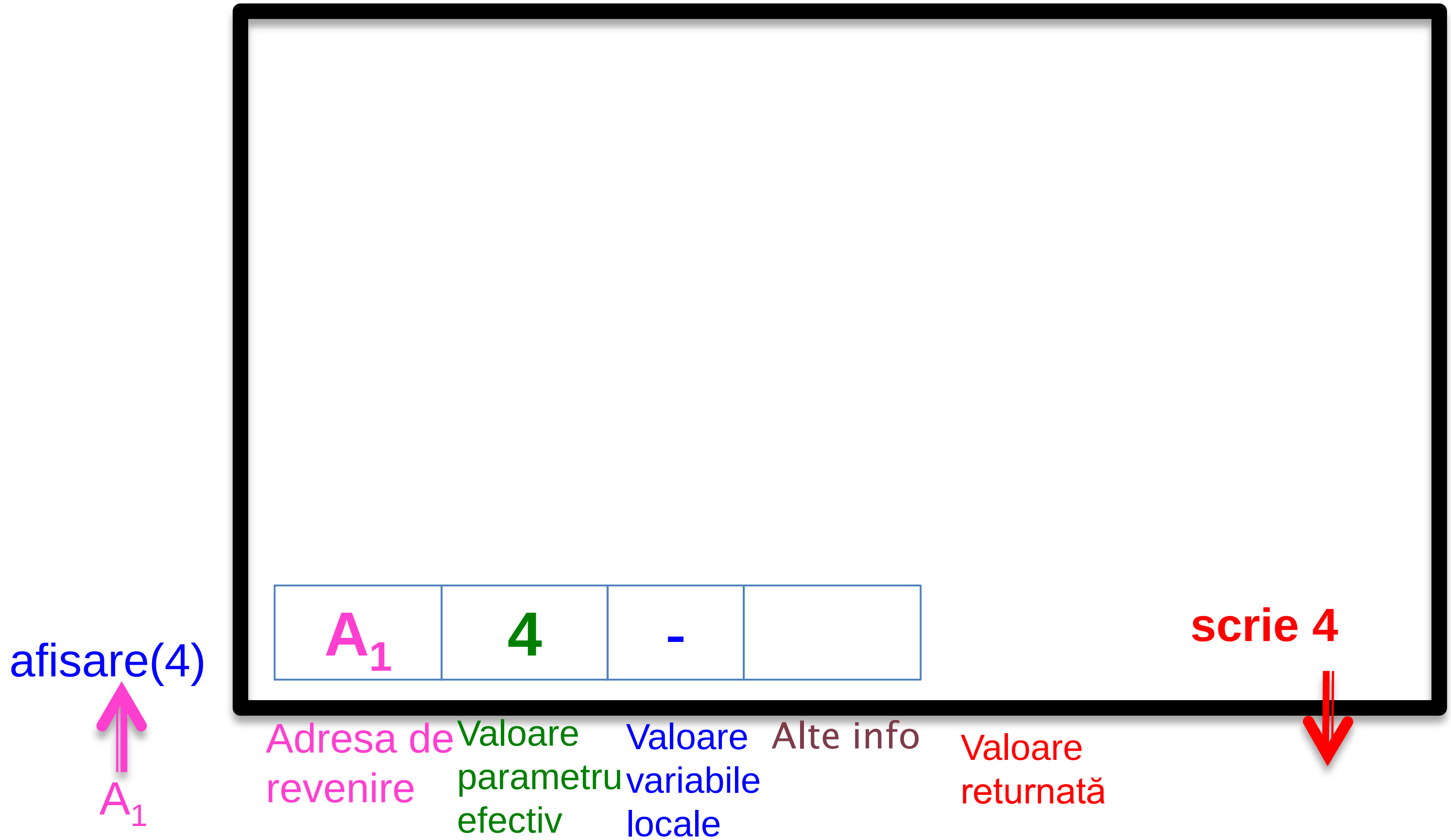
Recursivitate – stiva



1 2 3

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Recursivitate – stiva



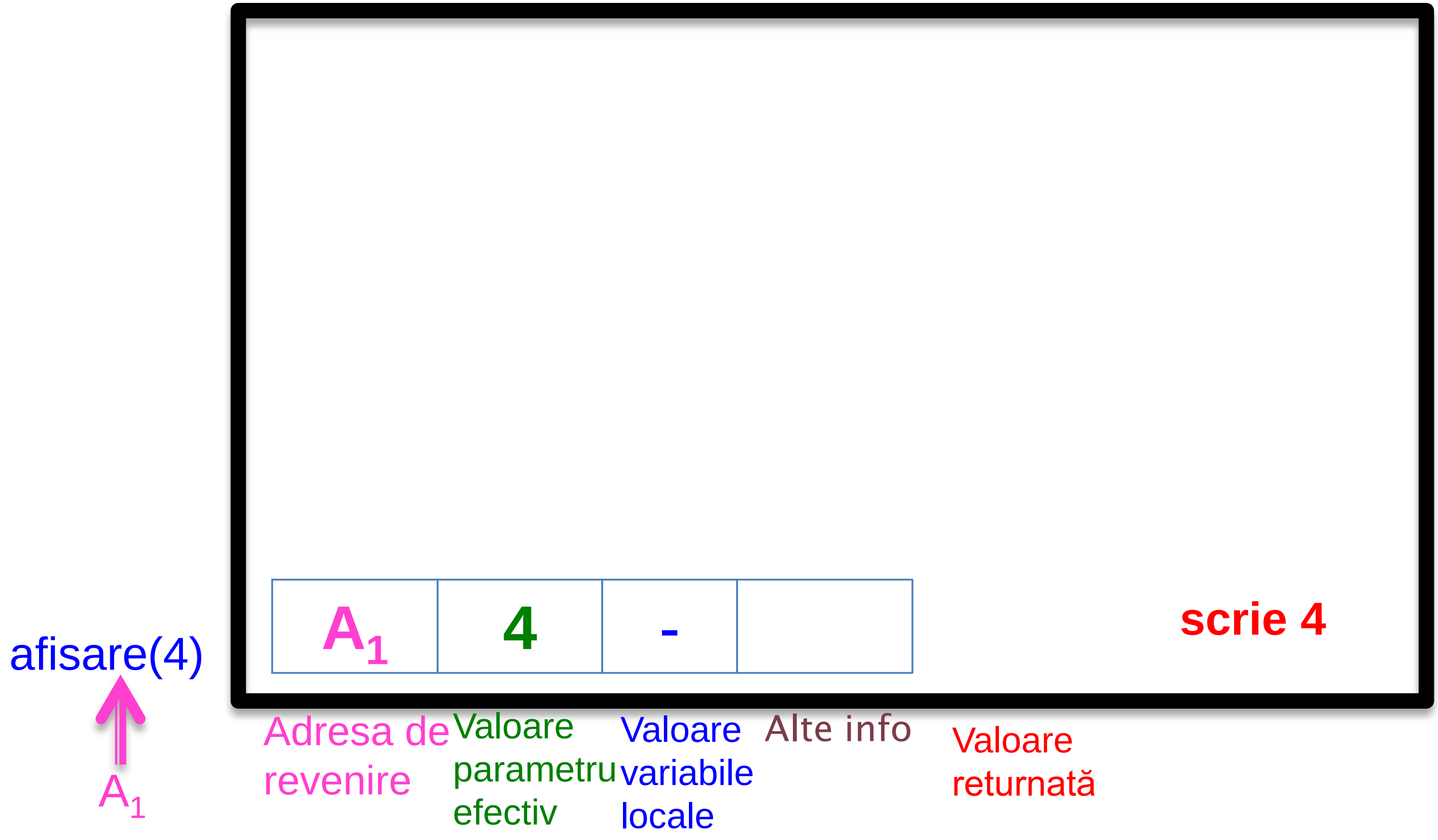
1 2 3 4

```
def afisare(n):
    if n>1:
        afisare(n-1)
    print(n, end = " ")
```

Exemplul 3 – afișare 1..n

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)

afisare(4)
```

Recursivitate – stiva



4

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate - stiva

afisare(3)

afisare(4)

A_1

A_2	3	-	
A_1	4	-	

scrie 3

Adresa de revenire | Valoare parametru efectiv | Valoare variabile locale | Alte info | Valoare returnată

4 3

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate – stiva

afisare(2)	A_3	2	-	
afisare(3)	A_2	3	-	
afisare(4)	A_1	4	-	

scrie 2

A_1

Adresa de revenire | Valoare parametru efectiv | Valoare variabile locale | Alte info | Valoare returnată

4 3 2

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate – stiva

	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info
afisare(1)	A_4	1	-	
afisare(2)	A_3	2	-	
afisare(3)	A_2	3	-	
afisare(4)	A_1	4	-	

scrie 1

Valoare returnată

A_1

4 3 2 1

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate – stiva

	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info
afisare(1)	A_4	1	-	
afisare(2)	A_3	2	-	
afisare(3)	A_2	3	-	
afisare(4)	A_1	4	-	

A_1

Valoare returnată

4 3 2 1

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate - stiva

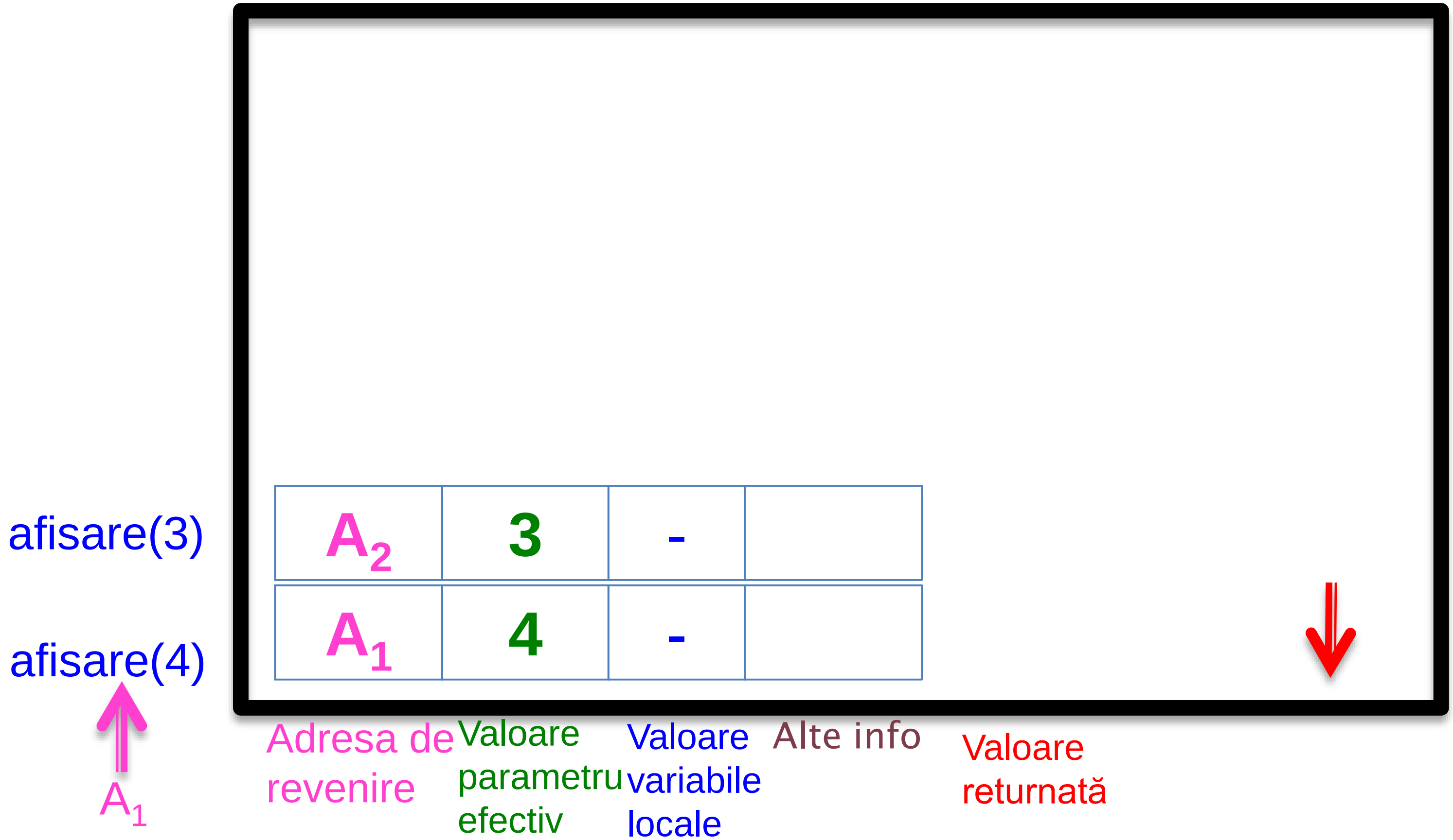
	Adresa de revenire	Valoare parametru efectiv	Valoare variabile locale	Alte info	Valoare returnată
afisare(2)	A_3	2	-		
afisare(3)	A_2	3	-		
afisare(4)	A_1	4	-		

A_1

1 2

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate – stiva



1 2 3

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Recursivitate – stiva

1 2 3 4

```
def afisare(n):
    print(n, end=" ")
    if n>1:
        afisare(n-1)
```

Exemplul 3 – afișare 1..n

Reducem problema la o problemă mai mică.

Varianta 2:

– scriem 1

– Mai trebuie să afișăm 2...n => problemă mai mică de același fel

Exemplul 3 – afișare 1..n

```
def afisare_numere(i,n):
    if i<=n:
        print(i, end=" ")
        afisare_numere(i+1,n)



afisare_numere(1,10)
```

Exemplul 3 – afișare 1..n

```
def afisare_numere(i,n):
    if i<=n:


        afisare_numere(i+1,n)
        print(i, end=" ")


afisare_numere(1,10)
```

Exemplul 3 – afișare 1..n

```
def afisare(n):
    if n<=0:
        return
    if n>1:
        afisare(n-1)
    print(n, end = " ")

afisare(0)
```

completare

Recursivitate

- Limită a numărului maxim de apeluri incuibate:

  - **`sys.getrecursionlimit()`**
  - **`sys.setrecursionlimit(noua_limita)`**

Recursivitate

- **Avantaje:**

  pentru anumite probleme, variata iterativă este foarte laborioasă => implementarea recursivă: elegantă, ușor de urmărit

Recursivitate

- **Dezavantaje:**
  - necesita spațiu suplimentar pe stiva pentru fiecare apel
  - pentru anumite tipuri de recurențe (de exemplu de grad 2), implementarea recursivă poate duce la algoritmi cu complexitate mare – exemplu: calculul termenului n din șirul lui Fibonacci 1,1,2,3,5,....

$$F_n = F_{n-1} + F_{n-2}$$

$$F_1 = F_0 = 1$$

Recursivitate – Fibonacci

```
def fibonacci(n):
    if n<=1:
        return 1
    return fibonacci(n-1)+fibonacci(n-2)
```

Exemplu

fibonacci(8)

fibonacci(7) fibonacci(6)

fibonacci(6) **fibonacci(5)** **fibonacci(5)** fibonacci(4)

fibonacci(5) fibonacci(4)

....

(valori care se recalculeaza, in mod redundant)

Fibonacci nerecursiv

```
def fibonacci_nerecursiv(n):
    f0, f1 = 1, 1
    for i in range(2,n+1):
        f0, f1 = f1, f0+f1
    return f1
```

Complexitate – ordin n – vom defini

Recursivitate

- **Greşeli frecvente:**
  - absența unei condiții de oprire
  - condiții de oprire în care nu apar variabilele locale sau parametrii
  - declararea parametrilor inutili, care încarcă stiva

Timpul de executare

Timpul de executare a algoritmilor

- se măsoară în **funcţie de lungimea n a datelor de intrare**

- **`T(n)`** = timpul de executare pentru orice set de date de intrare de lungime **`n`**

  – dat de numărul de operații elementare în funcție de n

Timpul de executare a algoritmilor

- Se numără operații elementare (de atribuire, aritmetice, de decizie, de citire/scriere)

- **Numărare aproximativă** => **ordinul de mărime** al numărului de **operații elementare**

- Pentru simplitate – se **fixează operație de bază**

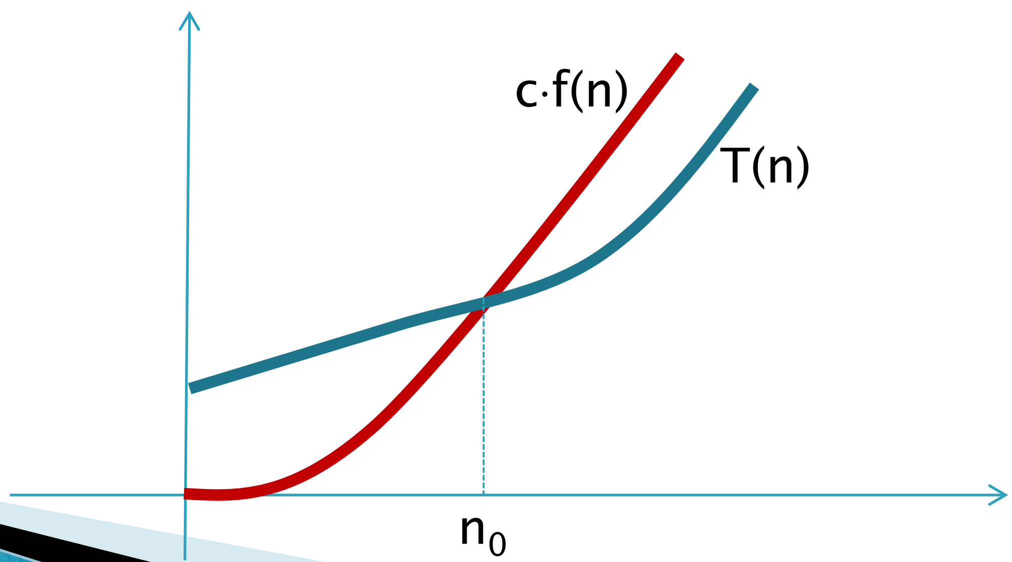
Timpul de executare a algoritmilor

- În majoritatea cazurilor ne mărginim la a evalua **ordinul de mărime** al timpului de executare = **ordin de complexitate al algoritmului**

$$T(n) = O(f(n))$$

$$\exists c,\ n_0 \text{ - constante a.î } \forall n \geq n_0$$

$$T(n) \leq c \cdot f(n)$$



Timpul de executare a algoritmilor

- Notație: `T(n) = O(f(n))`
- comportare **asimptotică**
- caz defavorabil
- O(expresie) = O(termen dominant)

$O(2n) \Rightarrow O(n)$

$O(2n^2+4n+1) \Rightarrow O(n^2)$

$O(n^2 - n) \Rightarrow O(n^2)$

Timpul de executare a algoritmilor

- Exemplul 1

```
citeste n
pentru i = 1, n executa
     pentru j = 1,n executa
           scrie j
     scrie linie noua
```

Timpul de executare a algoritmilor

- **Exemplul 1**

```
citeste n
pentru i = 1, n executa
     pentru j = 1,n executa
           scrie j
     scrie linie noua
```

Afișare

1 2 ... n
1 2 ... n
...
1 2 ... n

de n ori

Complexitate $O(n^2)$

Timpul de executare a algoritmilor

- **Exemplul 2**

```
citeste n
pentru i = 1, n executa
     pentru j = 1,i executa
           scrie j
     scrie linie noua
```

Timpul de executare a algoritmilor

- **Exemplul 2**

```
citeste n
pentru i = 1, n executa
     pentru j = 1,i executa
           scrie j
     scrie linie noua
```

Afișare

1

1 2

1 2 3

1 2 3 4

1 2 ....n

Timpul de executare a algoritmilor

- **Exemplul 2**

```
citeste n
pentru i = 1, n executa
      pentru j = 1,i executa
             scrie j
      scrie linie noua
```

Afișare
1
1 2
1 2 3
1 2 3 4
1 2 ....n

$1 + 2 + 3 + \ldots + n = n(n+1)/2$ afișări ale lui j

Complexitate $O(n^2)$

- **Exemplul 3**

```
citeste n
p = 1
pentru i = 1, n+1 executa
      pentru j = 1,p executa
            scrie j
      scrie linie noua
      p = p * 2
```

Timpul de executare a algoritmilor

- **Exemplul 3**

```
citeste n
p = 1
pentru i = 1, n+1 executa
      pentru j = 1,p executa
            scrie j
      scrie linie noua
     p = p * 2
```

Afișare

1

1 2

1 2 3 4

1 2 3 4 5 6 7 8

1 2 .... 2^n

Timpul de executare a algoritmilor

- **Exemplul 3**

```
citeste n
p = 1
pentru i = 1, n+1 executa
      pentru j = 1,p executa
            scrie j
      scrie linie noua
     p = p * 2
```

Afișare
1
1 2
1 2 3 4
1 2 3 4 5 6 7 8
1 2 .... 2^n

$1 + 2 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$ afișări ale lui j

$O(2^n)$

Timpul de executare a algoritmilor

- **Exemplul 4** – Cea mai mică putere a lui 2 mai mare ca n

```
citeste n
p = 1
cat timp p<=n executa:
  p = p * 2
scrie p
```

Timpul de executare a algoritmilor

- **Exemplul 4** – Cea mai mică putere a lui 2 mai mare ca n

```
citeste n
p = 1
cat timp p<=n executa:
  p = p * 2
scrie p
```

$O(\log_2(n))$

Timpul de executare a algoritmilor

Alte exemple

```
citeste n, m, p

pentru i = 1, n executa
    pentru j = 1, m executa
            operatii O(1)
    pentru k = 1, p executa
            operatii O(1)
```

$O(n(m+p))$

Timpul de executare a algoritmilor

Alte exemple

- Suma elementelor unui vector de lungime n **O(n)**
- Calculul transpusei unei matrice nxn **O(n^2)**
- Înmulțirea a două matrice A(n,m) B(m,p) **O(nmp)**

Timpul de executare a algoritmilor

Notație: `T(n) = O(f(n))`

Clase de complexitate uzuale:

- **Complexitate logaritmică** $O(\log_2(n))$, $O(\log(n))$

  Exemplu: căutarea binară

- **Complexitate liniară** $O(n)$

  Exemplu: minimul dintr-un vector

Timpul de executare a algoritmilor

- **Complexitate pătratică** $O(n^2)$

  Exemplu: suma elementelor unei matrice nxn,

  sortarea prin metoda bulelor, prin selecție

- **Complexitate polinomială** $O(n^k)$, $k \geq 3$

  Exemplu: înmulțirea a două matrice pătratice de dimensiune n

Timpul de executare a algoritmilor

- **Complexitate pătratică** $O(n^2)$

  Exemplu: suma elementelor unei matrice nxn,

  sortarea prin metoda bulelor, prin selecție

- **Complexitate polinomială** $O(n^k)$, $k \geq 3$

  Exemplu: înmulțirea a două matrice pătratice de dimensiune n

- **Complexitate exponențială** $O(k^n)$, $k \geq 2$

  Exemplu: generarea submulțimilor unei mulțimi cu n elemente

- **Complexitate factorială** $O(n!)$

  Exemplu: generarea permutărilor unui vector cu n elemente

Metoda Divide et Impera

desparte şi stăpâneşte

Metoda Divide et Impera

- Constă în
  - **împărţirea** **repetată** a unei probleme de dimensiuni mari în mai multe subprobleme **<u>de acelaşi tip</u>**

Metoda Divide et Impera

- Constă în
  - **împărţirea repetată** a unei probleme de dimensiuni mari în mai multe subprobleme **de acelaşi tip**
  - **rezolvarea** acestor subprobleme (în acelaşi mod sau prin rezolvare directă, dacă au dimensiune mică)

Metoda Divide et Impera

- Constă în
  - **împărţirea repetată** a unei probleme de dimensiuni mari în mai multe subprobleme **de acelaşi tip**
  - **rezolvarea** acestor subprobleme (în acelaşi mod sau prin rezolvare directă, dacă au dimensiune mică)
  - **combinarea rezultatelor** obţinute pentru a determina rezultatul corespunzător problemei iniţiale.

Metoda Divide et Impera

- caracter recursiv

- există și probleme cu implementare iterativă

Exemple clasice

- **Calculul x^n** – exemplu de problemă DI fără vector
- **Căutare binară**
- **Sortarea prin interclasare (Merge Sort)**
- **Sortarea rapidă (Quick Sort)**
- **Problema turnurilor din Hanoi**

Ridicarea la putere x^n

Calculul x^n

- **Relație de recurență 1:**

$$x^n = \begin{cases} x^{n-1} \cdot x, & \text{daca } n > 0 \\ 1, & \text{altfel} \end{cases}$$

Calcul x^n

```
def power(x, n):
    if n == 0:
        return 1
    return x * power(x, n - 1)


import sys
sys.setrecursionlimit(3000)


x=3
n=2000

print(power(x,n))
```

Calculul x^n

```
def power(x, n):
    if n == 0:
        return 1
    return x * power(x, n - 1)
```

$\text{power}(x, n)$ *o înmulțire*

$\text{power}(x, n-1)$ *o înmulțire*

$\text{power}(x, n-2)$ *o înmulțire*

...

$\text{power}(x, 1)$ *o înmulțire*

$\text{power}(x, 0)$

Calculul x^n

```
def power(x, n):
    if n == 0:
        return 1
    return x * power(x, n - 1)
```

Ordin complexitate n – notație O(n)

- aproximativ n înmulțiri /operații elementare, modulo o constantă

$\mathrm{power}(x, n)$ *o înmulțire*

$\mathrm{power}(x, n-1)$ *o înmulțire*

$\mathrm{power}(x, n-2)$ *o înmulțire*

...

$\mathrm{power}(x, 1)$ *o înmulțire*

$\mathrm{power}(x, 0)$

Calculul x^n

- **Relație de recurență 2:**

$$x^n = \begin{cases} (x^{[n/2]})^2, & \text{daca } n > 0 \text{ si este par} \\ (x^{[n/2]})^2 \cdot x, & \text{daca } n > 0 \text{ si este impar} \\ 1, & \text{daca n = 0} \end{cases}$$

se reduce la o subproblemă de dimensiune n/2

Calculul x^n

```
def power_DI(x, n):
    if n == 0:
        return 1
    y = power_DI(x, n//2)
    if n%2 == 1:
        return x*y*y
    else:
        return y*y


print(power_DI(3,20))
```

Calculul x^n

Pentru $n = 2^k$

$$\text{power\_DI}(x, n)$$ $\leq c$ *înmulțiri*

$$\downarrow$$

$$\text{power\_DI}\left(x, \frac{n}{2}\right)$$

$$\downarrow$$

$$\text{power\_DI}\left(x, \frac{n}{2^2}\right)$$

$$\downarrow$$

...

$$\text{power\_DI}\left(x, \frac{n}{2^k}\right) = \text{power\_DI}(x, 1)$$

$$\downarrow$$

$$\text{power\_DI}(x, 0)$$

$k+2 = \log_2(n) + 2$

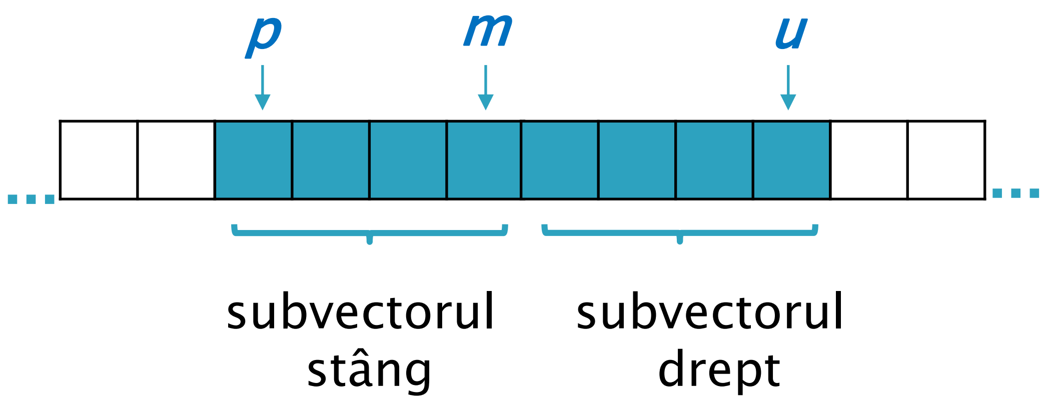
Ordin complexitate $\log_2 n$, notație $O(\log_2 n)$

- aproximativ $\log_2 n$ înmulțiri /operații elementare, modulo o constantă

Metoda Divide et Impera pentru vectori

- Vectorul se împarte în subvectori mai mici (corespunzători subproblemelor)

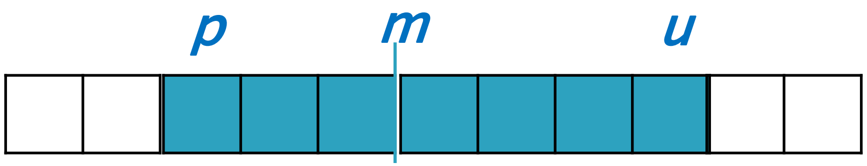
- Un subvector este **identificat de indicele primului și indicele ultimului element, notate p și u**
  - !!! nu se copiază subvectorul în alt vector



- Cel mai frecvent se împarte în 2 subvectori aproape egali, deci

$$m = (p+u)/2$$

Schema posibilă



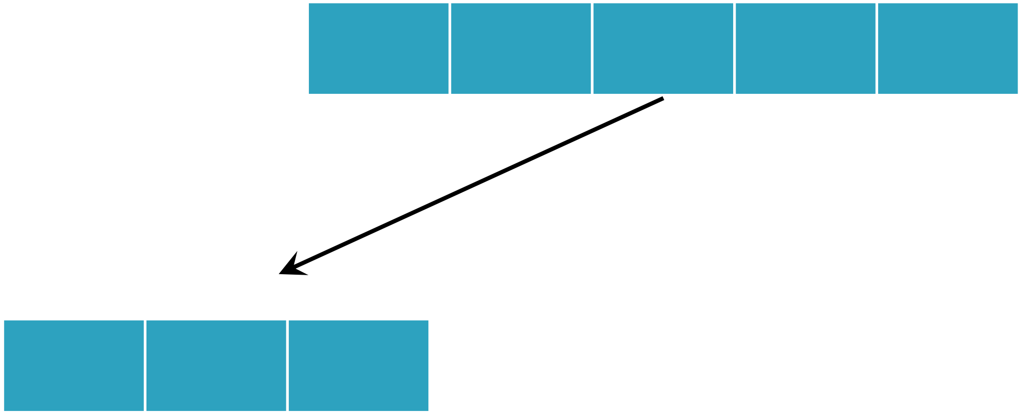

```
function DivImp(p,u)          – pentru a[p..u]
     if u – p < ε – subproblema mica
          r ← RezolvaDirect(p,u)
    else
          m ← Pozitie(p,u) – de obicei mijlocul
          r1 ← DivImp(p,m)
          r2 ← DivImp(m+1,u)
          r ← Combina(r1,r2)
    return r
end
```

Apel:

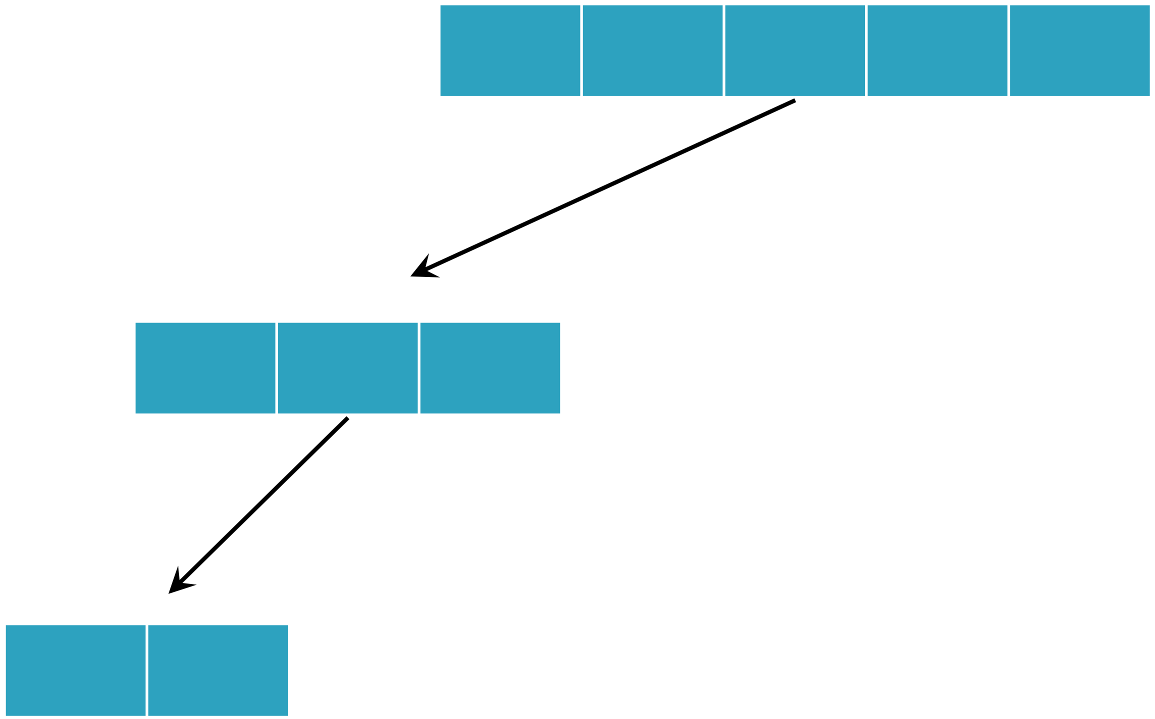
```
DivImp(1, n)
DivImp(0, n-1)
```

Metoda Divide et Impera

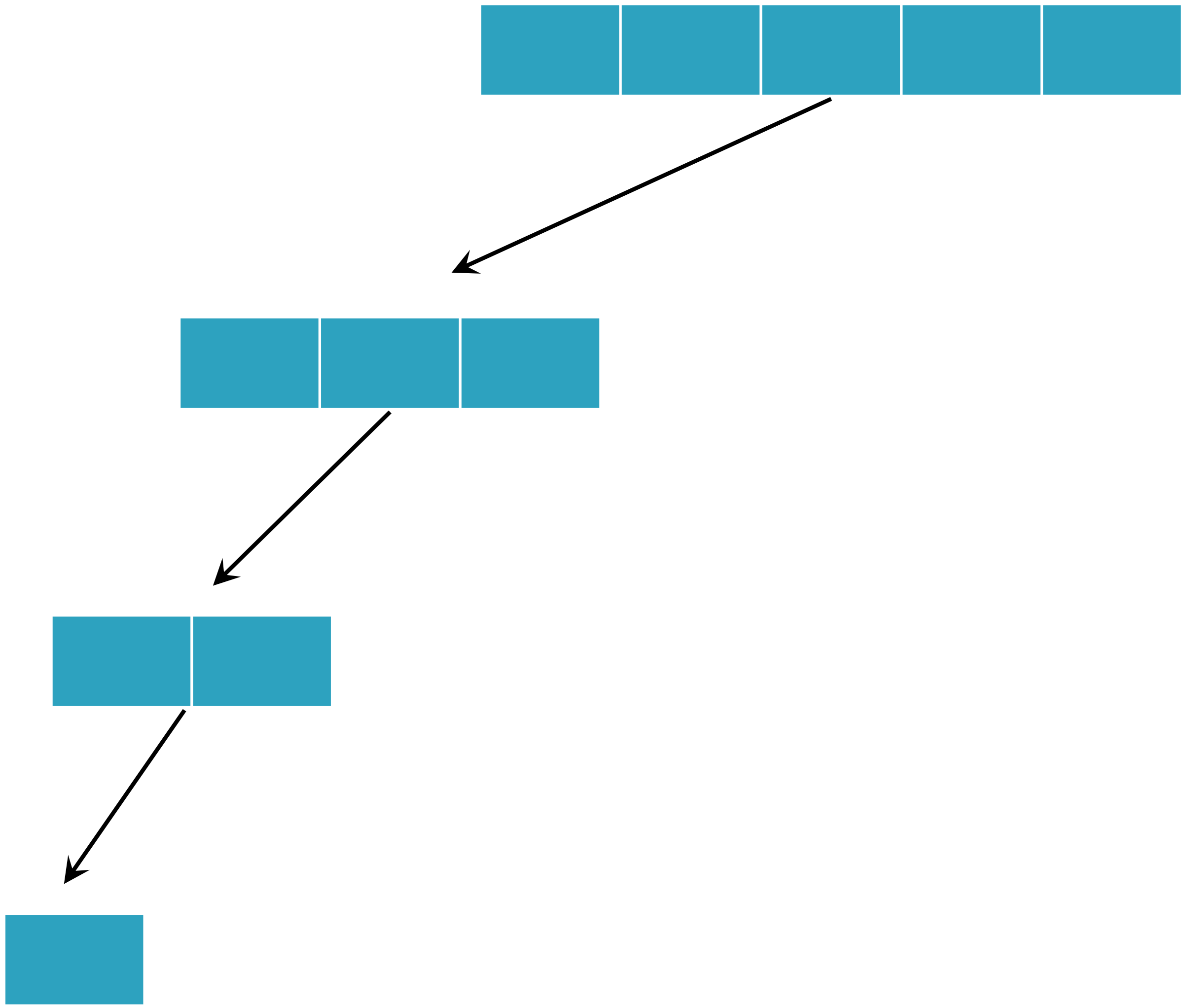
Metoda Divide et Impera



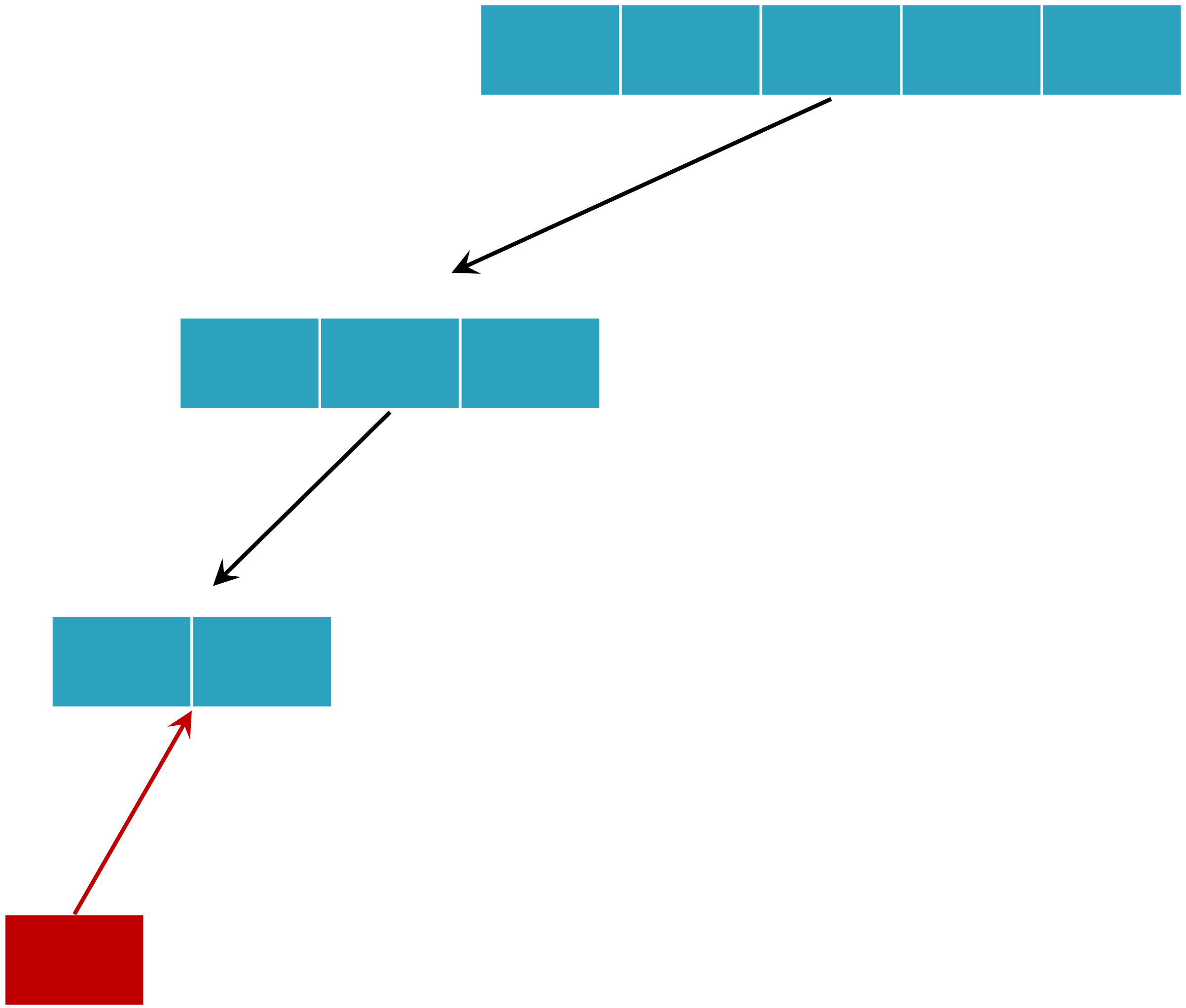
Metoda Divide et Impera



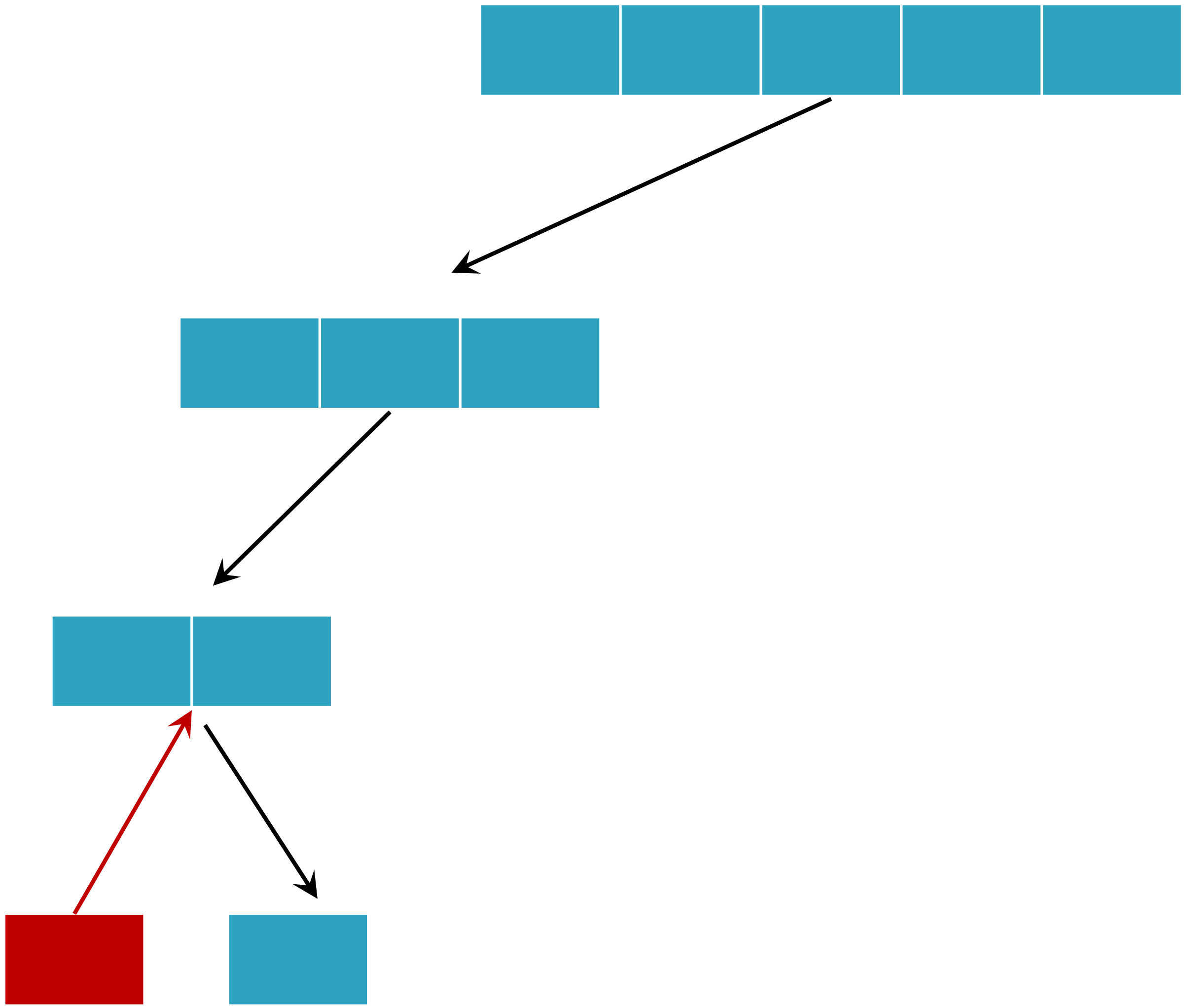
Metoda Divide et Impera



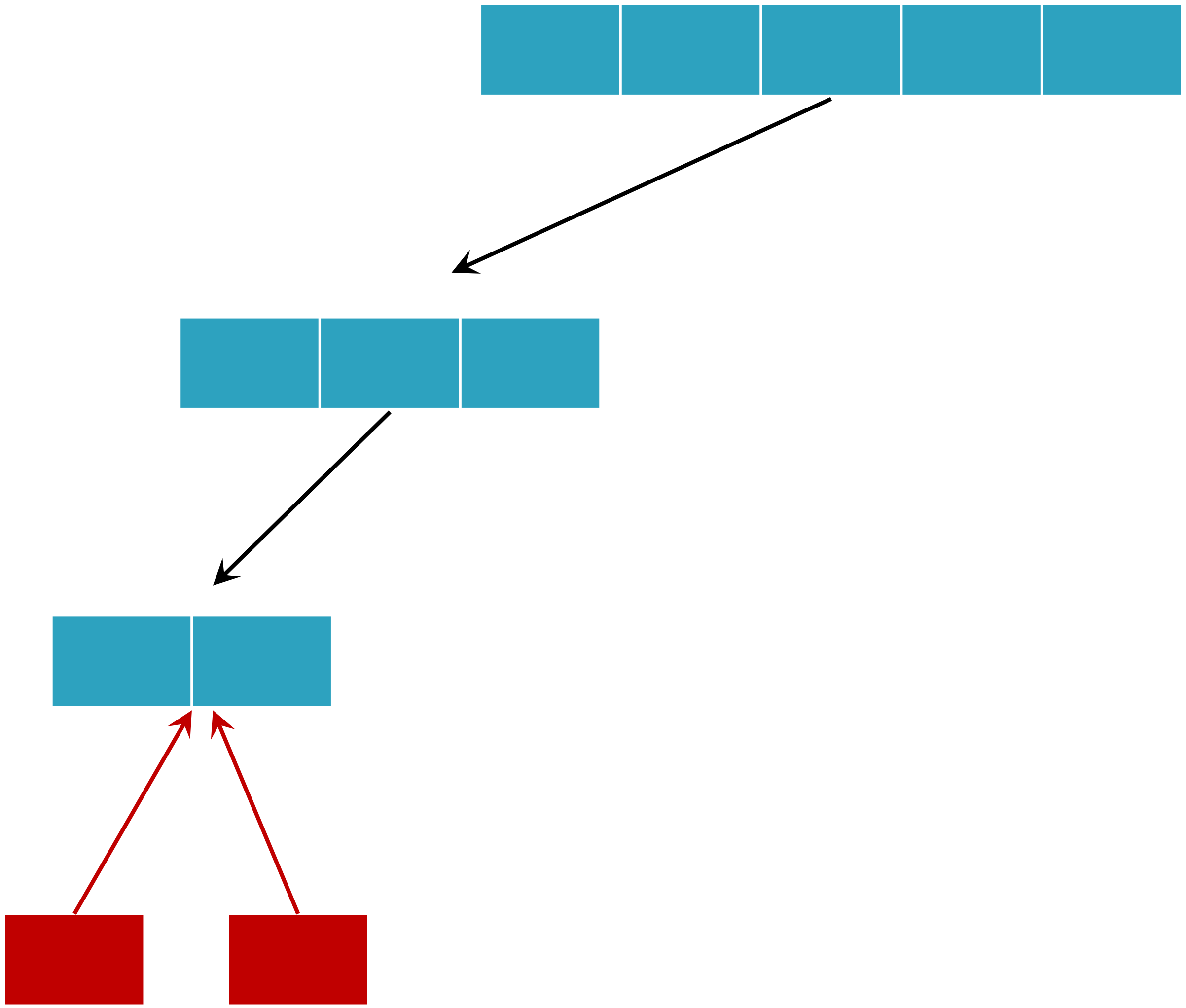
Metoda Divide et Impera



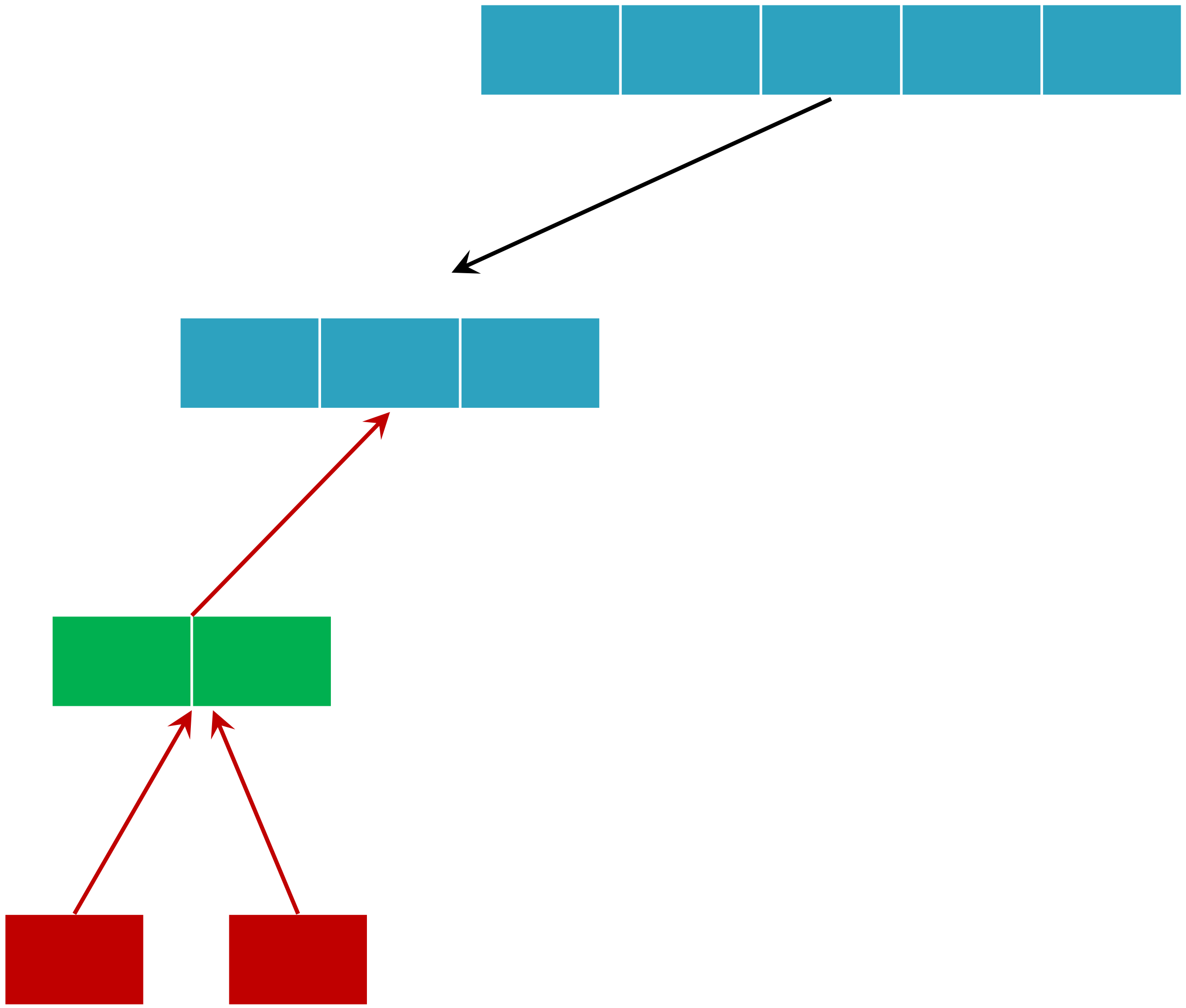
Metoda Divide et Impera



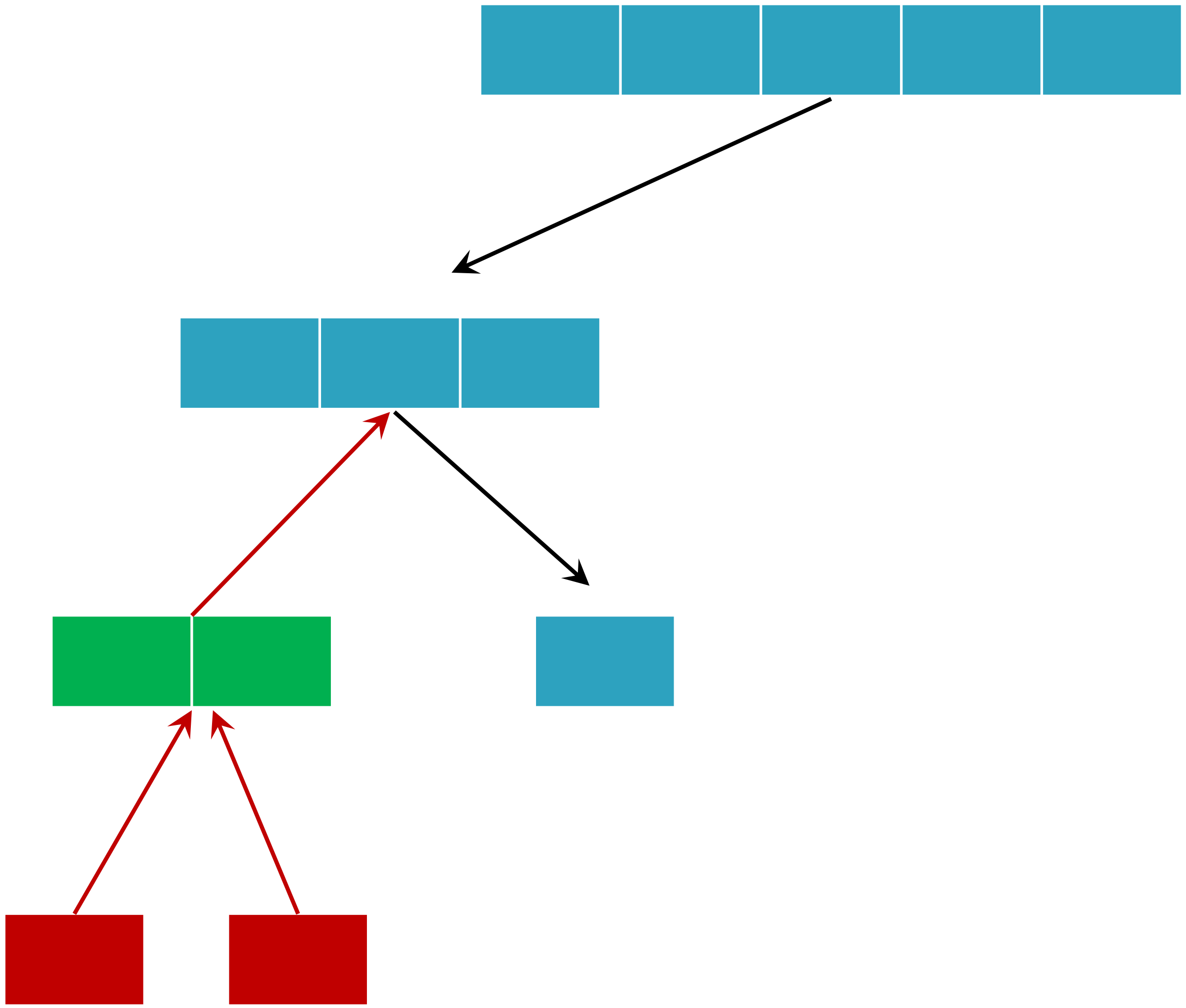
Metoda Divide et Impera



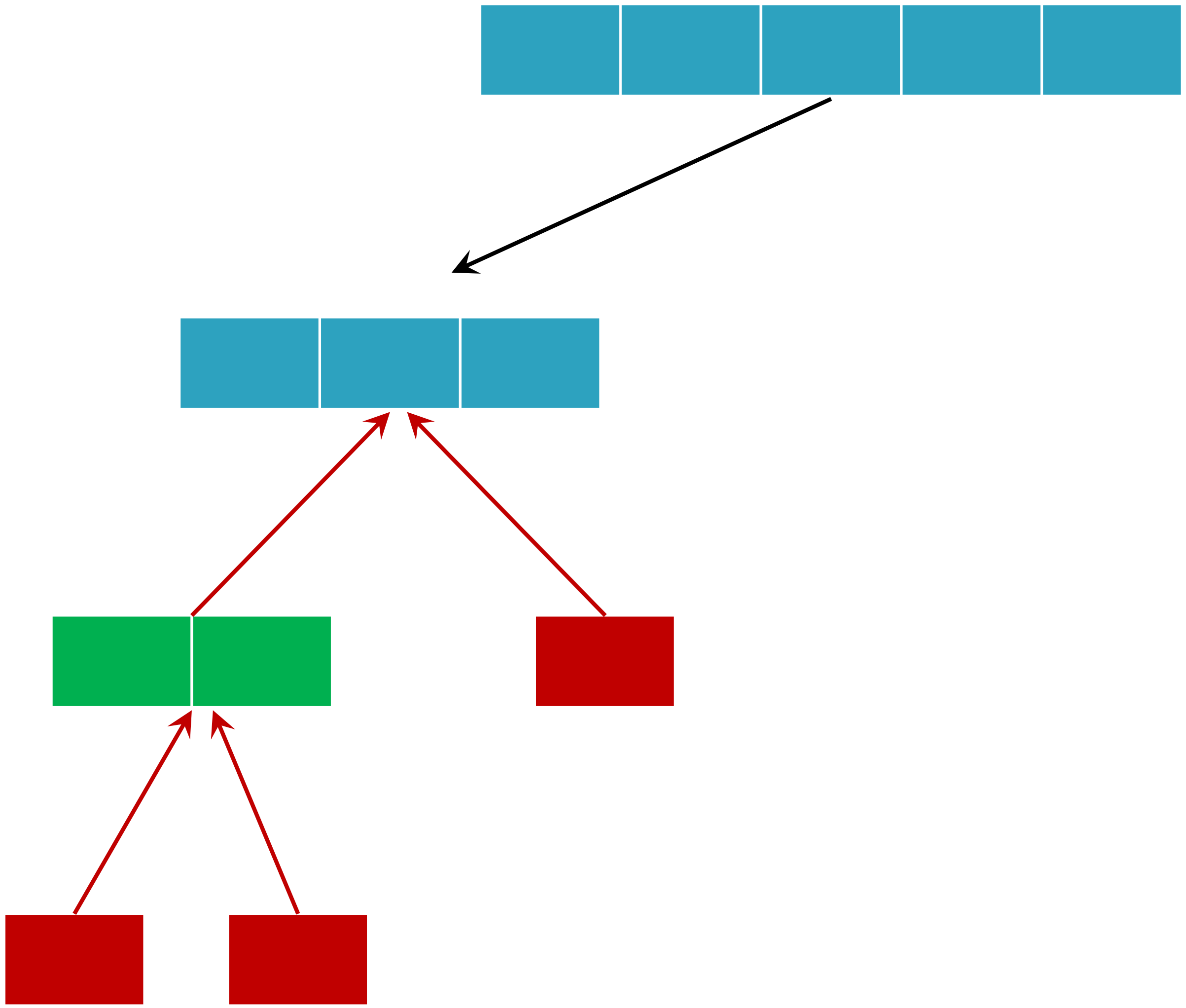
Metoda Divide et Impera



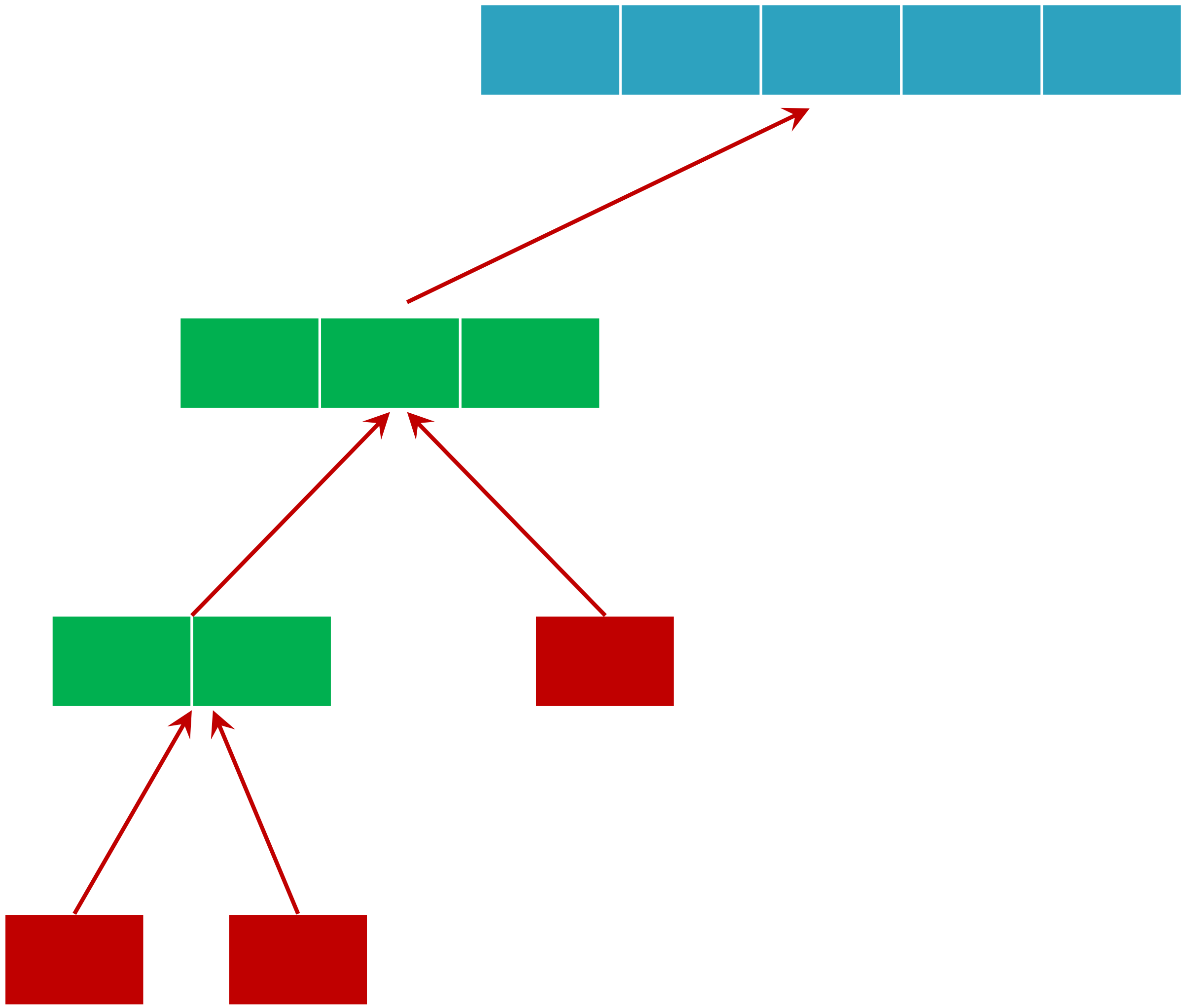
Metoda Divide et Impera



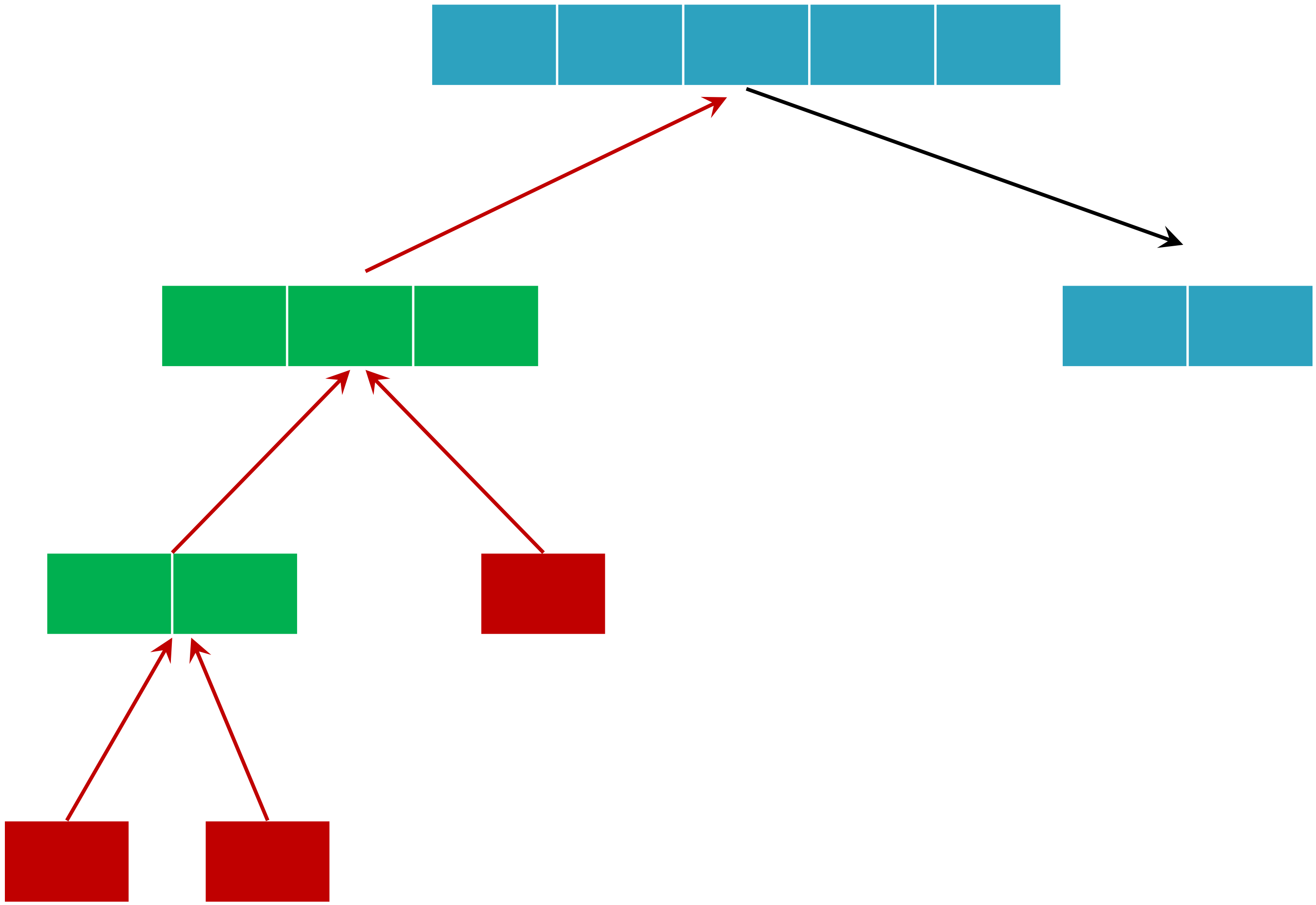
Metoda Divide et Impera



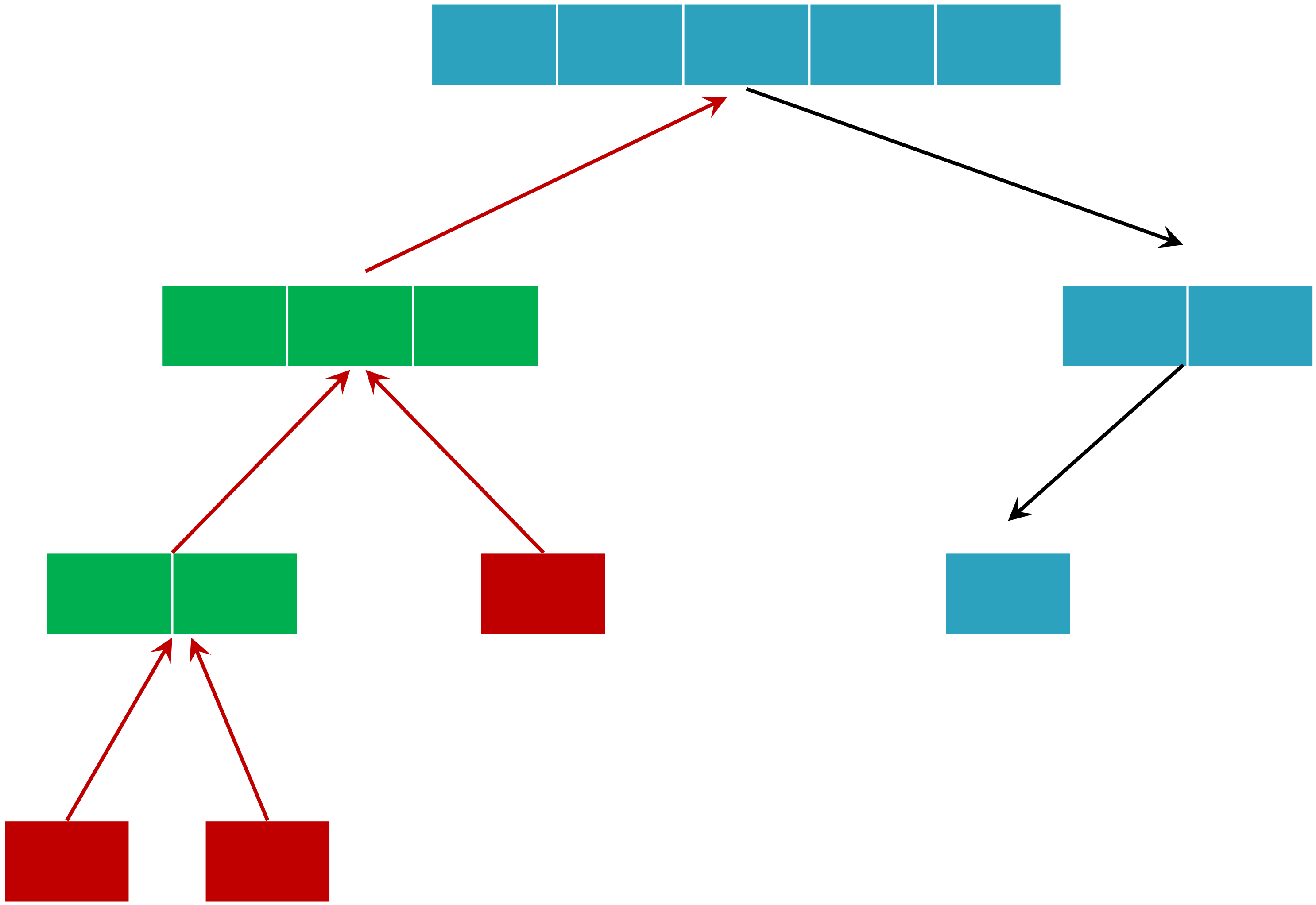
Metoda Divide et Impera



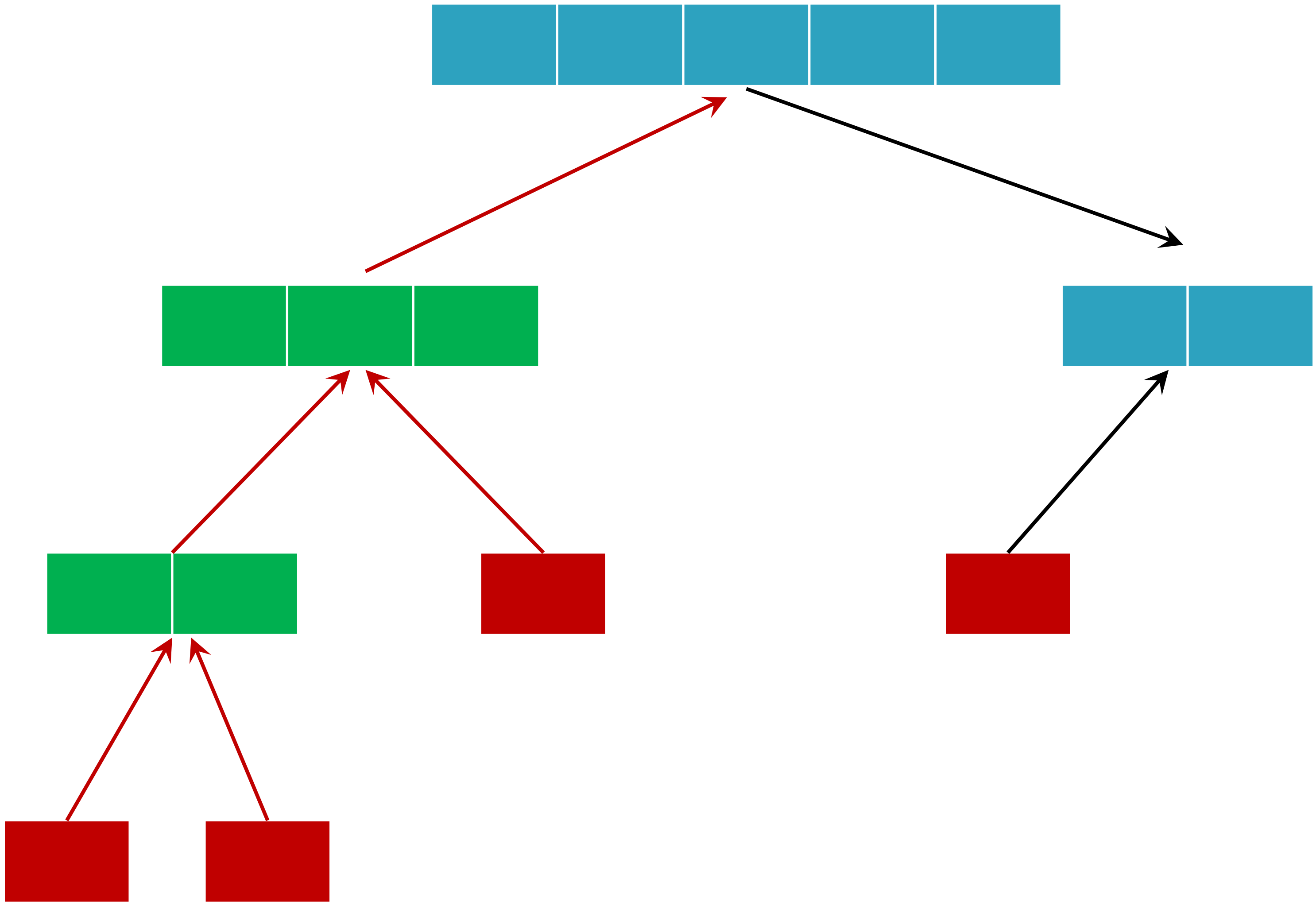
Metoda Divide et Impera



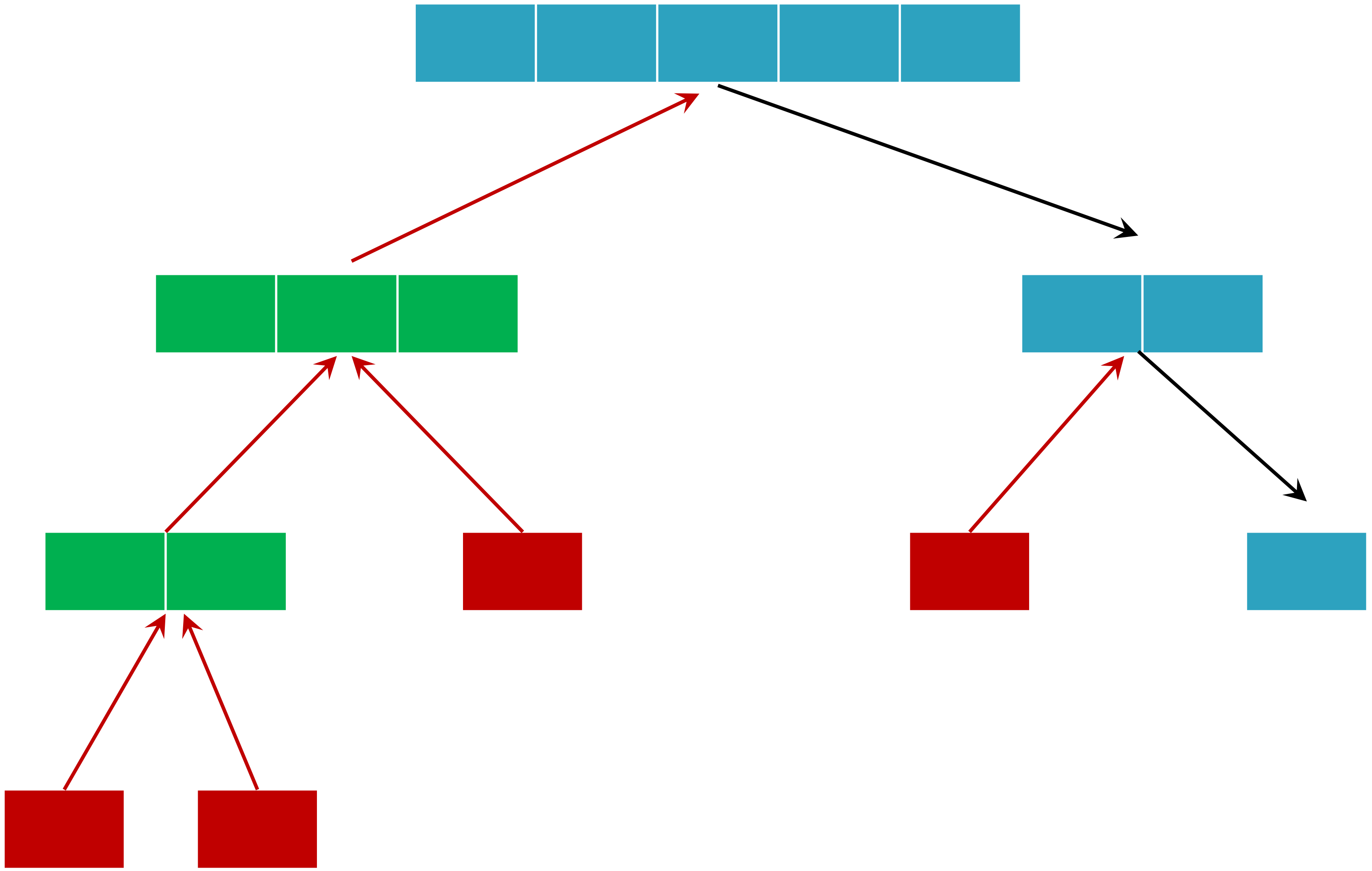
Metoda Divide et Impera



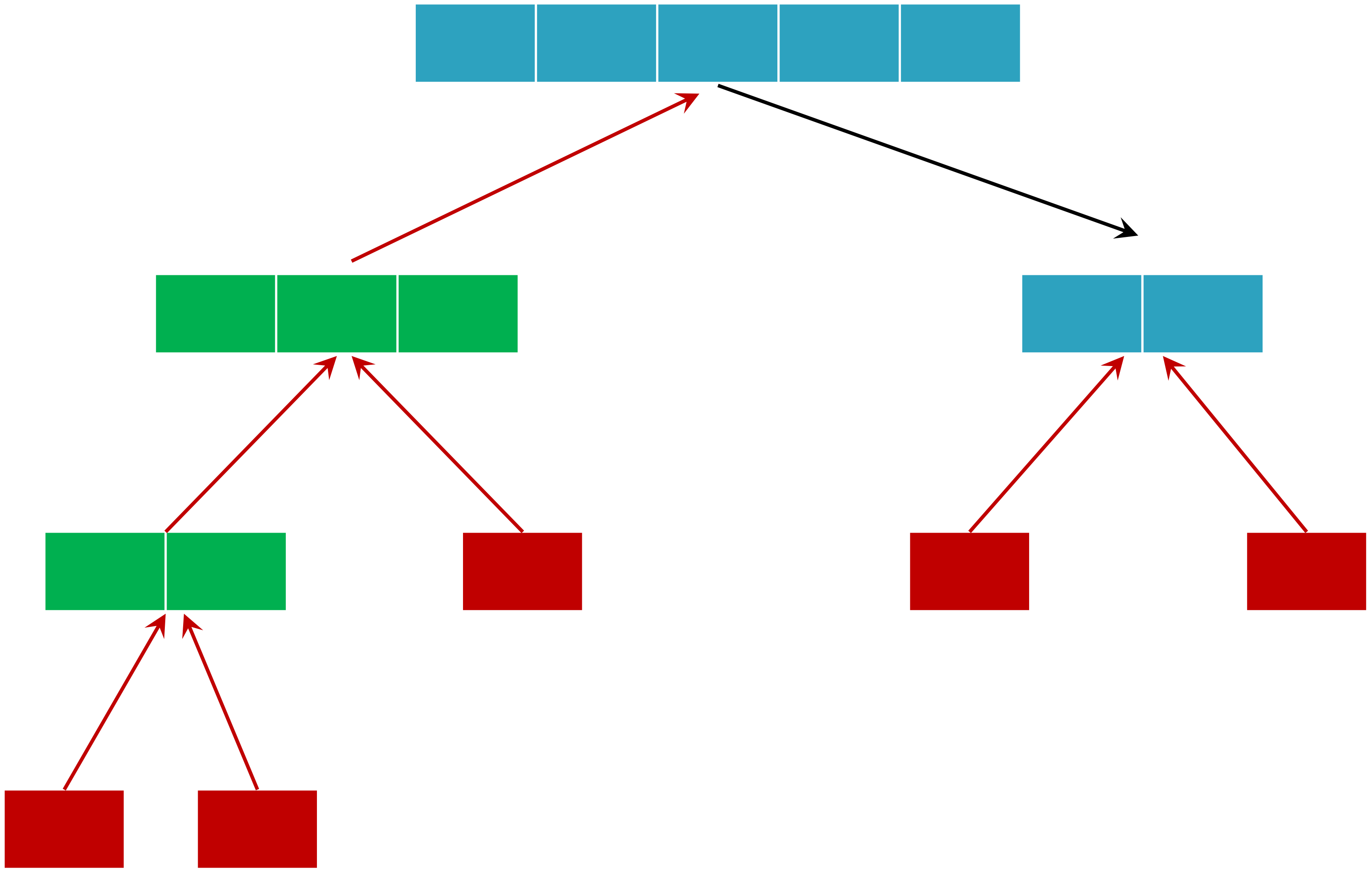
Metoda Divide et Impera



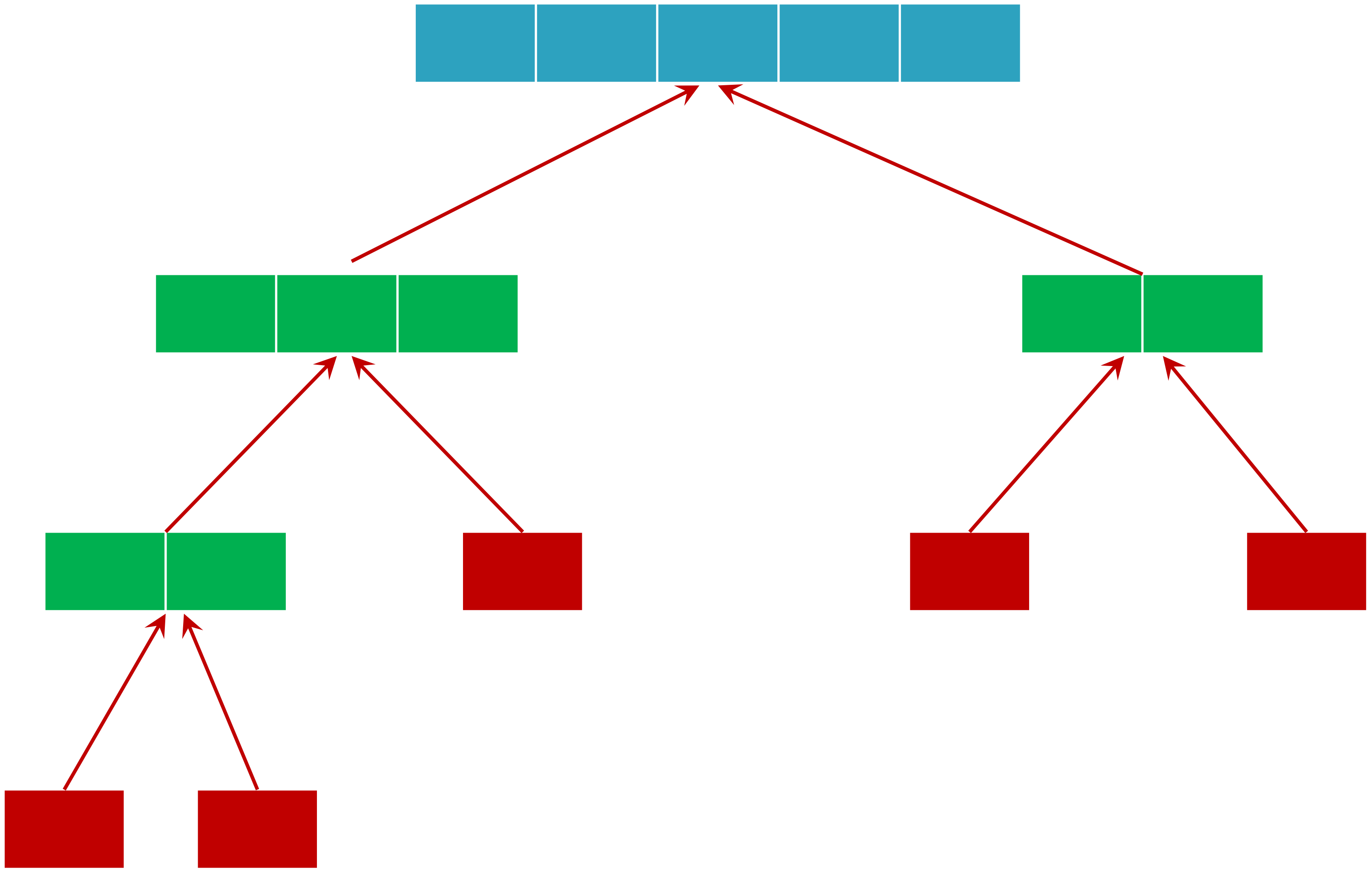
Metoda Divide et Impera



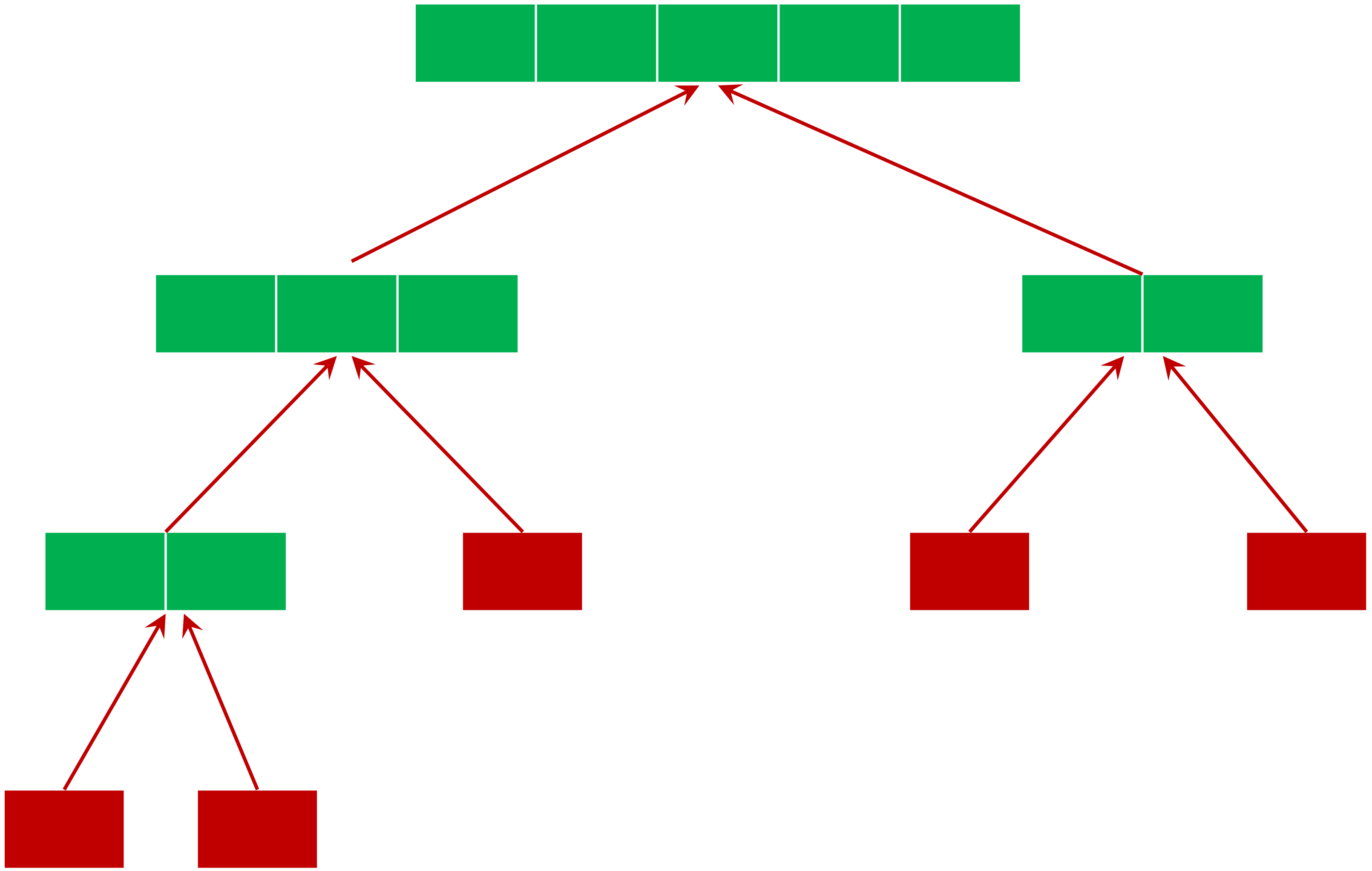
Metoda Divide et Impera



Metoda Divide et Impera



Metoda Divide et Impera

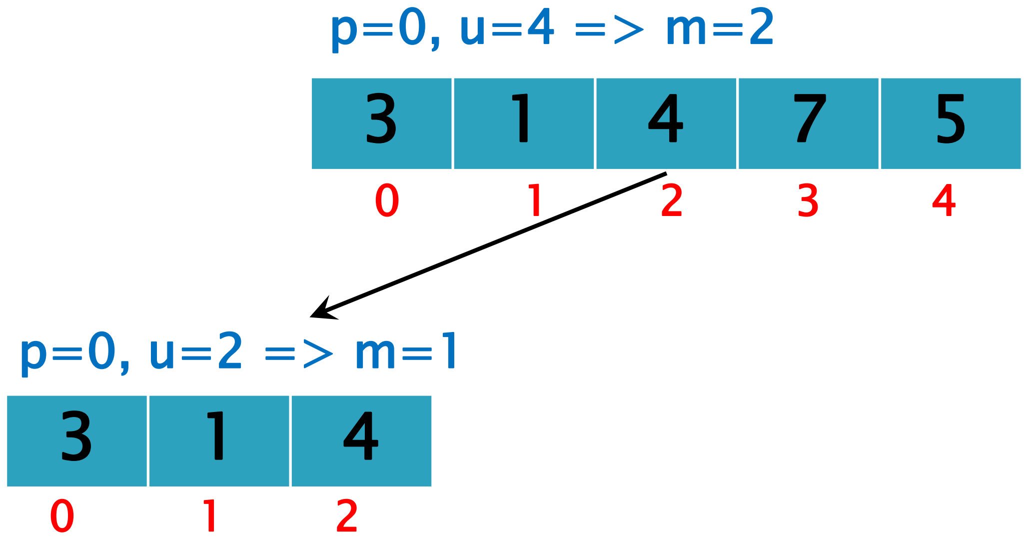


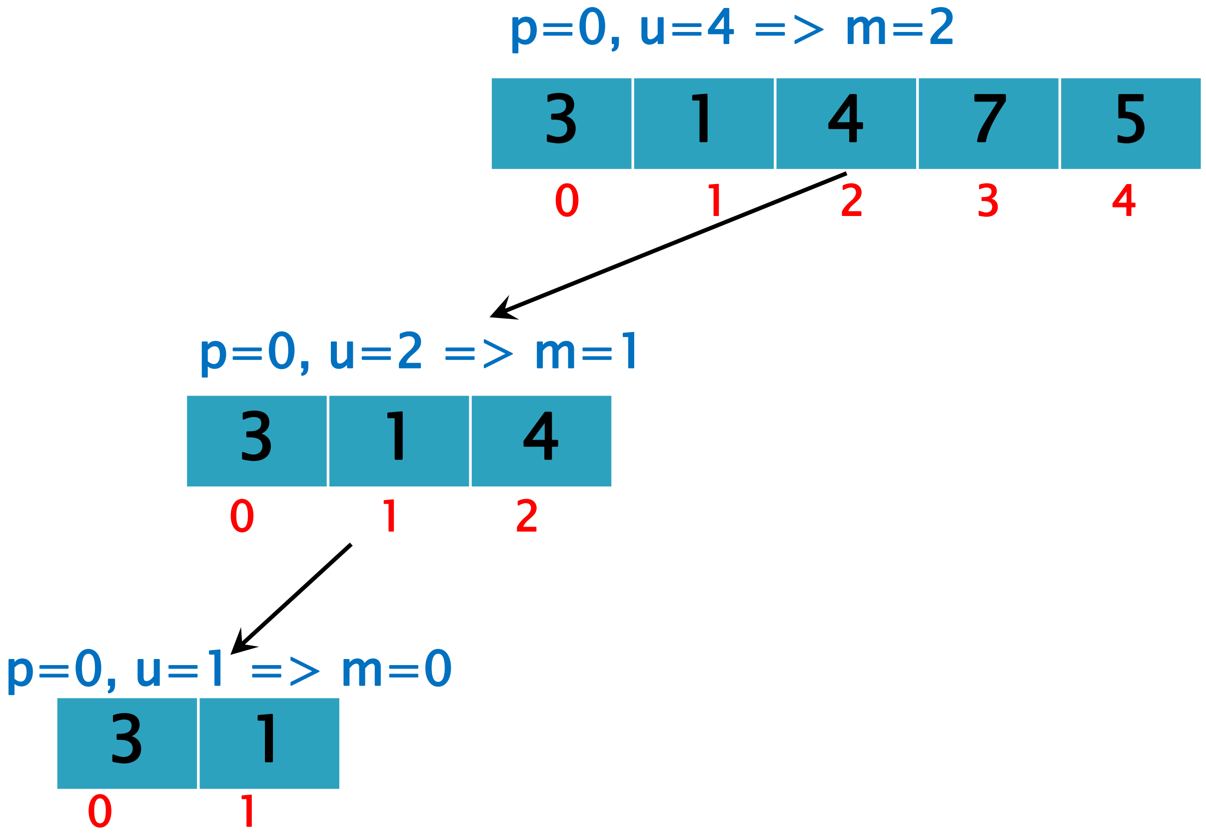
Exemplu –maximul elementelor unui vector

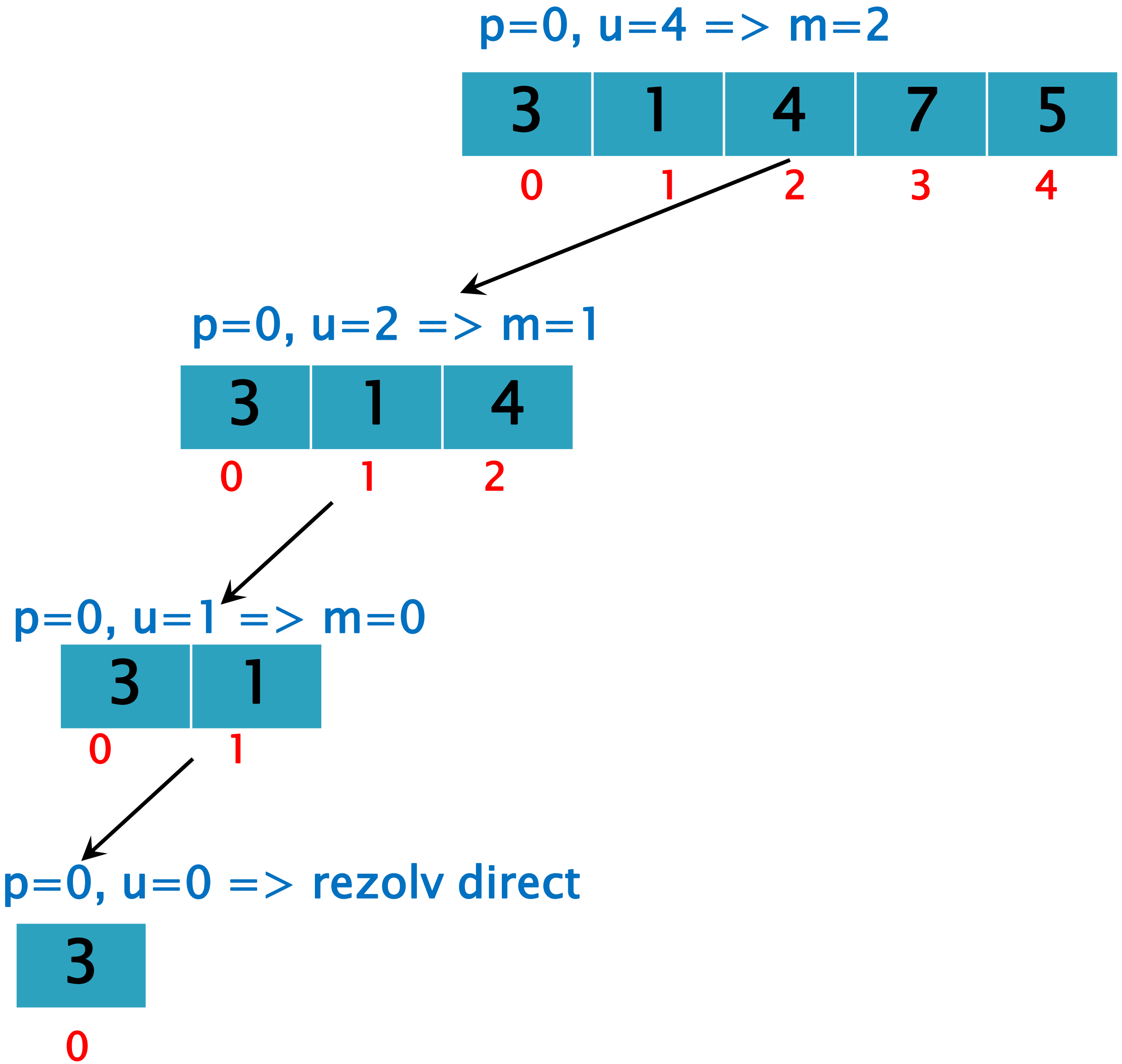
```
def DIMax(v, p, u):
    if p == u:
        return v[p]
    m = (p+u)//2
    r1 = DIMax(v, p, m)
    r2 = DIMax(v, m+1, u)
    if r1 > r2:
        return r1
    else:
        return r2

v = [3,1,4,7,5]
DIMax(v, 0, len(v)-1)
```

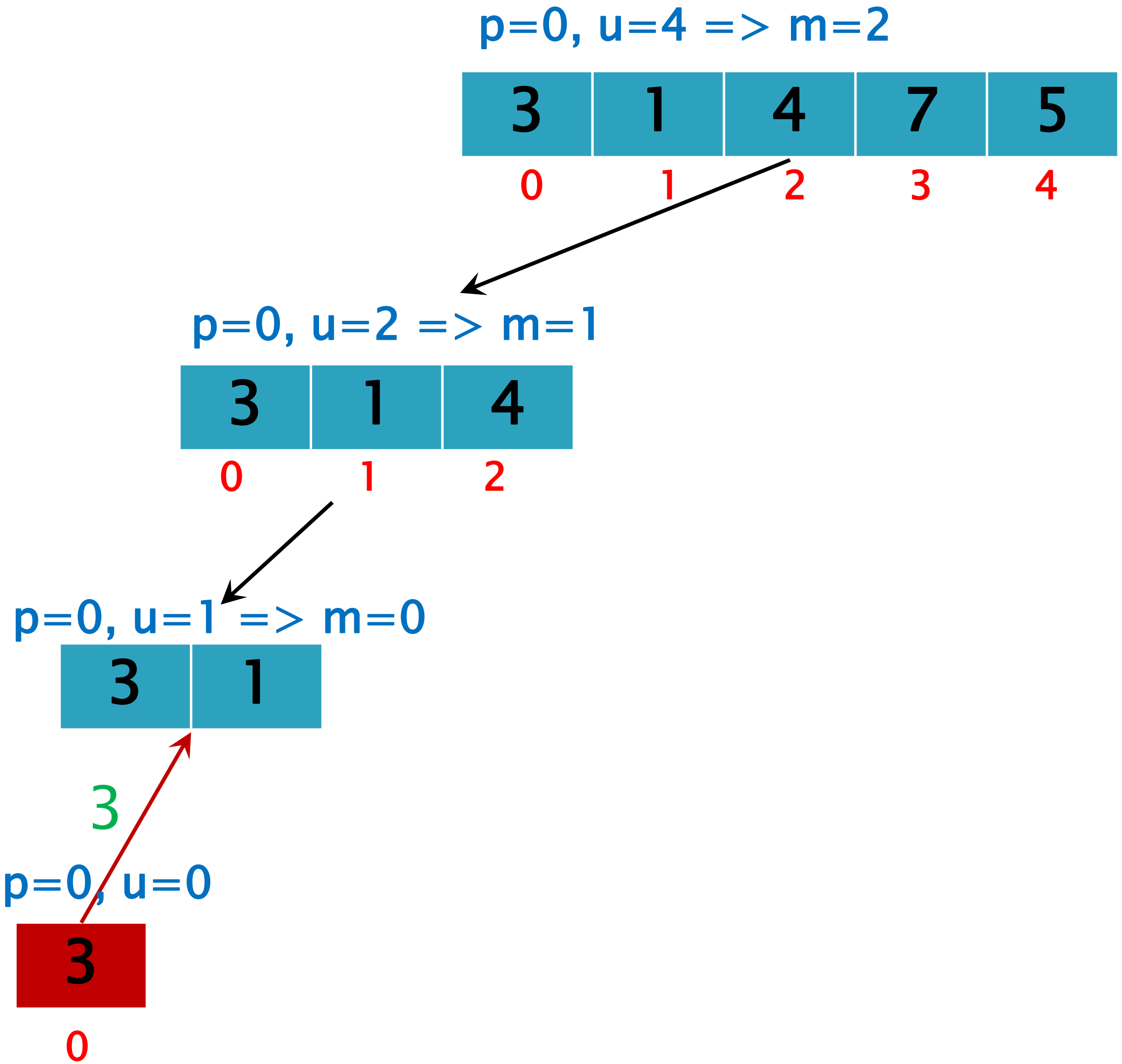
p=0, u=4 => m=2

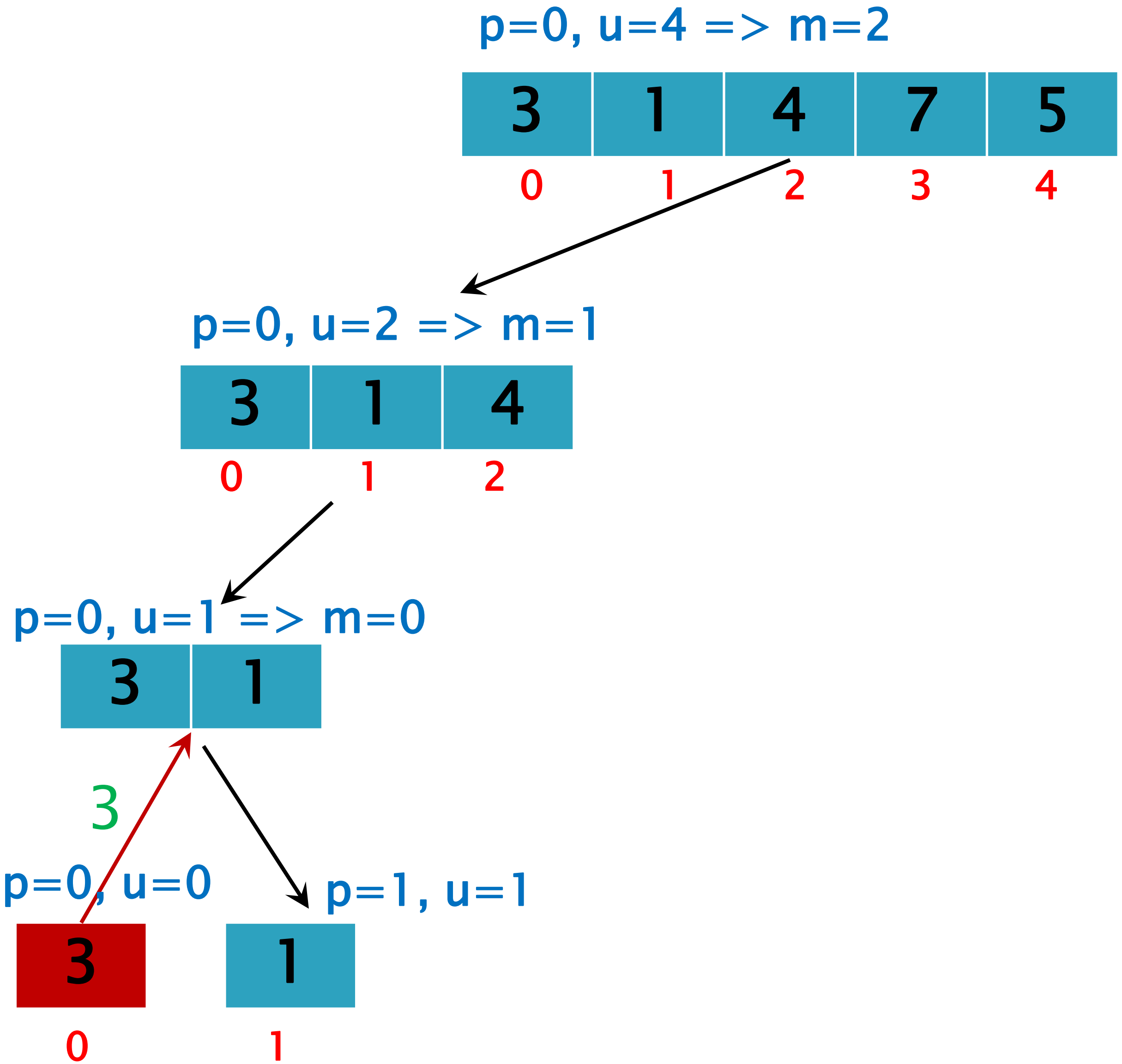
3	1	4	7	5
0	1	2	3	4

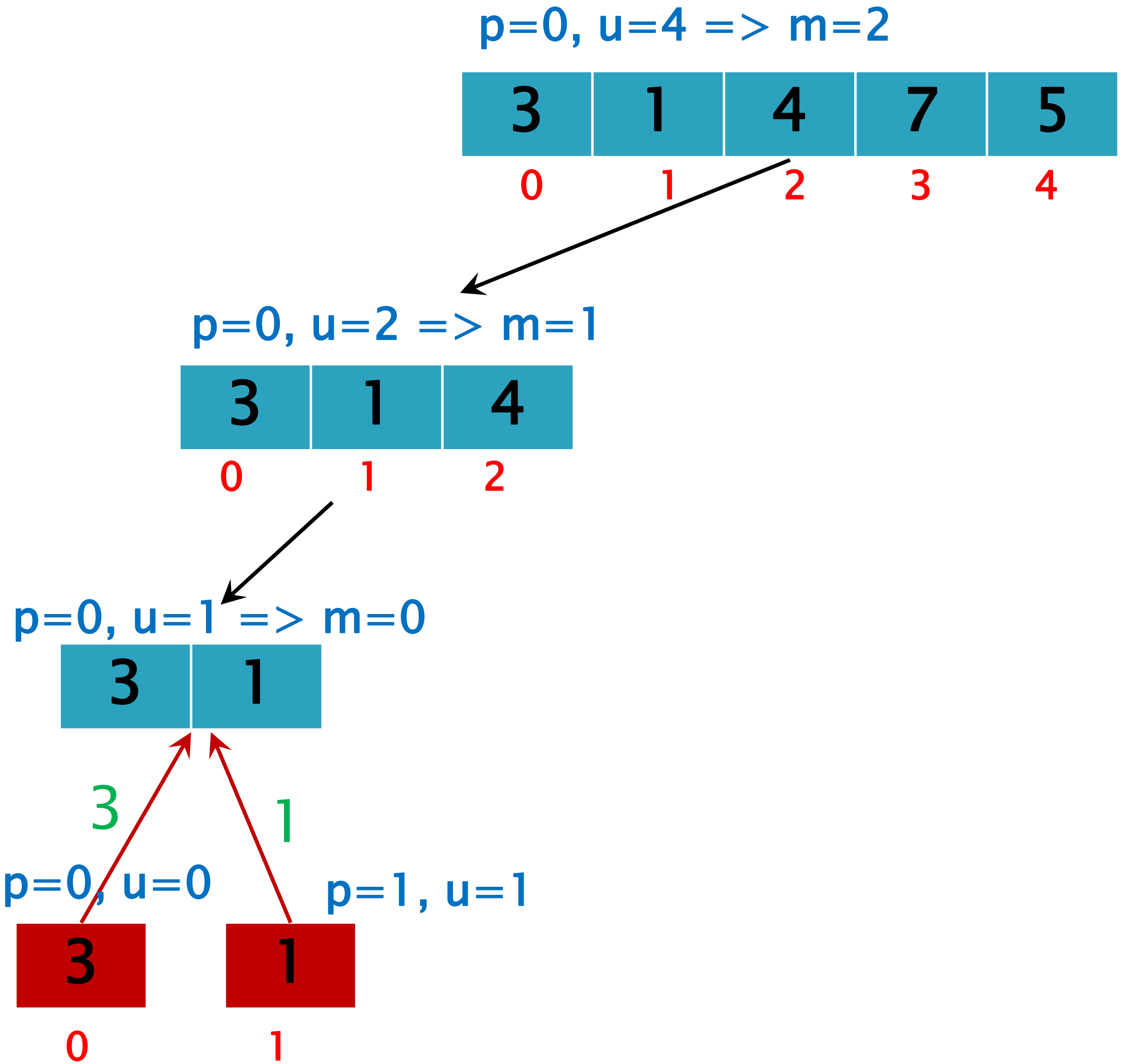

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2

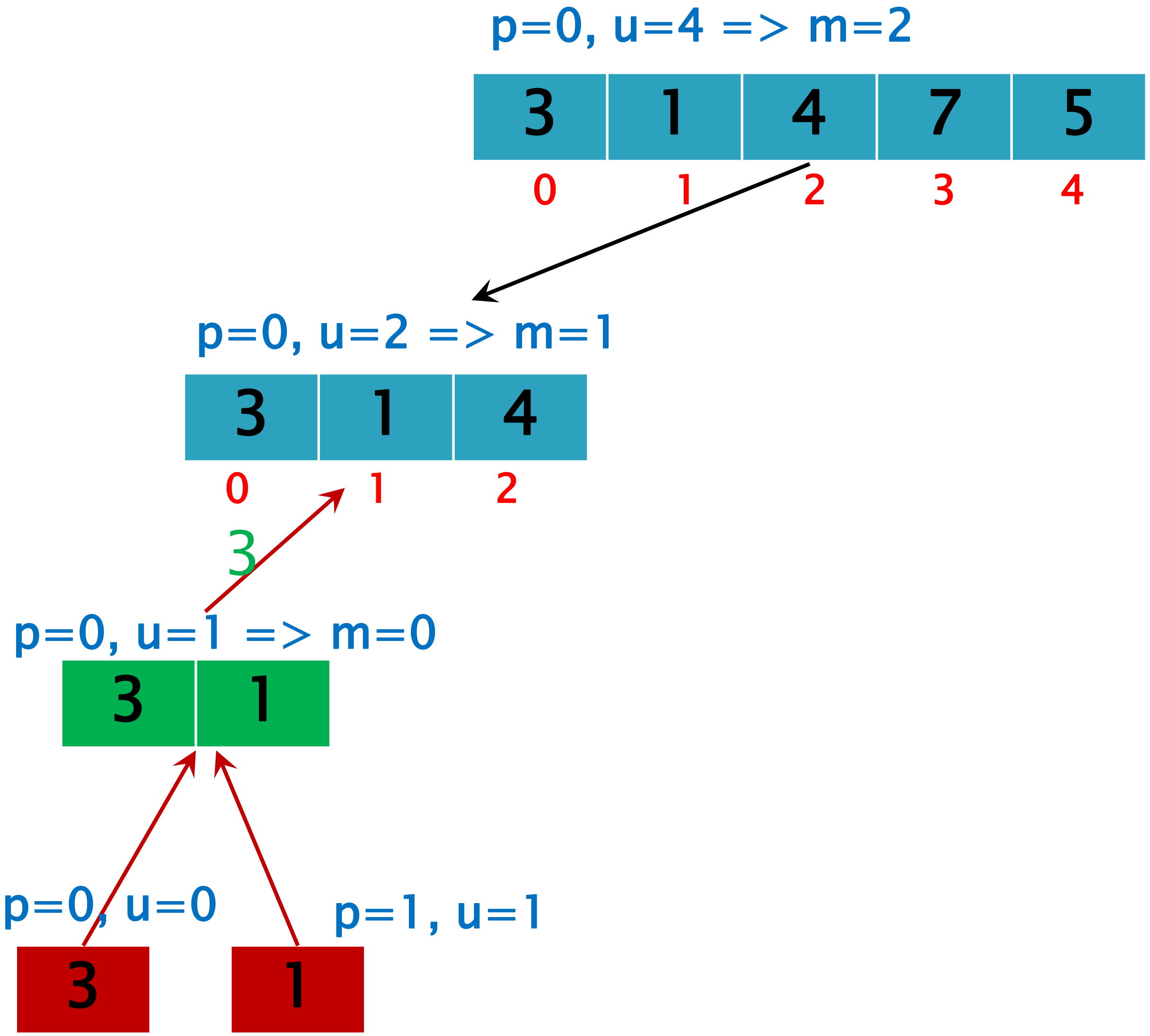

p=0, u=4 => m=2
3
1
4
7
5
0
1
2
3
4
p=0, u=2 => m=1
3
1
4
0
1
2
p=0, u=1 => m=0
3
1
0
1

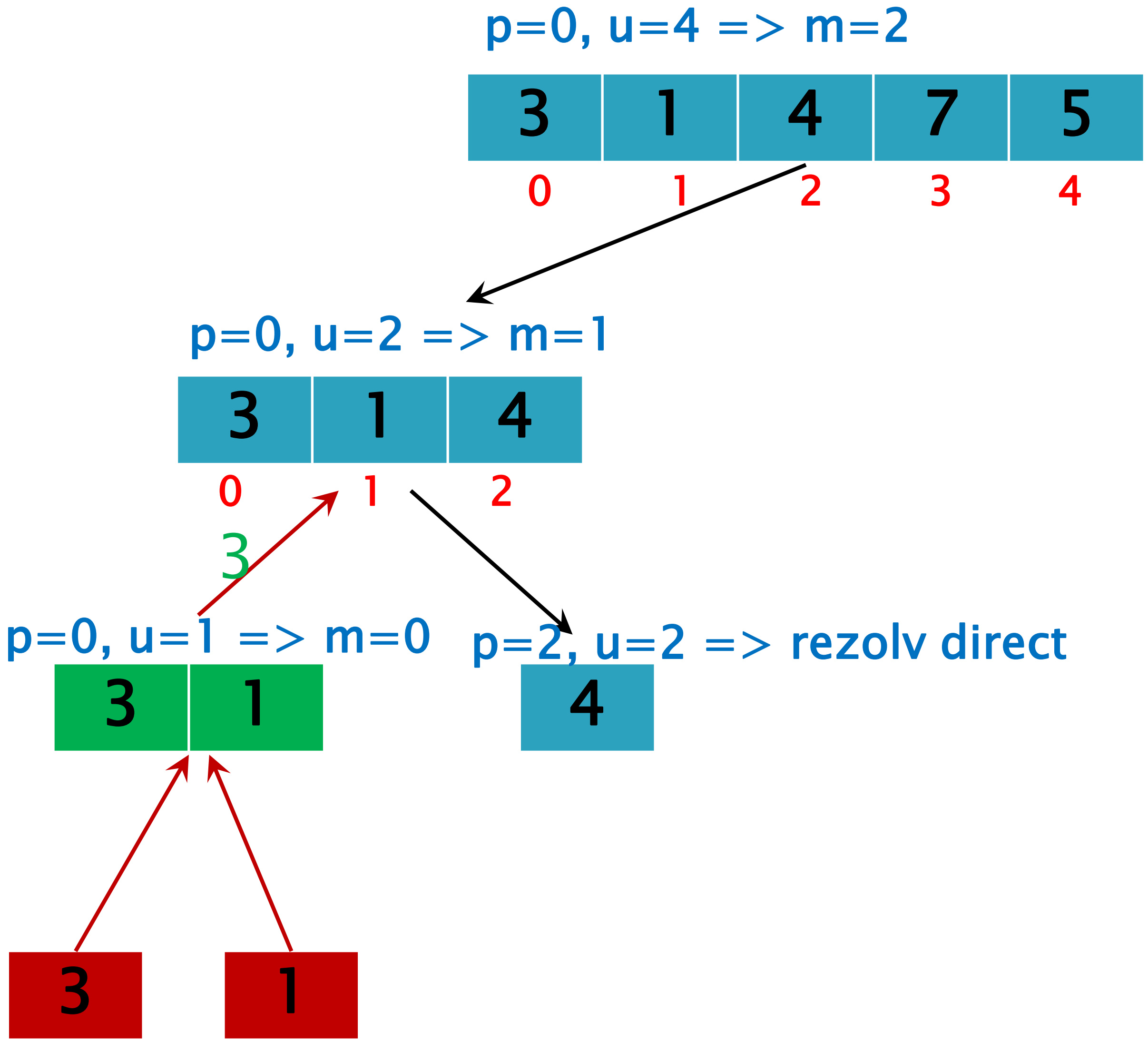

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
p=0, u=1 => m=0
3 1
0 1
p=0, u=0 => rezolv direct
3
0

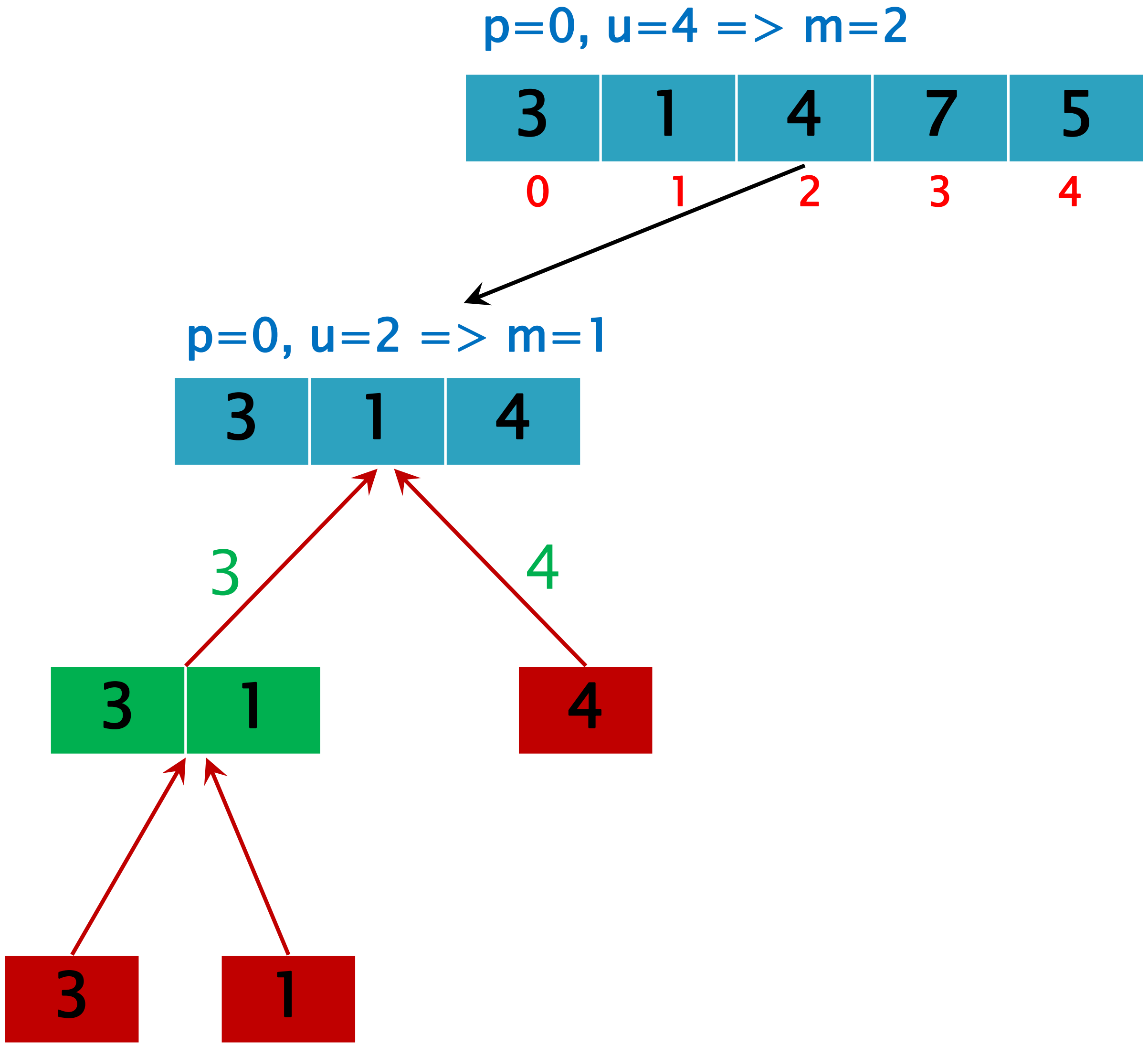

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
p=0, u=1 => m=0
3 1
3
p=0, u=0
3
0

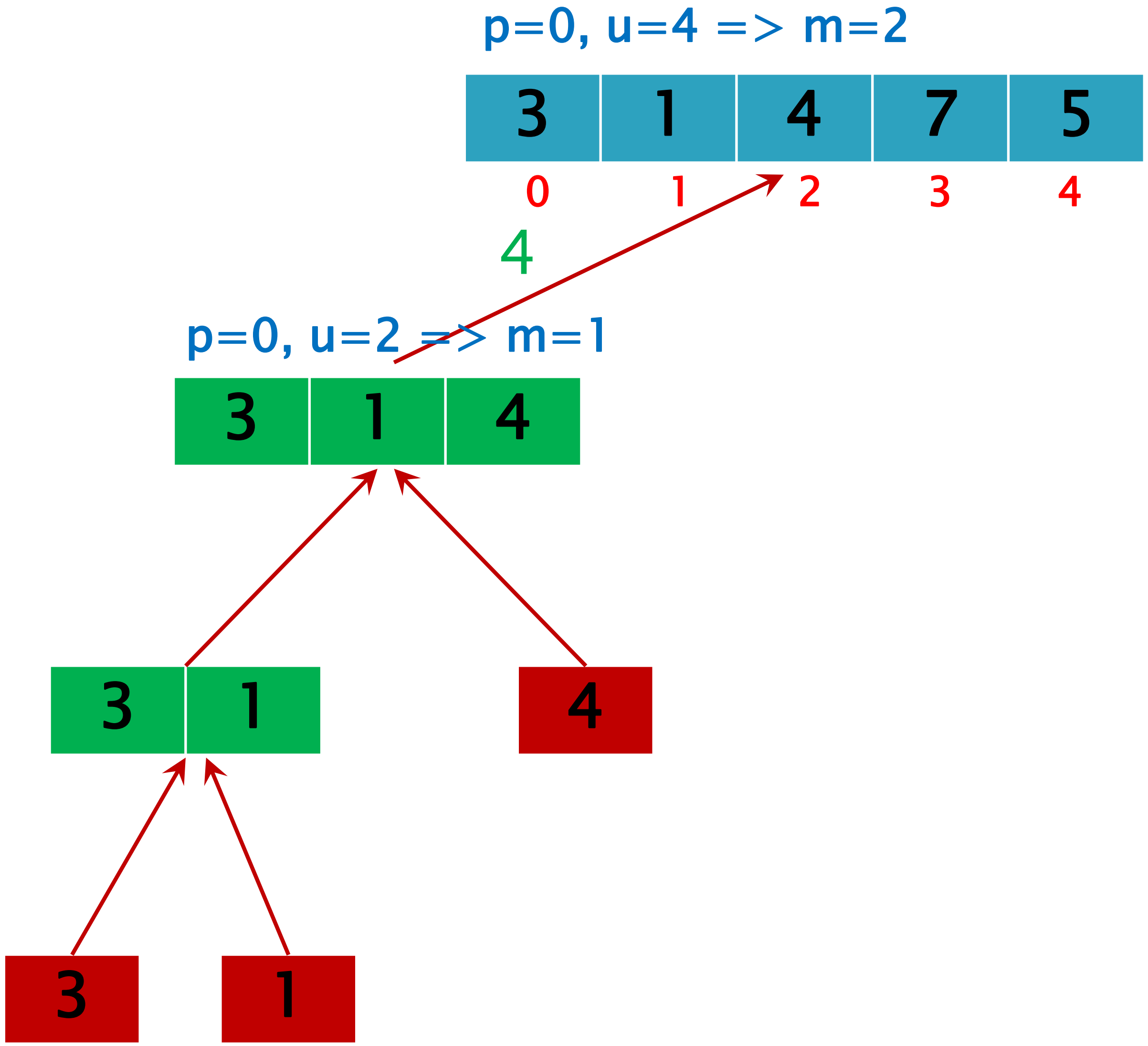

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
p=0, u=1 => m=0
3 1
3
p=0, u=0
p=1, u=1
3
1
0
1

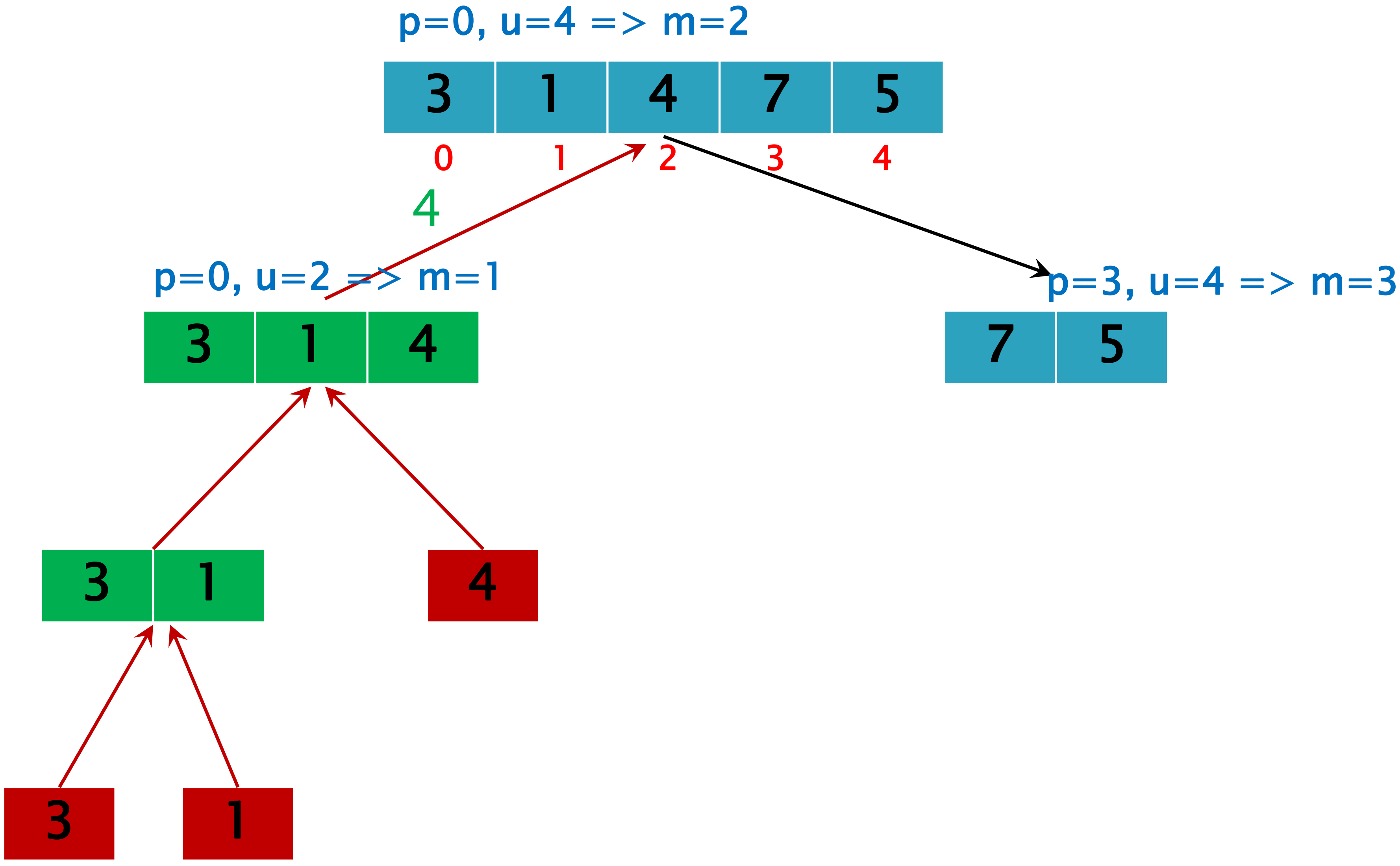

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
p=0, u=1 => m=0
3 1
3
1
p=0, u=0
p=1, u=1
3
1
0
1

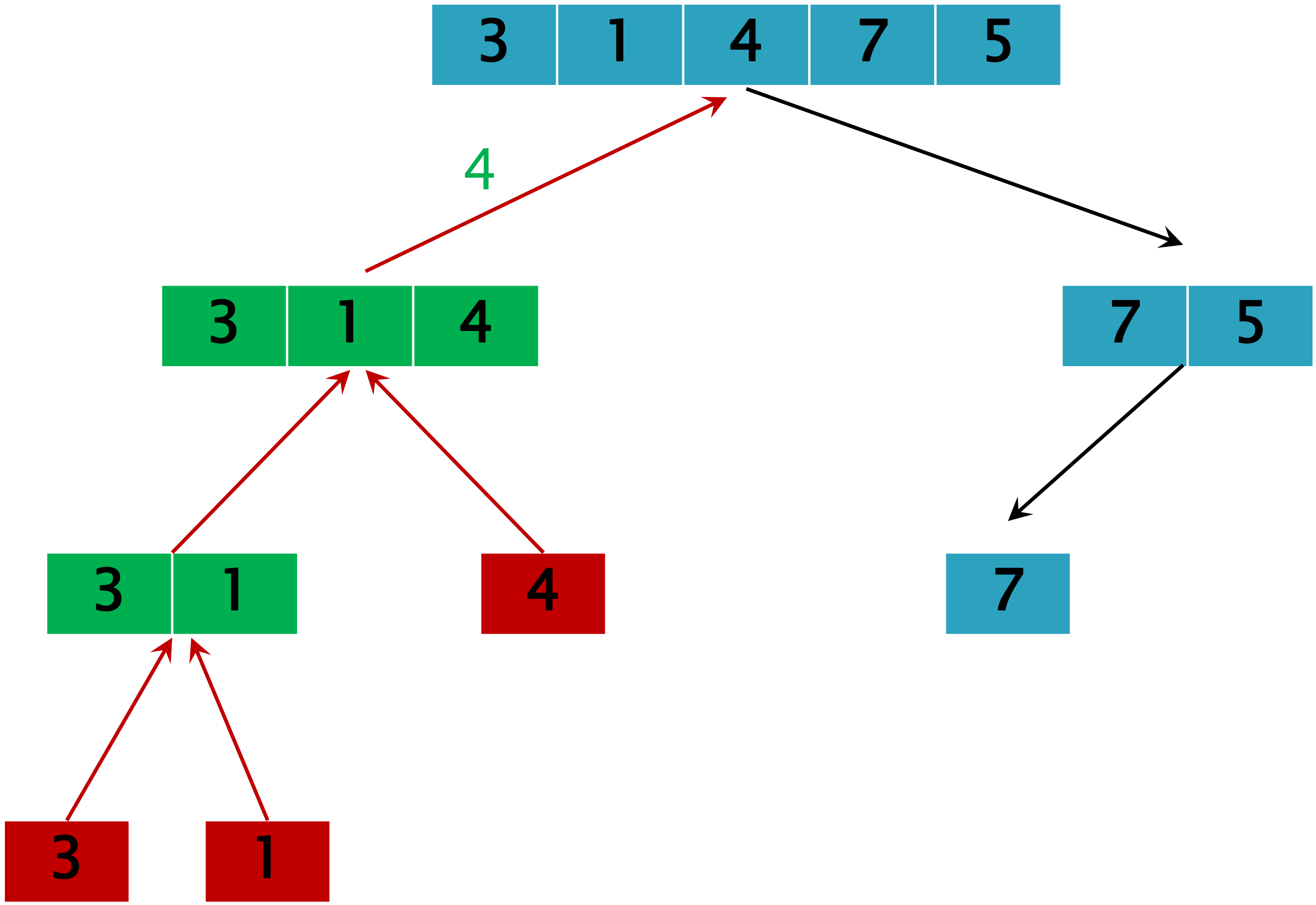

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
3
p=0, u=1 => m=0
3 1
p=0, u=0
3
p=1, u=1
1

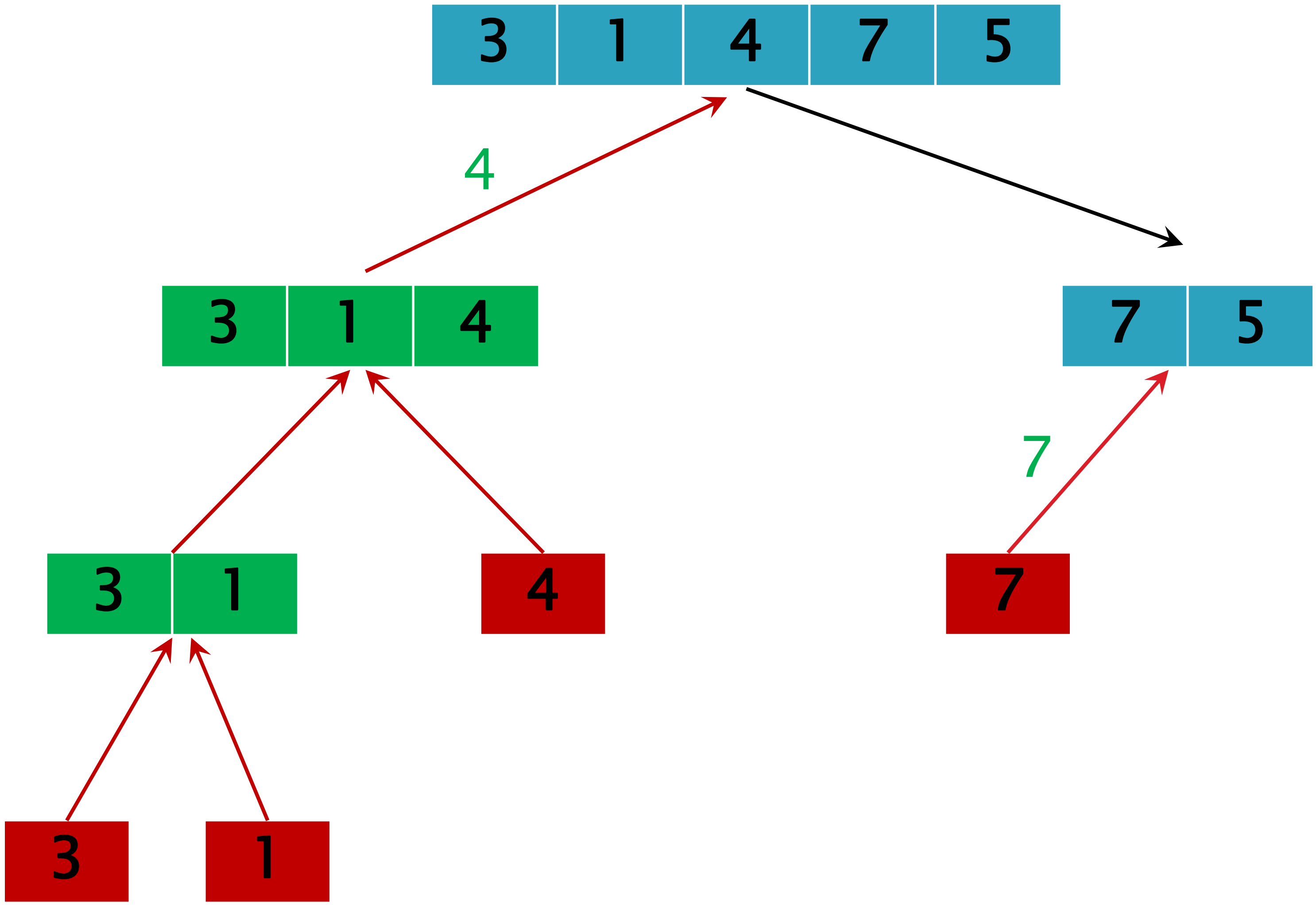

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
0 1 2
3
p=0, u=1 => m=0
p=2, u=2 => rezolv direct
3 1
4
3
1

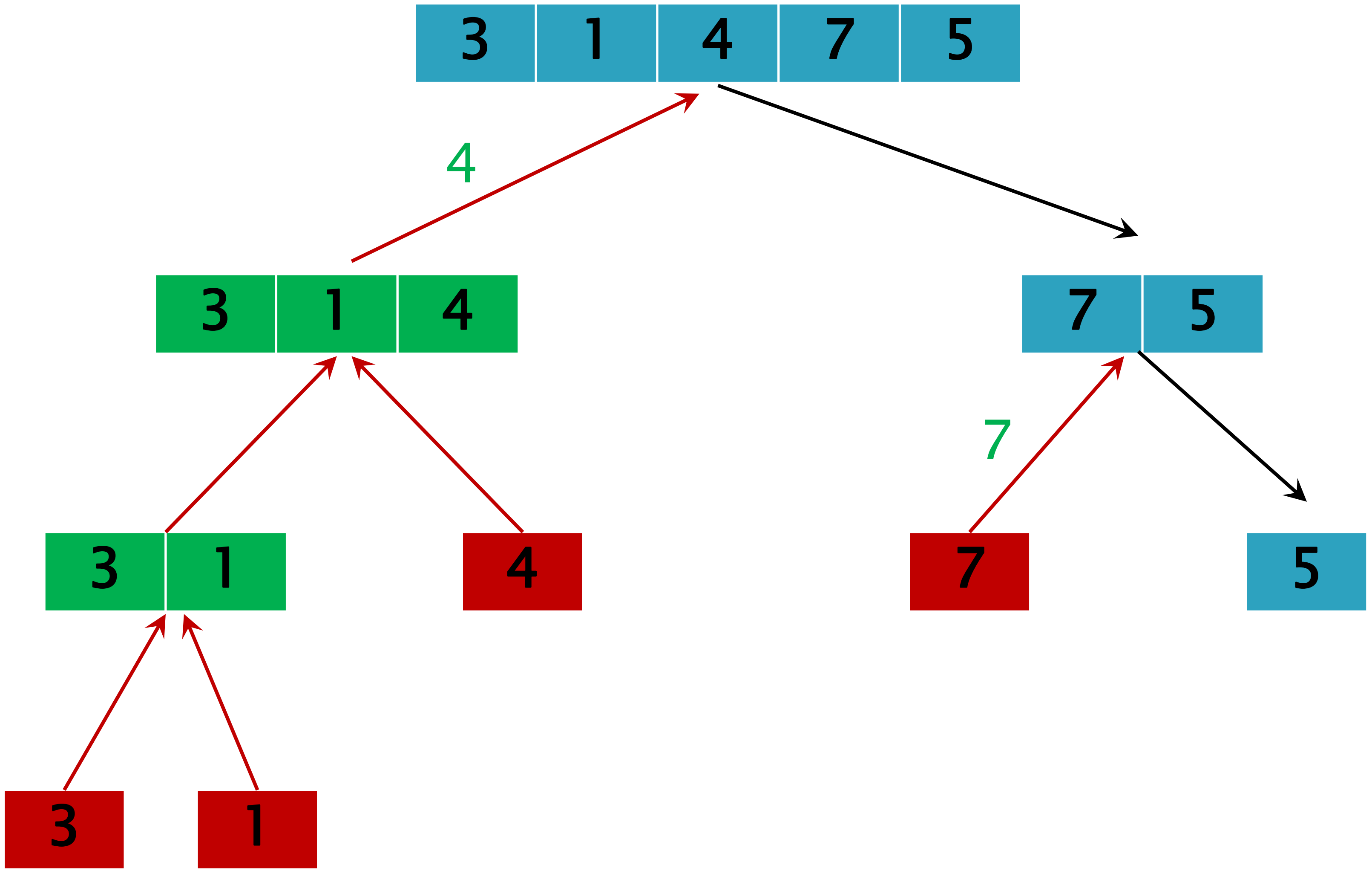

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
p=0, u=2 => m=1
3 1 4
3
4
3 1
4
3
1

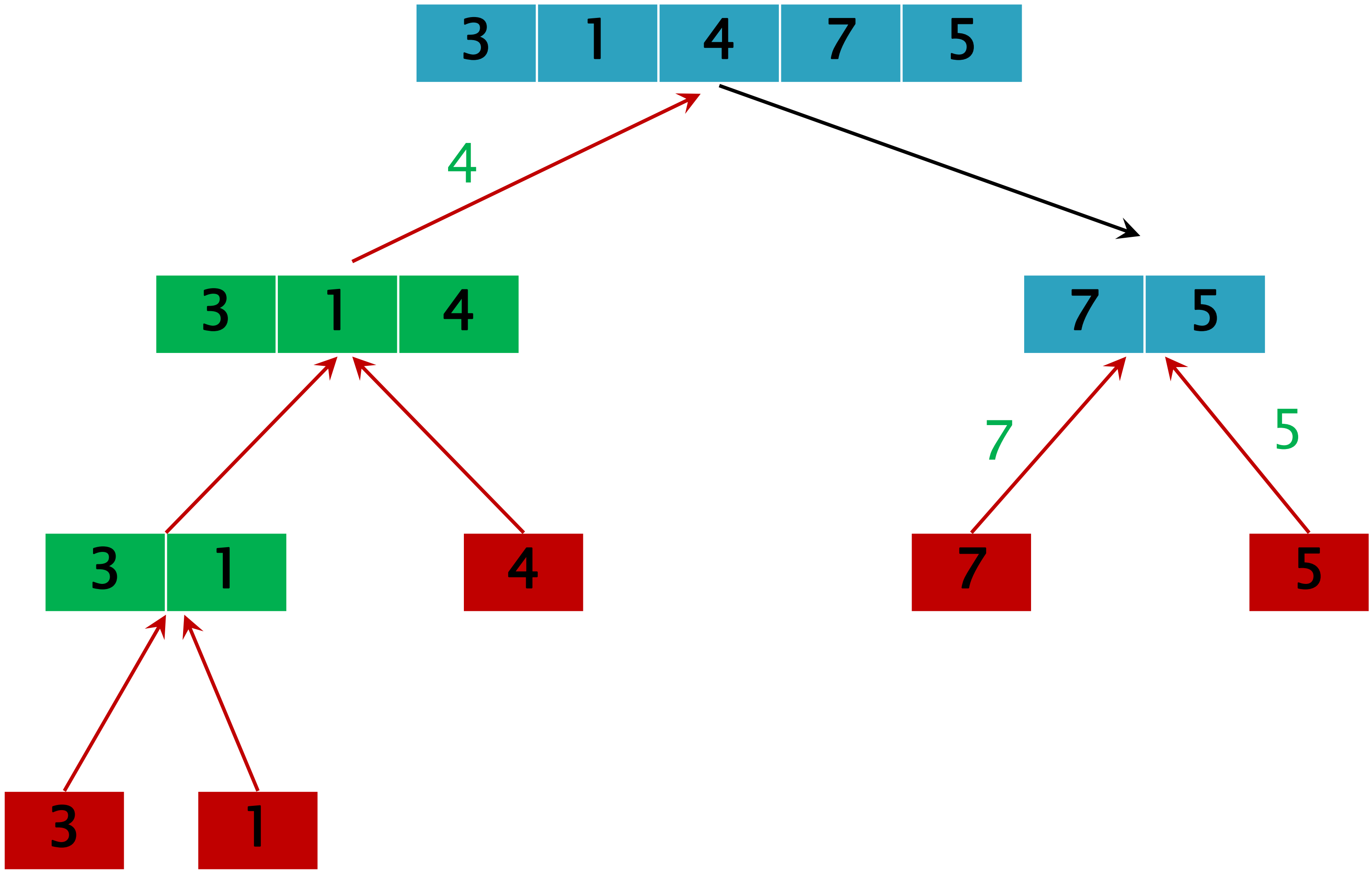

p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
4
p=0, u=2 => m=1
3 1 4
3 1
4
3
1


p=0, u=4 => m=2
3 1 4 7 5
0 1 2 3 4
4
p=0, u=2 => m=1
3 1 4
p=3, u=4 => m=3
7 5
3 1
4
3
1

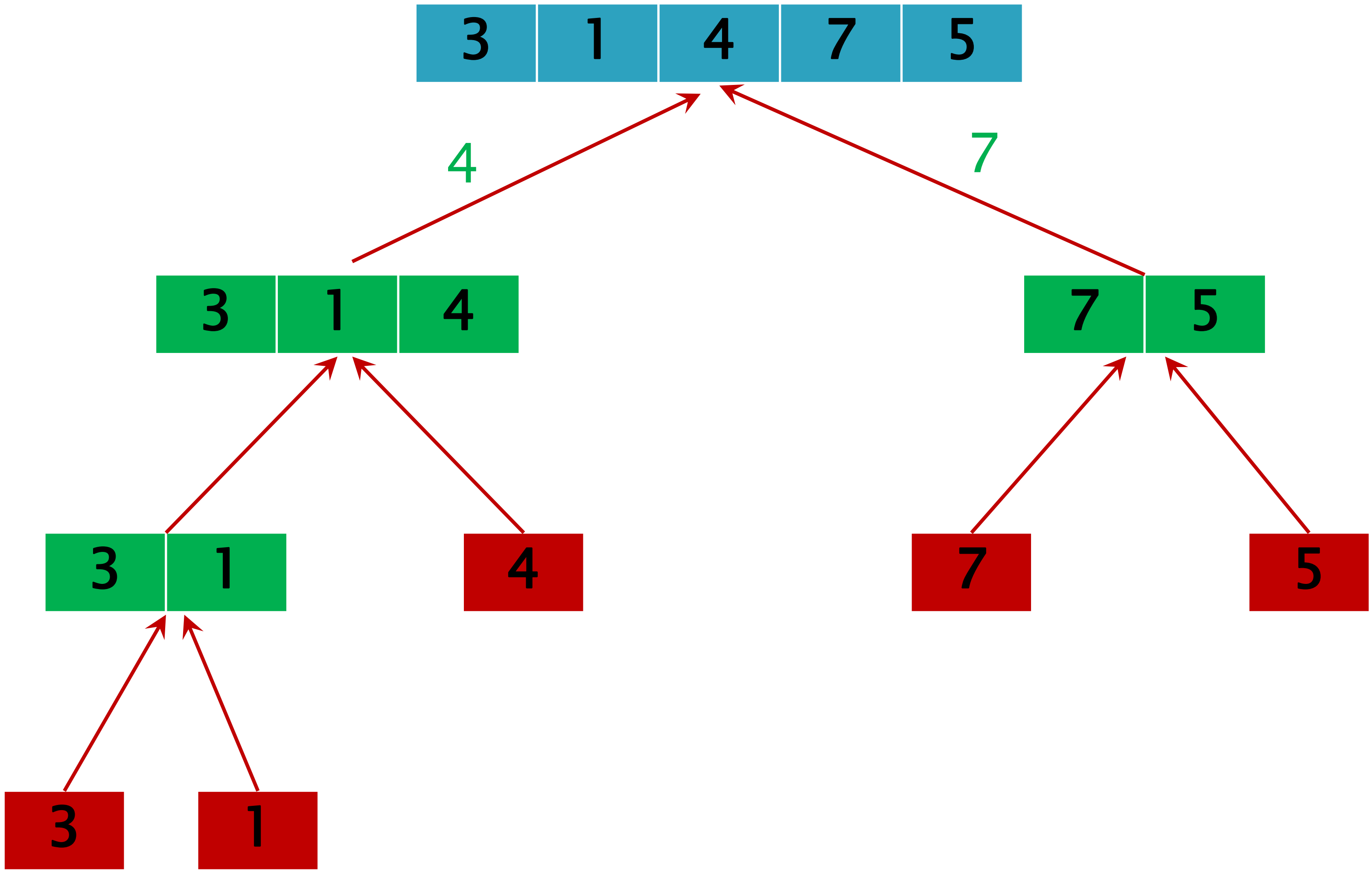

3 1 4 7 5
4
3 1 4
7 5
3 1
4
7
3
1


3 1 4 7 5
4
3 1 4
7 5
7
3 1
4
7
3
1

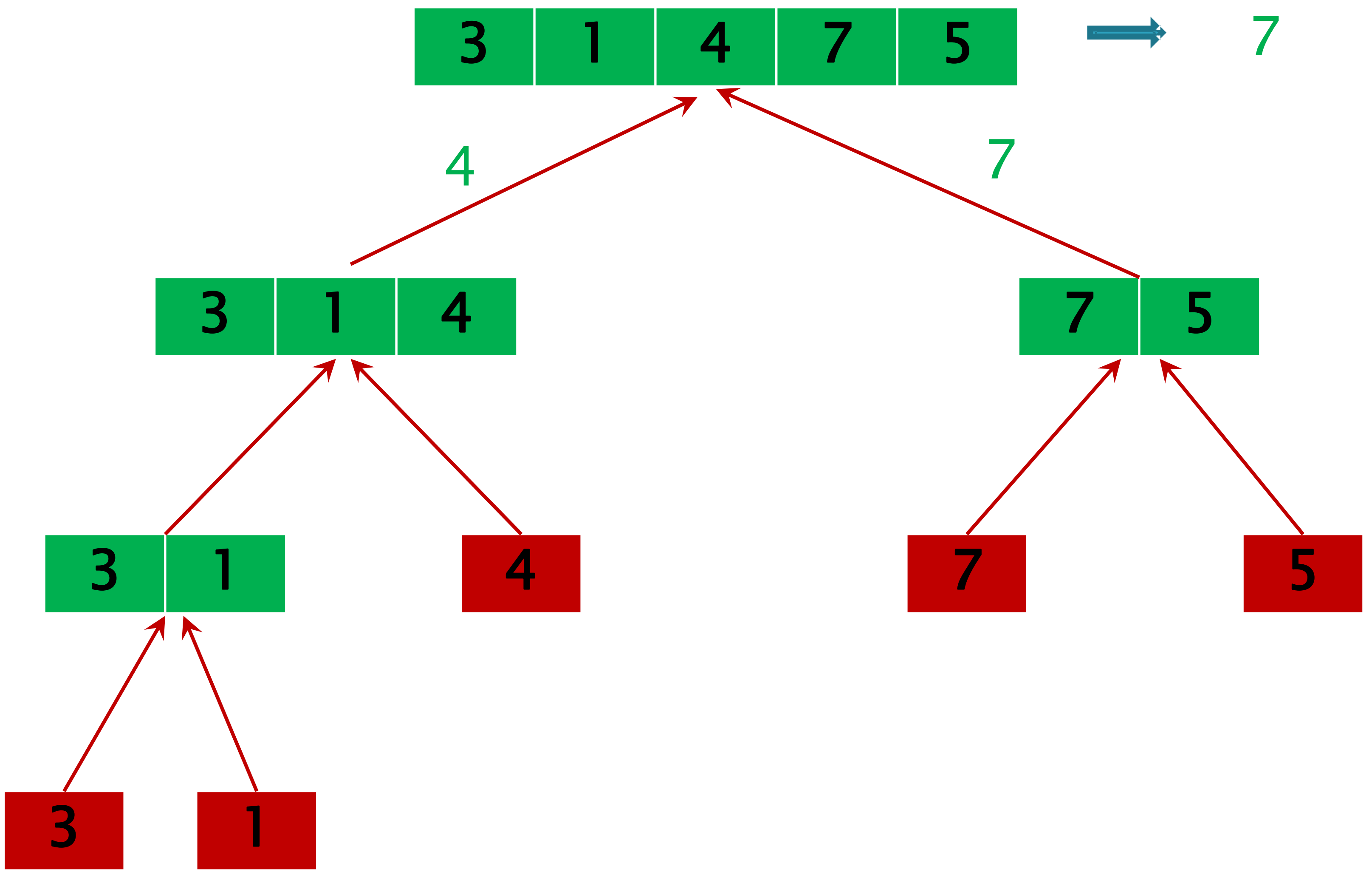

3 1 4 7 5
4
3 1 4
7 5
7
3 1
4
7
5
3
1



3 1 4 7 5
4
3 1 4
7 5
7
5
3 1
4
7
5
3
1



3 1 4 7 5
4
7
3 1 4
7 5
3 1
4
7
5
3
1



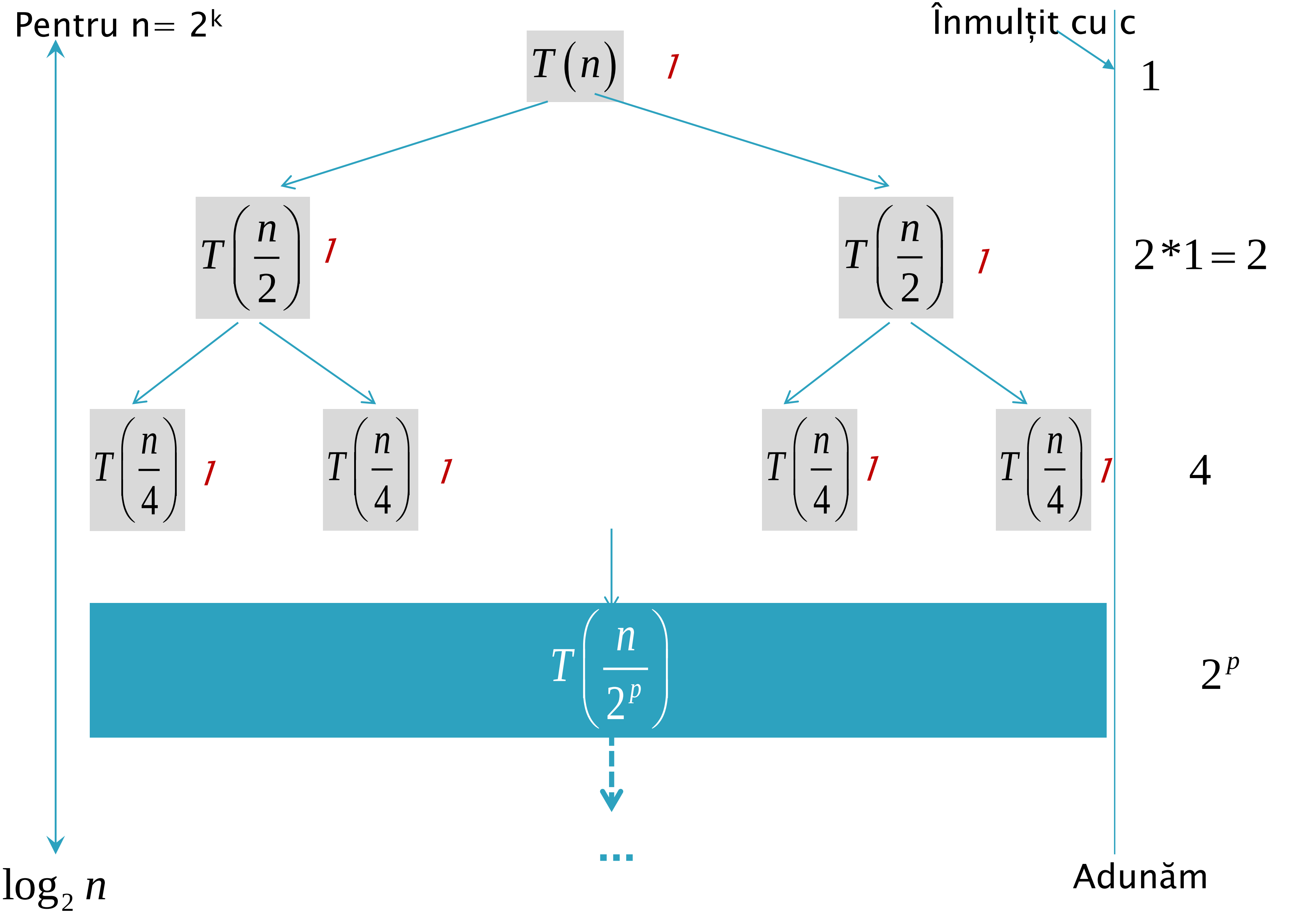
3 1 4 7 5
7
4
7
3 1 4
7 5
3 1
4
7
5
3
1

Exemplu –maximul elementelor unui vector

```
def DIMax(v, p, u):
    if p == u:
        return v[p]
    m = (p+u)//2
    r1 = DIMax(v, p, m)
    r2 = DIMax(v, m+1,u)
    if r1 > r2:
        return r1
    else:
        return r2

v = [3,1,4,7,5]
DIMax(v, 0, len(v)-1)
```

Aceeași “eficiență” (complexitate) ca și varianta nerecursivă

(c*n comparații, c –constantă)



$$1+2+2^2+\ldots+2^k \Rightarrow O(2^k)=O(n)$$

Căutarea binară

Căutarea binară

- **Se consideră vectorul $a=(a_0,a_1,\ldots,a_{n-1})$ ordonat crescător şi o valoare x. Se cere să se determine dacă x este element în vectorul a.**

- Mai exact căutăm perechea (b,i) dată de:
  - `(True,i)` dacă $a_i = x$;
  - `(False,i)` dacă $a_i < x < a_{i+1}$

    (sau $i=-1$ daca $x< a_0$,

    $i = n-1$ daca $x > a_{n-1}$)

cautare_binara(x=17, p=0, u=8)

p=0, u=8 => m=4

1	2	4	6	11	15	17	20	22
0	1	2	3	4	5	6	7	8

x =17 > a[m] = a[4] = 11

cautare_binara(x=17, p=0, u=8)

p=0, u=8 => m=4

1	2	4	6	11	15	17	20	22
0	1	2	3	4	5	6	7	8

x =17 > a[m] = a[4] = 11

p=5, u=8 => m=6

15	17	20	22
5	6	7	8

cautare_binara(x=17, p=0, u=8)

p=0, u=8 => m=4

1	2	4	6	11	15	17	20	22
0	1	2	3	4	5	6	7	8

x =17 > a[m] = a[4] = 11

p=5, u=8 => m=6

15	17	20	22
5	6	7	8

x =17 = a[m] = a[6]

STOP – găsit pe poziția 6

return (True,6)

cautare_binara(x=16, p=0, u=8)

p=0, u=8 => m=4

1	2	4	6	11	15	17	20	22
0	1	2	3	4	5	6	7	8

x =16 > a[m] = a[4] = 11

p=5, u=8 => m=6

15	17	20	22
5	6	7	8

x =16 < a[6] = 17

cautare_binara(x=16, p=0, u=8)

p=0, u=8 => m=4

1	2	4	6	11	15	17	20	22
0	1	2	3	4	5	6	7	8

x =16 > a[m] = a[4] = 11

p=5, u=8 => m=6

15	17	20	22
5	6	7	8

x =16 < a[6] = 17

p=5, u=5 => m=5

15
5

x =16 > a[5] = 15

p=6, u=5 => nu este în a
return (False, 5)

Căutarea binară

```
def cautare_binara(x,ls,p,u):
    if p > u:
        return (False, u)
    else:
        mij = (p + u) // 2
        if x == ls[mij]:
            return (True,mij)
        elif x < ls[mij]:
            return cautare_binara(x,ls, p, mij-1)
        else:
            return cautare_binara(x,ls, mij+1, u)

def cautare(x,ls):
    n = len(ls)
    return cautare_binara(x,ls,0,n-1)
```

Căutarea binară

Implementare nerecursivă

```
def cautare_binara_nerecursiv(x,ls):
      p = 0
      u = len(ls) - 1
      while p <= u:
            mij = (p + u) // 2
            if x == ls[mij]:
                  return (True, mij)
            elif x < ls[mij]:
                  u = mij - 1
            else:
                  p = mij + 1
      return (False,u)
```

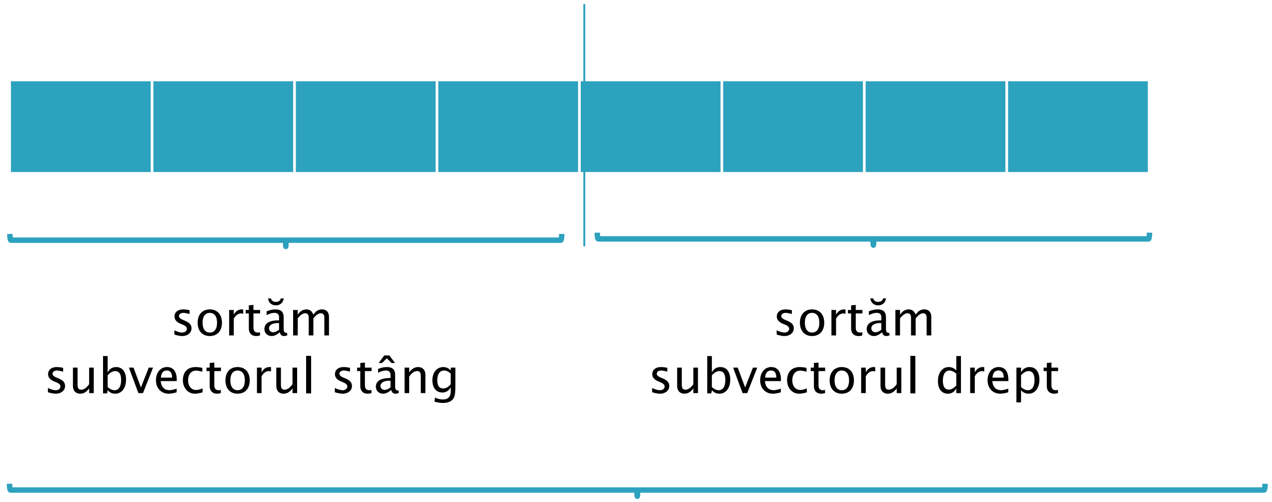
Căutarea binară

Complexitate $O(\log_2 n)$ – ca la power_DI

Sortarea prin interclasare
(Merge Sort)

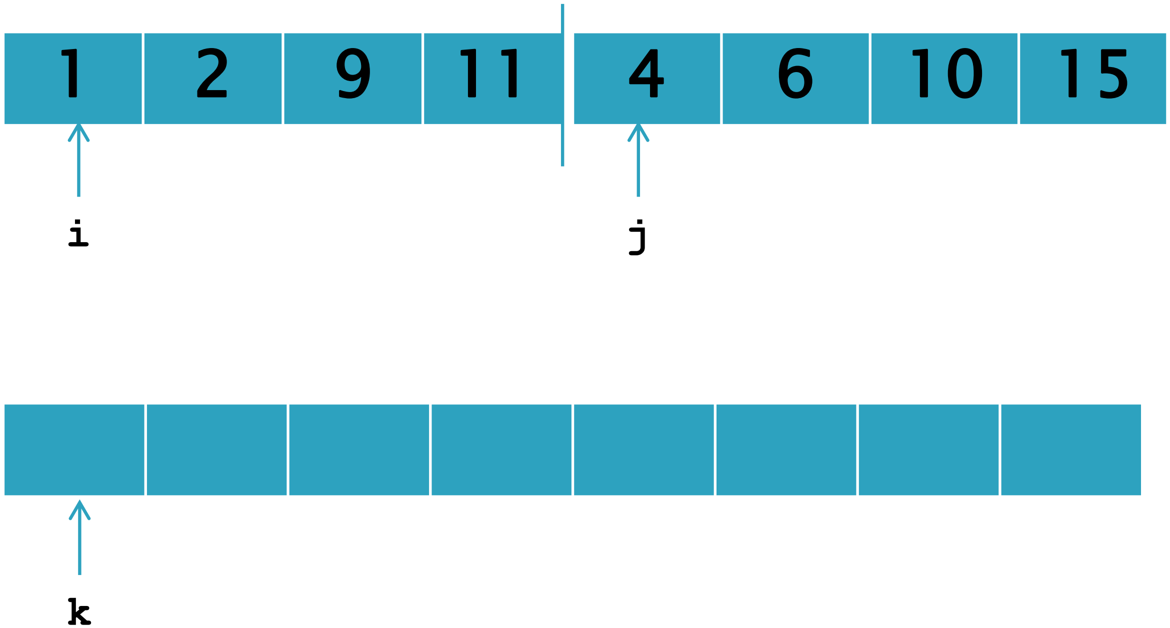
Sortare prin interclasare

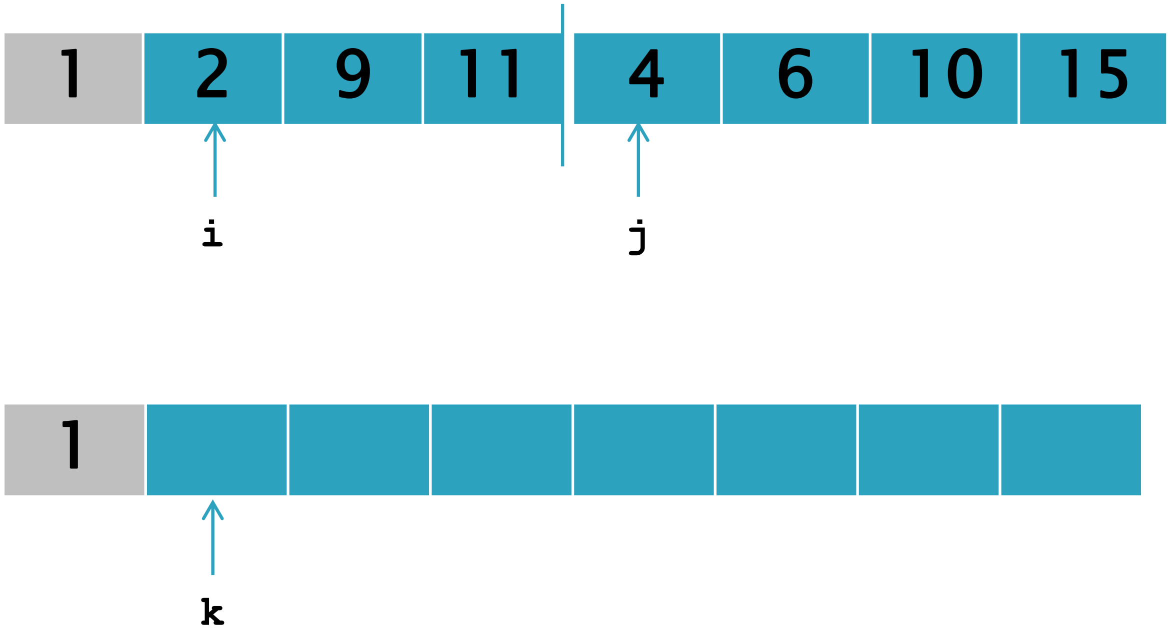
- **Idee:**
  - împărţim vectorul în doi subvectori
  - ordonăm crescător fiecare subvector
  - asamblăm rezultatele prin ***interclasare***

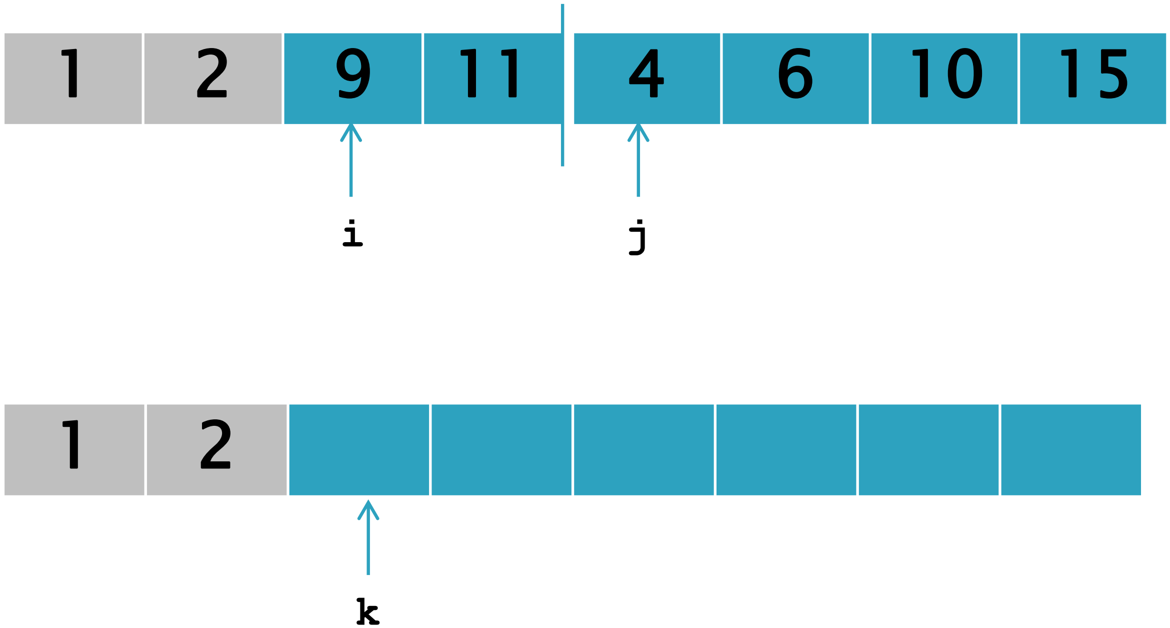


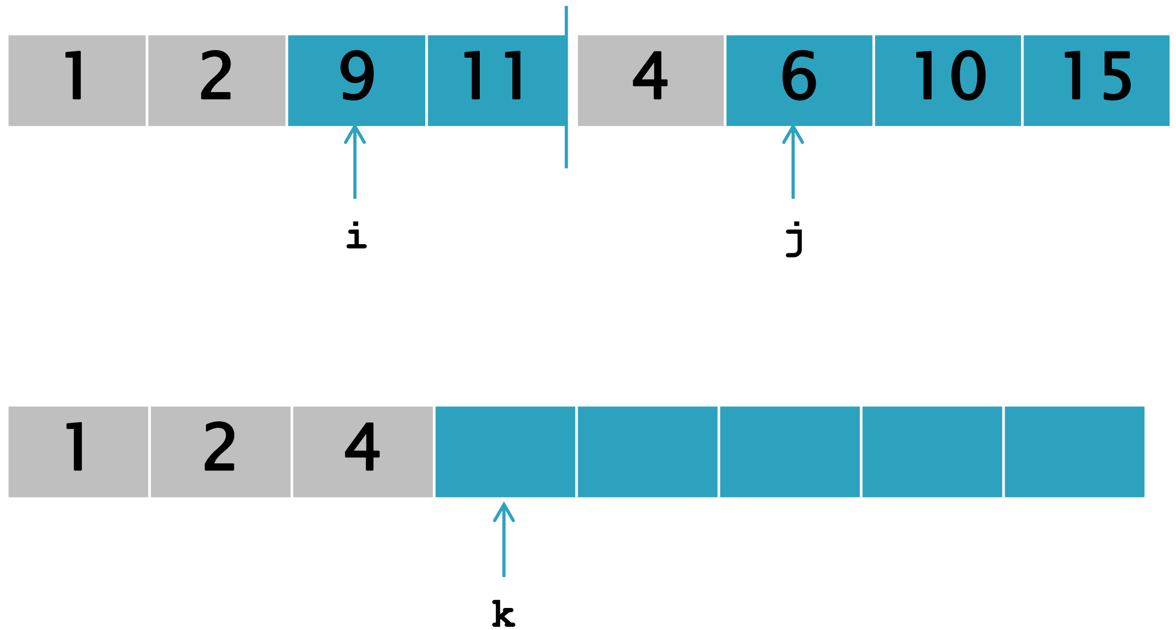
Interclasăm cei doi subvectori

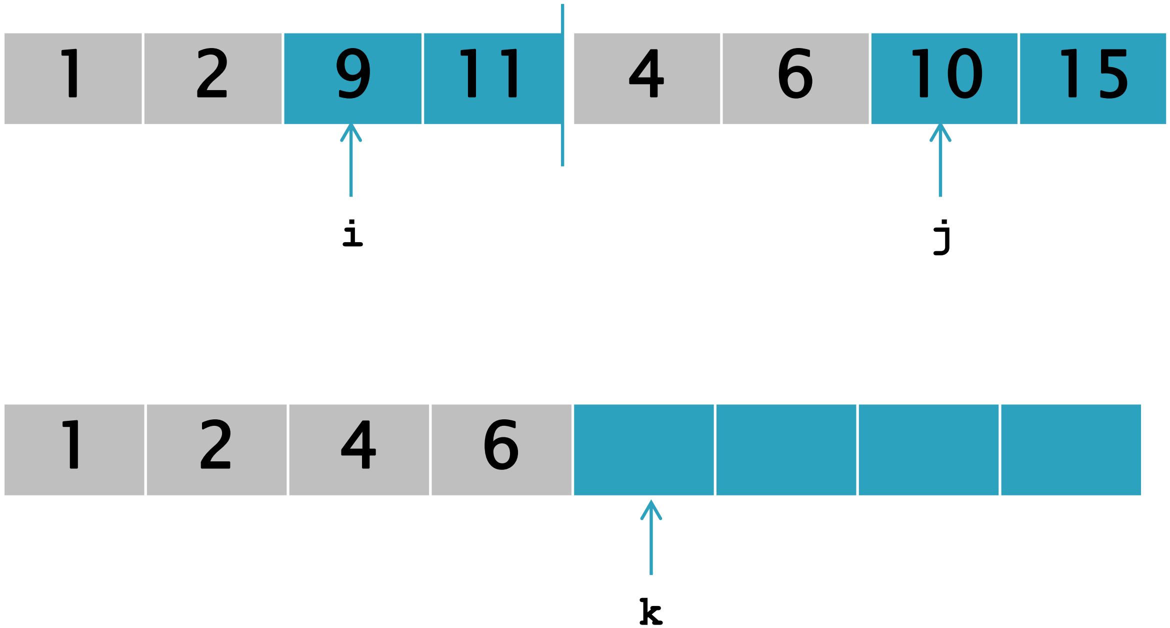
- "amestecăm" elementele din cei doi subvectori pentru a obține un vector ordonat
- parcurgem simultan cei doi subvectori

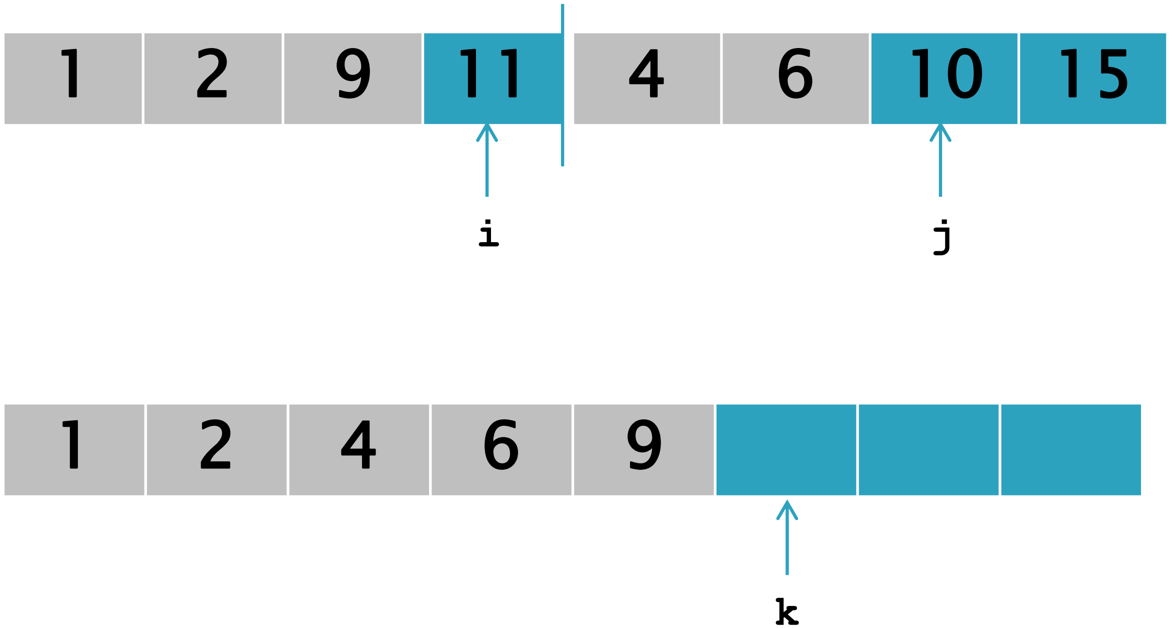

1
2
9
11
4
6
10
15
i
j
k

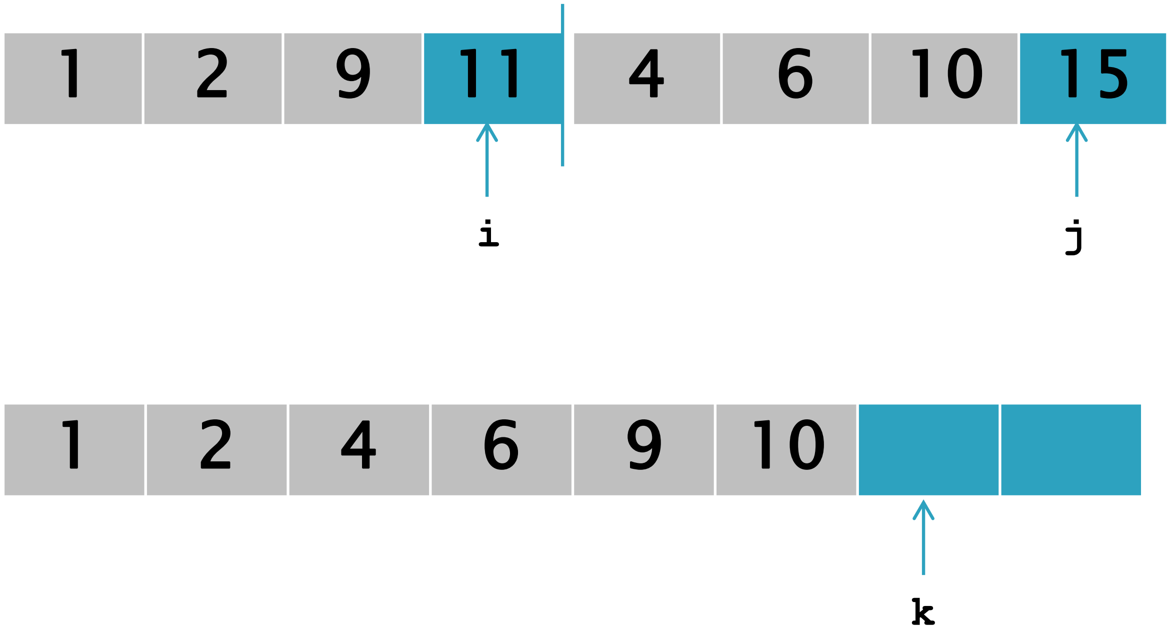

1
2
9
11
4
6
10
15
i
j
1
k

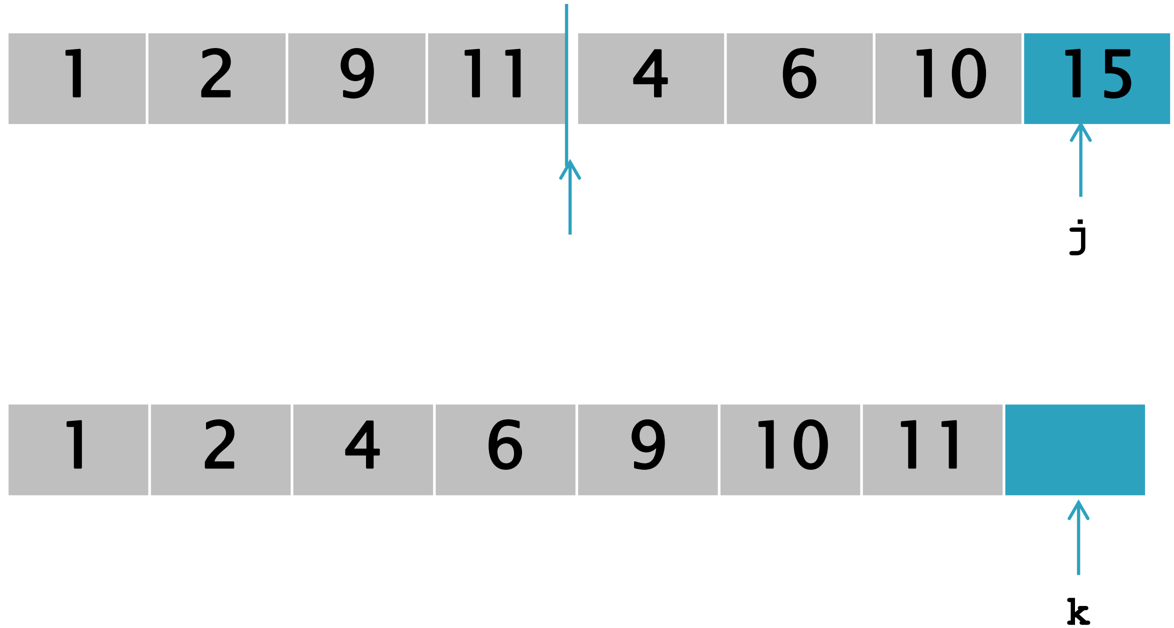

1
2
9
11
4
6
10
15
i
j
1
2
k


1
2
9
11
4
6
10
15
i
j
1
2
4
k


1
2
9
11
4
6
10
15
i
j
1
2
4
6
k


1 2 9 11 4 6 10 15
i
j
1 2 4 6 9
k


1 2 9 11 4 6 10 15
i
j
1 2 4 6 9 10
k


1 2 9 11 4 6 10 15
j
1 2 4 6 9 10 11
k

1	2	9	11	4	6	10	15

1	2	4	6	9	10	11	15

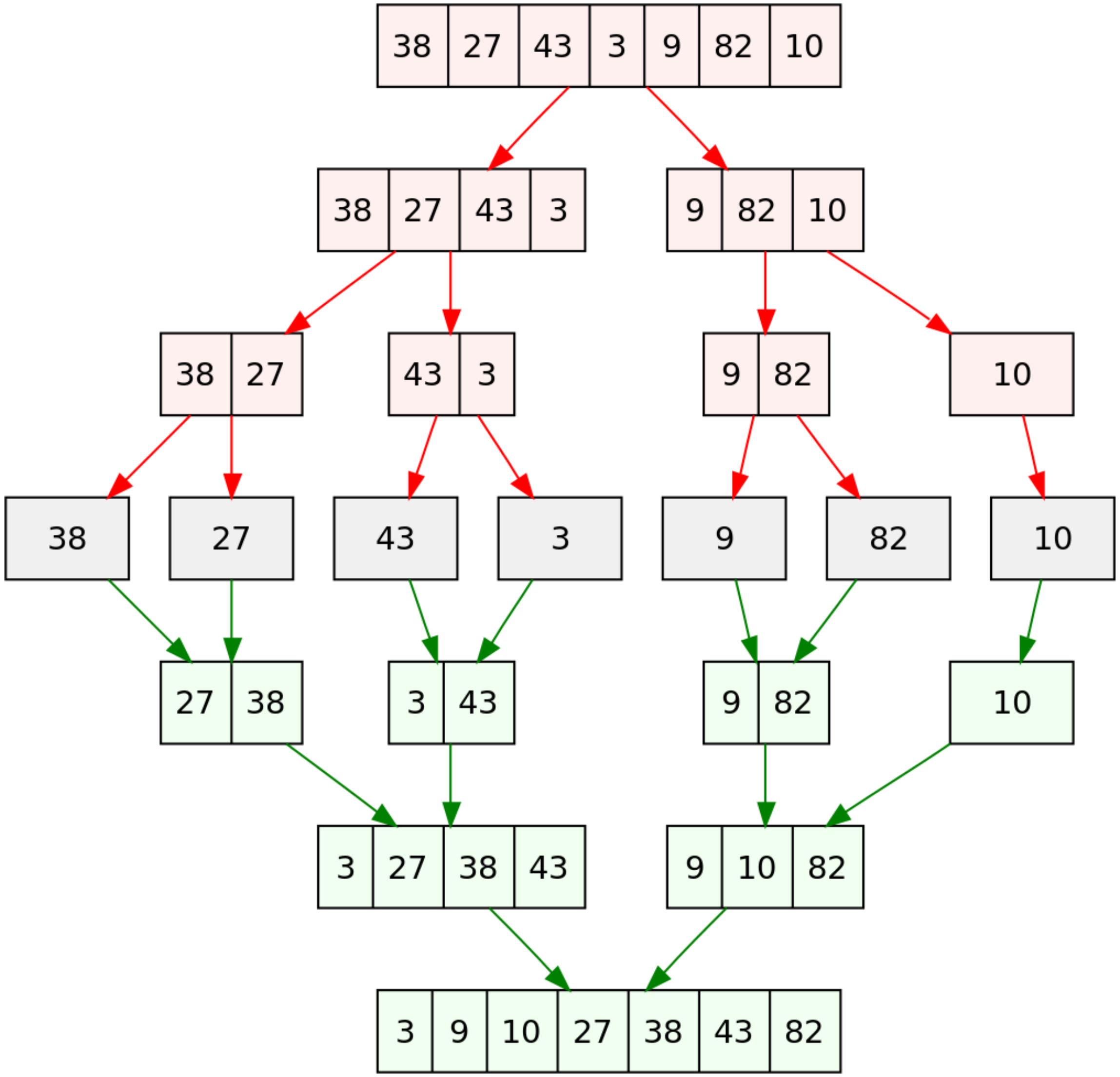
1	2	9	11	4	6	10	15

copiat

1	2	4	6	9	10	11	15

Sortare prin interclasare

```
def sort_interclasare(v, p, u):
    if p == u:
        pass
    else:
        m = (p+u)//2
        sort_interclasare(v, p, m)
        sort_interclasare(v, m+1, u)
        interclaseaza(v, p, m, u)
```



https://en.wikipedia.org/wiki/Merge_algorithm

```
def interclaseaza(a, p, m, u):
   b = [None]*(u-p+1)
   i = p
   j = m + 1
   k = 0
   while (i <= m) and (j <= u):
         if a[i] <= a[j]:
            b[k] = a[i]; i += 1
         else:
            b[k] = a[j]; j+= 1
         k+=1
```

```python
def interclaseaza(a, p, m, u):
   b = [None]*(u-p+1)
   i = p
   j = m + 1
   k = 0
   while (i <= m) and (j <= u):
         if a[i] <= a[j]:
            b[k] = a[i]; i += 1
         else:
            b[k] = a[j]; j+= 1
         k+=1

   while i <= m:
      b[k] = a[i]; k += 1; i += 1

   while j <= u:
      b[k] = a[j]; k += 1; j += 1
```

```
def interclaseaza(a, p, m, u):
   b = [None]*(u-p+1)
   i = p
   j = m + 1
   k = 0
   while (i <= m) and (j <= u):
         if a[i] <= a[j]:
            b[k] = a[i]; i += 1
         else:
            b[k] = a[j]; j+= 1
         k+=1

   while i <= m:
      b[k] = a[i]; k += 1; i += 1

   while j <= u:
      b[k] = a[j]; k += 1; j += 1

   for i in range(p,u+1):
      a[i] = b[i-p]
```

Sortare prin interclasare

➢**Complexitate:**

```
def sort_interclasare(v, p, u):
    if p == u:
        pass
    else:
        m = (p+u)//2
        sort_interclasare(v, p, m)
        sort_interclasare(v, m+1, u)
        interclaseaza(v, p, m, u)
```

Problema de dimensiune n

Timp constant O(1)

Două subprobleme de dimensiune n/2

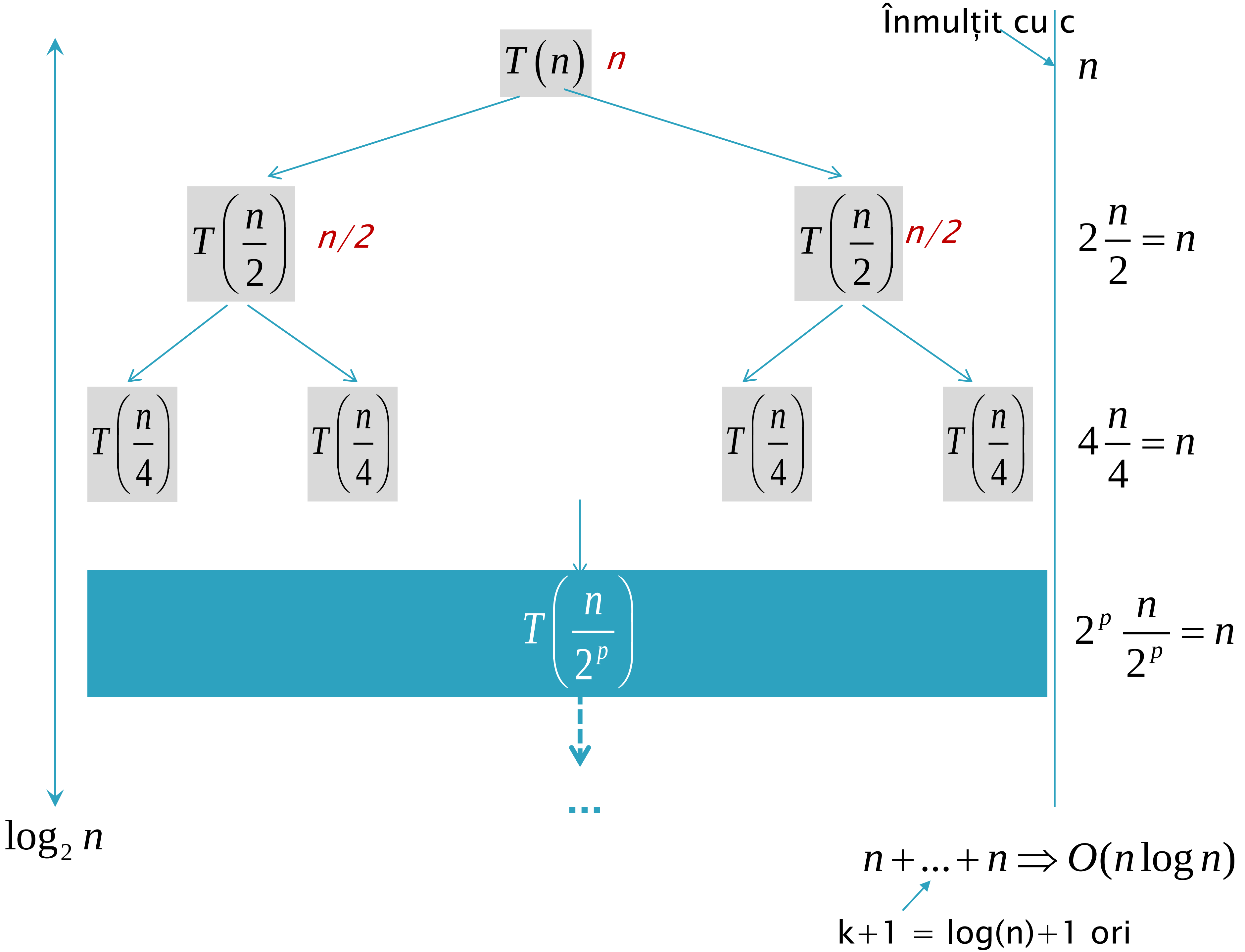
O(n) (interclaseaza cele doua jumatati, adica doi vectori de dimensiune n/2)

$\Rightarrow T(n) = 2T(n/2) + O(n)$

$T(n) = 2T(n/2) + n$

Sortare prin interclasare

- **Complexitate: $O(n \log_2(n))$**

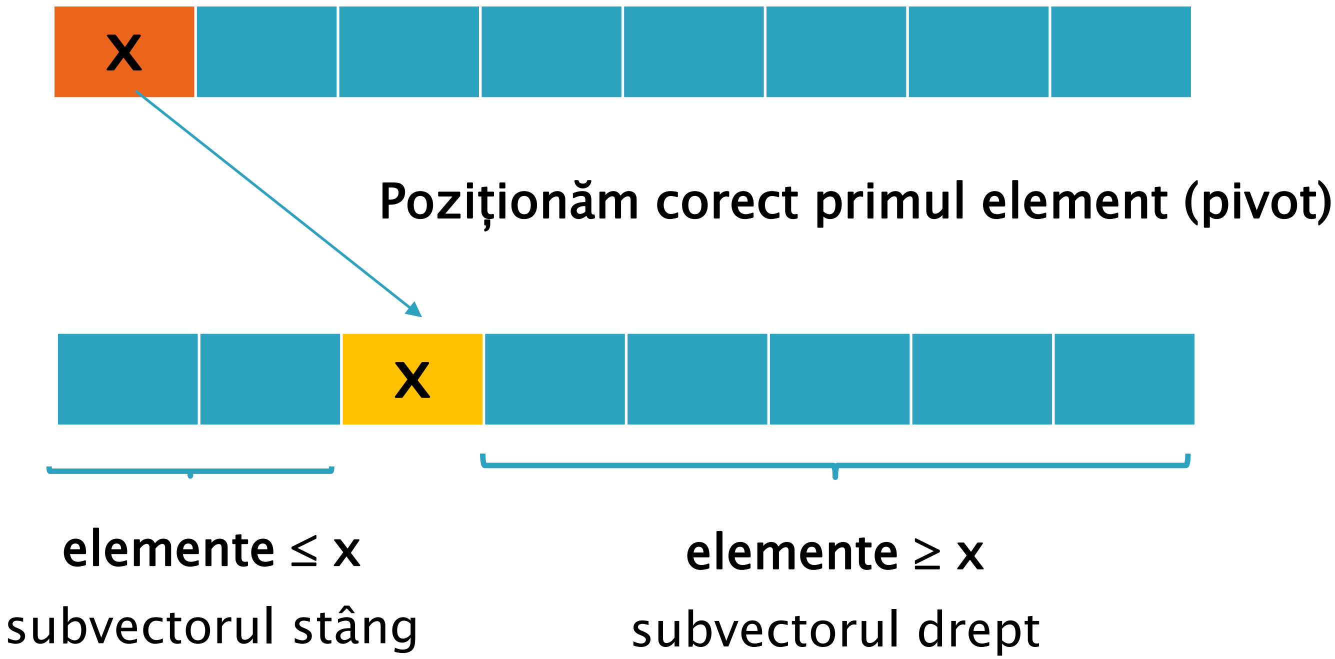


Înmulțit cu c
$T(n)$
n
n
$T\left(\frac{n}{2}\right)$
$n/2$
$T\left(\frac{n}{2}\right)$
$n/2$
$2\frac{n}{2} = n$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$4\frac{n}{4} = n$
$T\left(\frac{n}{2^p}\right)$
$2^p\frac{n}{2^p} = n$
...
$\log_2 n$
$n + \ldots + n \Rightarrow O(n \log n)$
k+1 = log(n)+1 ori

Quicksort
Sortarea rapidă

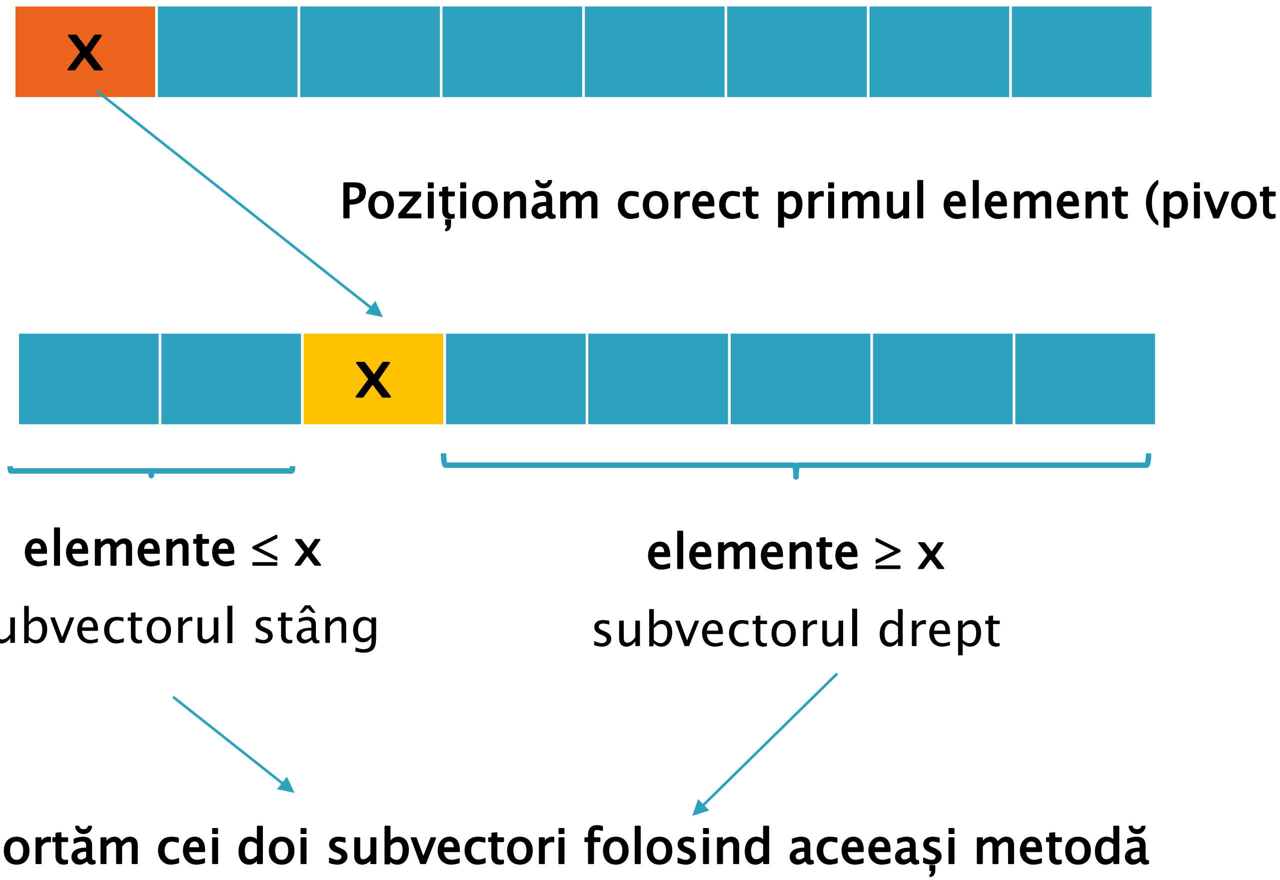
Quicksort

- **Idee:**



Quicksort

- **Idee:**

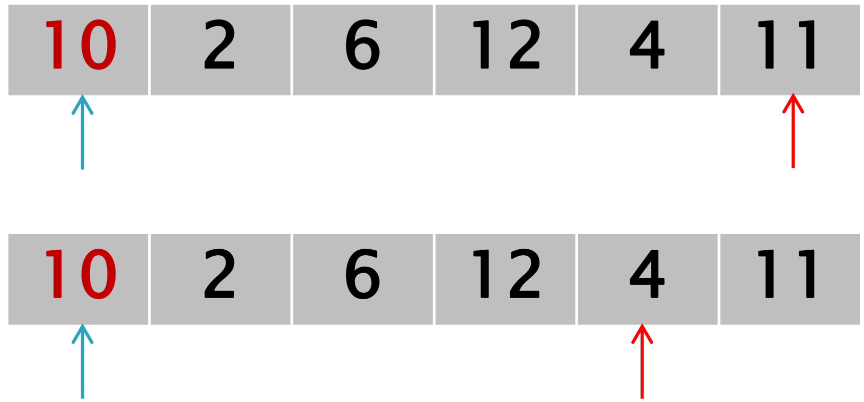


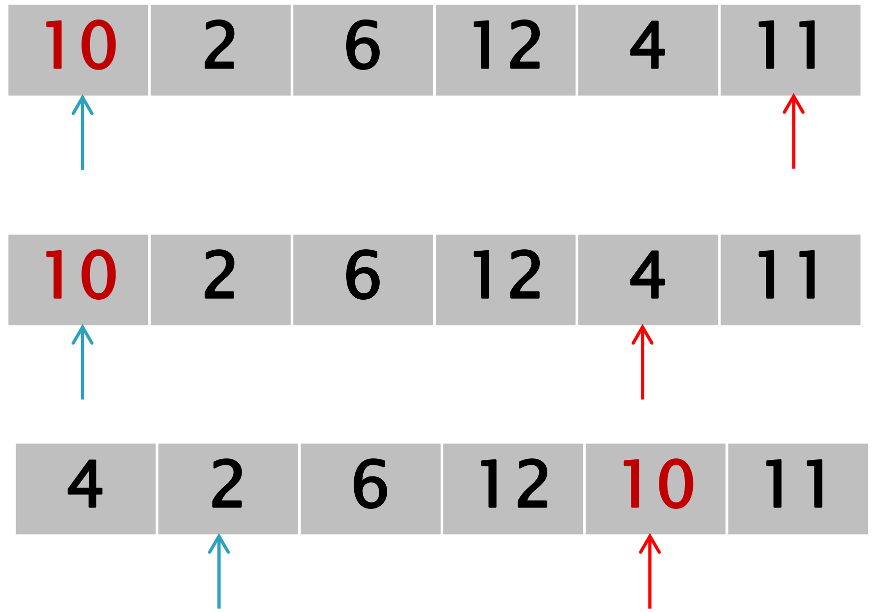
Quicksort

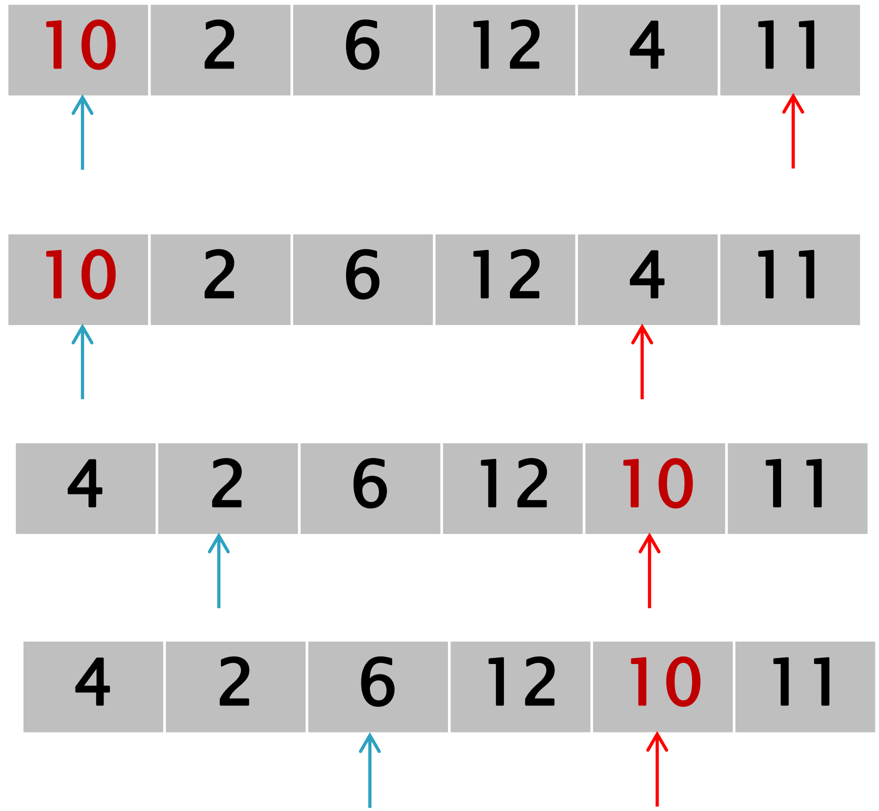
- **Idee:**
  - poziționăm primul element al secvenței (**pivotul**) pe poziția sa finală = astfel încât elementele din stânga sa sunt mai mici, iar cele din dreapta mai mari
  - ordonăm crescător elementele din stânga
  - ordonăm crescător elementele din dreapta

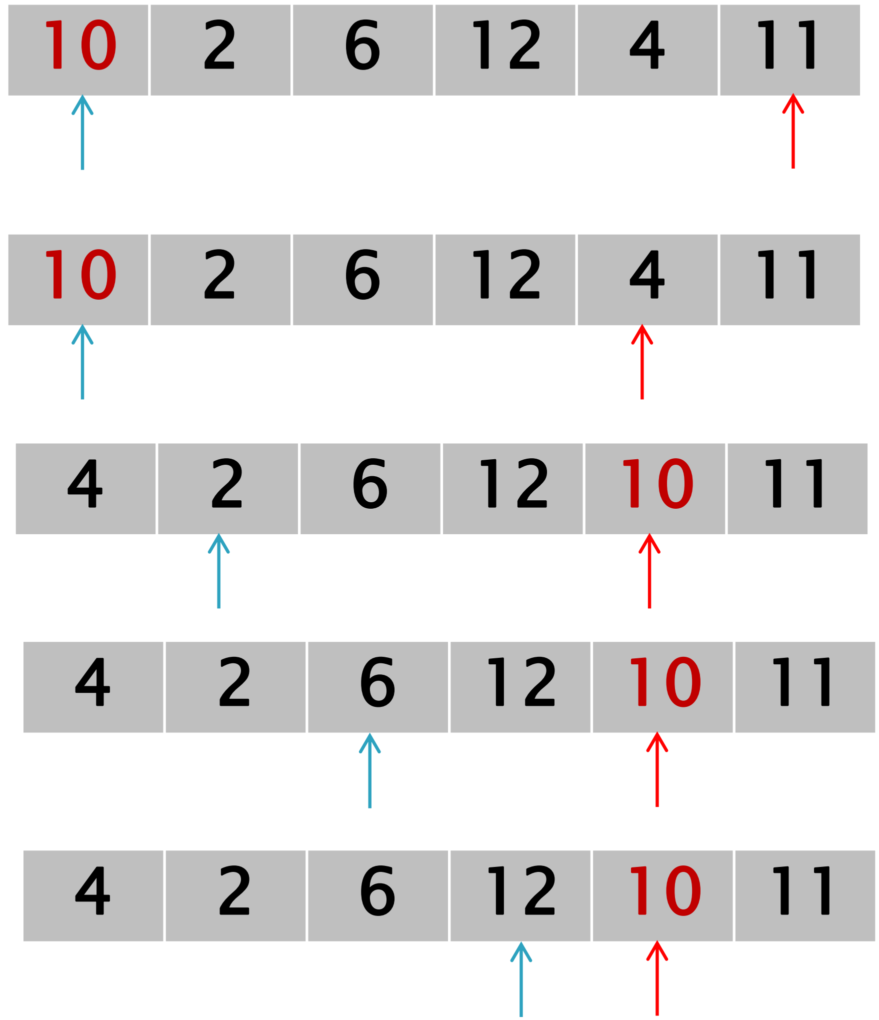
Exemplu – poziţionare pivot

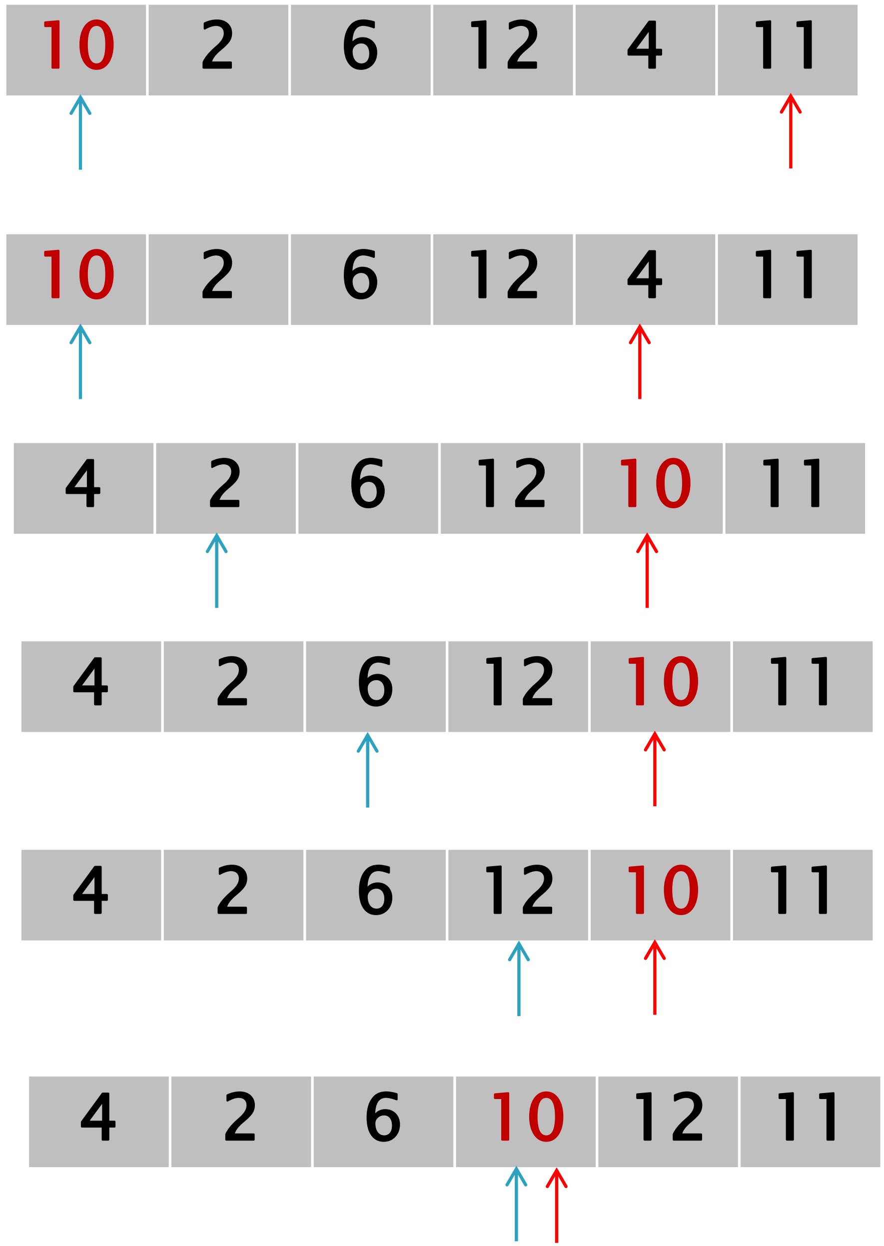
10	2	6	12	4	11


10 2 6 12 4 11
10 2 6 12 4 11


10 2 6 12 4 11
10 2 6 12 4 11
4 2 6 12 10 11


10 2 6 12 4 11
10 2 6 12 4 11
4 2 6 12 10 11
4 2 6 12 10 11


10 2 6 12 4 11
10 2 6 12 4 11
4 2 6 12 10 11
4 2 6 12 10 11
4 2 6 12 10 11


10 2 6 12 4 11
10 2 6 12 4 11
4 2 6 12 10 11
4 2 6 12 10 11
4 2 6 12 10 11
4 2 6 10 12 11

Quicksort

```
def quick_sort_di(v, p, u):
    if p >= u:
        return
    m = poz(v, p, u)
    quick_sort_di(v, p, m - 1)
    quick_sort_di(v, m + 1, u)
```

```python
def poz(v, p, u):
    i = p
    j = u
    depli = 0
    deplj = -1
    while i < j:
        if v[i] > v[j]:
            v[i], v[j] = v[j], v[i]
            depli, deplj = -deplj, -depli
            #aux= depli; depli= -deplj; deplj= -aux;
        i += depli
        j += deplj
    return i
```

Quicksort

- **Complexitate:**
  - **Defavorabil**: $O(n^2)$

    pentru vector deja sortat =>

    - una dintre subprobleme are dimensiune n-1
    - pivotarea n-1

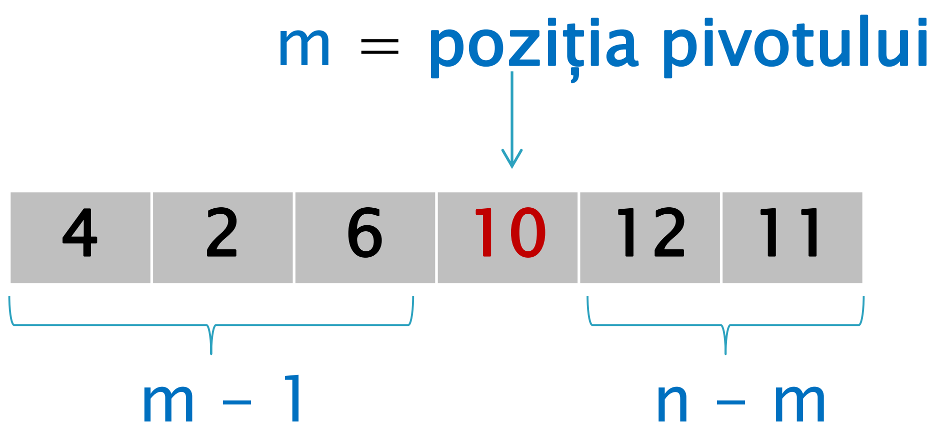
n-1 + n-2+ .... + 1 => $O(n^2)$

m = 1 = **poziția pivotului**

$$\overset{\downarrow}{1}\quad \underbrace{2\quad 6\quad 10\quad 12\quad 14}_{n\,-\,1}$$

Quicksort

- **Complexitate:**
  - **Mediu**: O(n log n) – cu **pivot ales aleator**
    - Alegem aleator un element ca pivot, îl interschimbăm cu primul element și folosim procedura de pivotare anterioară



Quicksort – pivot aleator

```
def poz_rand(v, p, u):
    r = random.randint(p, u) – poziție pivot aleasa random
    v[r], v[p] = v[p], v[r] – mut pivot pe pozitia p (la inceput)
    return poz(v, p, u) – pentru pozitionare pivot folosim
                                procedura anterioara
```

Quicksort – pivot aleator

```
def quick_sort_di(v, p, u):
    if p >= u:
        return
    m = poz_rand(v, p, u)
    quick_sort_di(v, p, m - 1)
    quick_sort_di(v, m + 1, u)
```

Aplicații

Statistici de ordine

Statistici de ordine

Dat un vector a de n numere şi un indice k, $1 \leq k \leq n$, să se determine al k-lea cel mai mic element din vector.

Statistici de ordine

A i-a statistică de ordine a unei mulțimi = al i-lea cel mai mic element.

- **Minimul** = prima statistică de ordine
- **Maximul** = a n-a statistică de ordine

Statistici de ordine

- **Mediana** = punctul de la jumătatea unei mulțimi

  = o valoare v a.î. numărul de elemente din mulțime mai mici decât v este egal cu numărul de elemente din mulțime mai mari decât v.

Statistici de ordine

- **Mediana**

Dacă n este impar, atunci mediana este

a $\lceil n/2 \rceil$–a statistică de ordine, altfel, prin **convenție** mediana este **media aritmetică** dintre a

$\lfloor n/2 \rfloor$–a statistică și a $(\lfloor n/2 \rfloor + 1)$–a statistică de ordine

1	12	15	16	38	40

15,5

- **Mediană inferioară / superioară**

Statistici de ordine – utilitate

- **Statistică**
- **Mediana** – pentru o mulţime A={a_1,…,a_n} valoarea μ care minimizează expresia

$$\sum_{i=1}^{n} \left| \mu - a_i \right|$$

Statistici de ordine

Idee Al k-lea minim

Statistici de ordine

Idee **Al k-lea minim** – folosim poziţionarea de la quicksort (**pivot aleator**)

Statistici de ordine

Idee Al k-lea minim – folosim poziţionarea de la quicksort (**pivot aleator**)

Fie m poziţia pivotului

- Dacă m = k
- Dacă **m** > **k**
- Dacă **m** < **k**

Statistici de ordine

Idee **Al k-lea minim** – folosim poziţionarea de la quicksort (**pivot aleator**)

Fie m poziţia pivotului

- Dacă m = k, pivotul este al k-lea minim
- Dacă **m > k**
- Dacă **m < k**

Statistici de ordine

Idee **Al k-lea minim** – folosim poziţionarea de la quicksort (**pivot aleator**)

Fie m poziţia pivotului

- Dacă m = k, pivotul este al k-lea minim
- Dacă **m > k**, al k-lea minim este în **stânga** pivotului (al k-lea minim din stanga)
- Dacă **m < k**, al k-lea minim este în **dreapta** pivotului (al (k-m)-lea minim din dreapta)

k = 2

10	2	6	12	4	11

poziţionare pivot

4	2	6	10	12	11

$m = 4 > k \Longrightarrow$ stânga

k = 2

10	2	6	12	4	11

poziţionare pivot

4	2	6	10	12	11

$m = 4 > k \longrightarrow$ stânga

4	2	6

poziţionare pivot

2	4	6

$m = 2 = k \longrightarrow$ stop;

pivotul este al k-lea minim

```python
#pentru numerotare de la 0

def sel_k_min(v, k, p, u):
    m = poz_rand(v, p, u)
    if m == k - 1:
        return v[m]
    if m < k - 1:
        return sel_k_min(v, k, m + 1, u)
    return sel_k_min(v, k, p, m - 1)
```

Apel: `x = sel_k_min(v, k, 0, len(v) - 1)`

```
#pentru numerotare de la 0

def sel_k_min(v, k, p, u):
    m = poz_rand(v, p, u)
    if m == k - 1:
        return v[m]
    if m < k - 1:
        return sel_k_min(v, k, m + 1, u)
    return sel_k_min(v, k, p, m - 1)
```

Apel: `x = sel_k_min(v, k, 0, len(v) - 1)`

Complexitate medie O(n)

Numărarea inversiunilor

Numărare inversiuni

Problemă

Se consideră un vector cu n elemente distincte.

Să de determine numărul de inversiuni din acest vector

- **Inversiune** = pereche (i, j) cu proprietatea că $i < j$ şi $a_i > a_j$

- **Exemplu** **`1,2,11,9,4,6`** $\Rightarrow$ **`5 inversiuni`**

  `((11,9), (11,4), (9,4), (11,6), (9,6))`

Aplicații

- **Măsură a diferenței între două liste ordonate**
- **“Gradul de ordonare” al unui vector**
- **Probleme de analiză a clasificărilor (ranking)**
  - Asemănarea între preferințele a doi utilizatori – sugestii de utilizatori cu preferințe similare
  - Asemănări dintre rezultatele întoarse de motoare diferite de căutare pentru aceeași cerere
  - **collaborative filtering**

Aplicații

- Suficient să presupunem că prima clasificare este

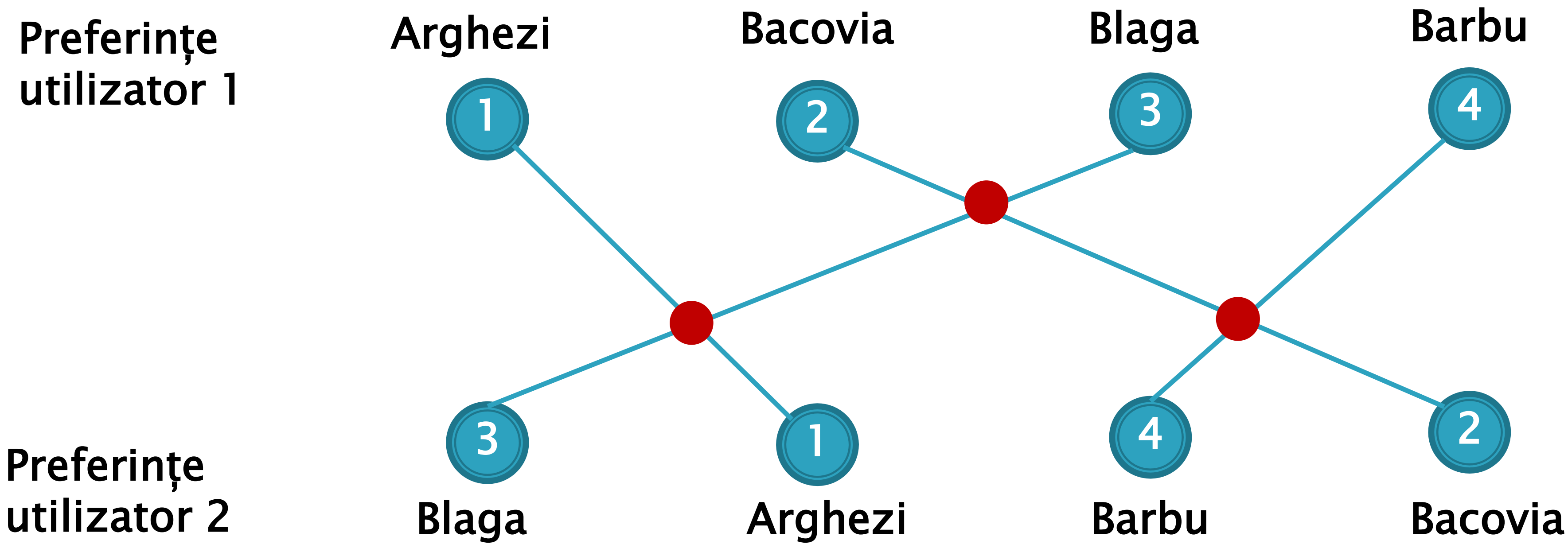
  `1, 2, 3, …, n`

- Gradul de asemănare dintre clasificări = numărul de inversiuni din a doua clasificare

Aplicații

Preferințe utilizator 1	Arghezi	Bacovia	Blaga	Barbu
	1	2	3	4
Preferințe utilizator 2	3	1	4	2
	Blaga	Arghezi	Barbu	Bacovia

Aplicații



3 inversiuni

Numărare inversiuni

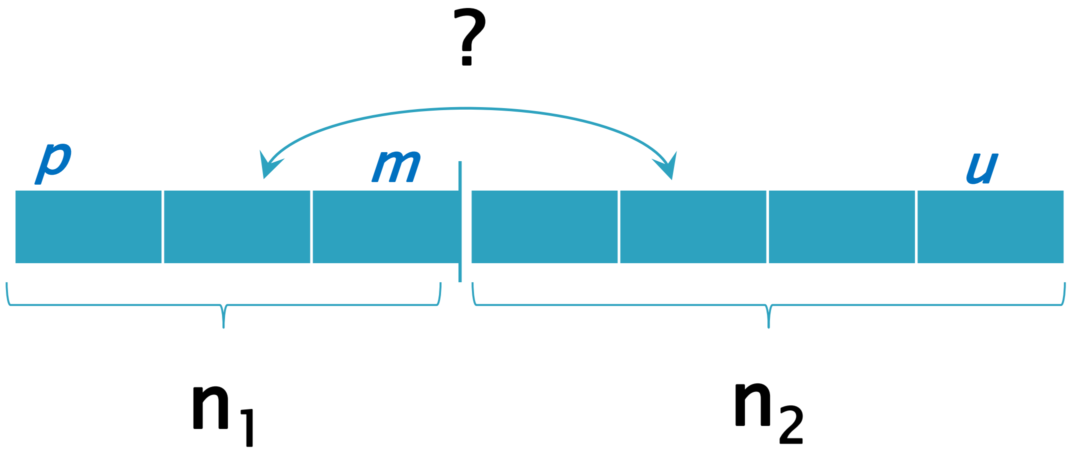
Numărul maxim de inversiuni pentru un vector cu n elemente?

Numărare inversiuni

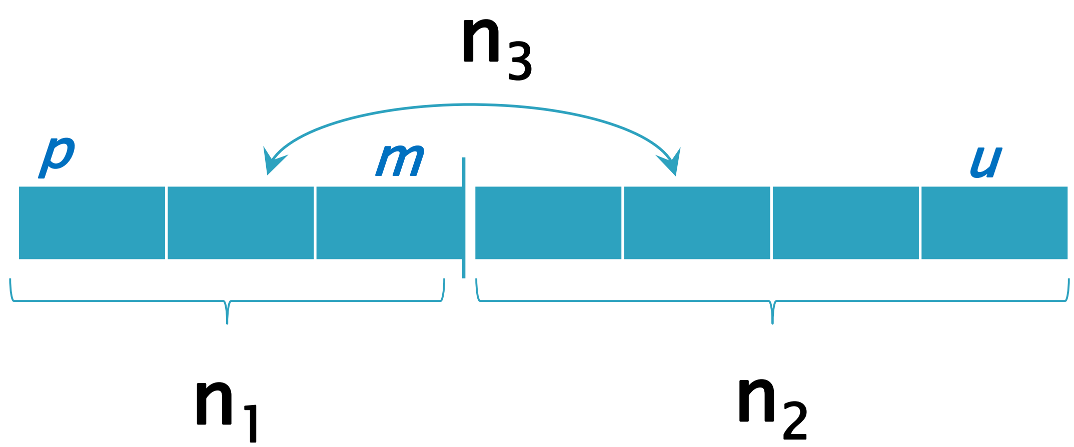
- **Algoritm $\Theta(n^2)$** – evident

Numărare inversiuni

- Algoritm Divide et Impera

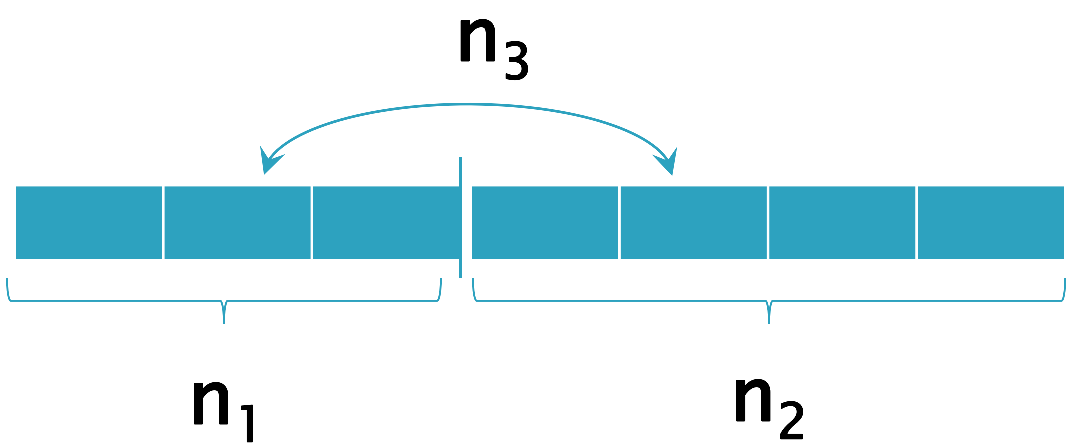


$$n_1+n_2+?$$

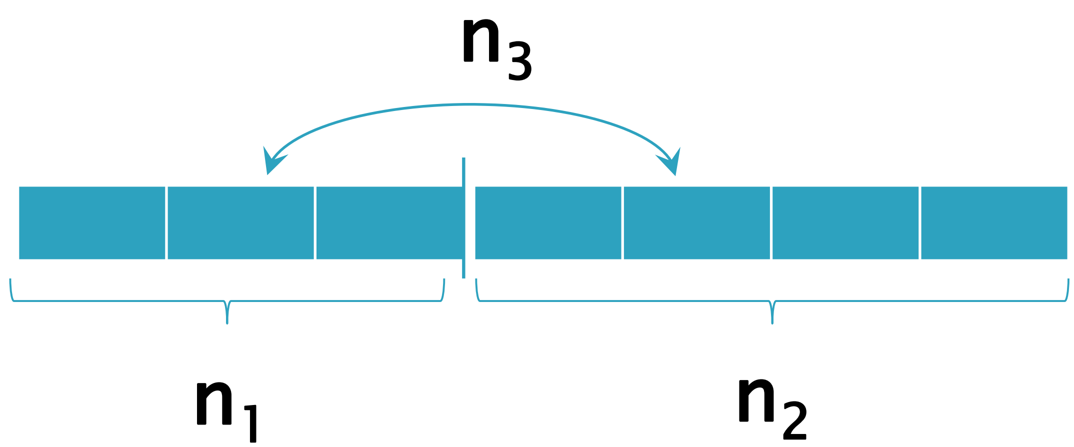


$$n_1+n_2+n_3$$

Cum calculăm eficient n_3?

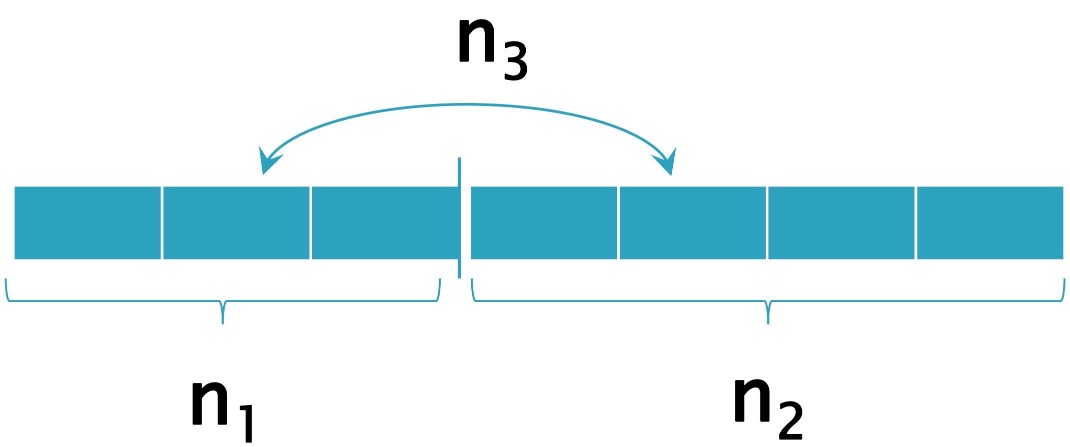


Cum calculăm eficient n_3?



Cum calculăm eficient n_3?

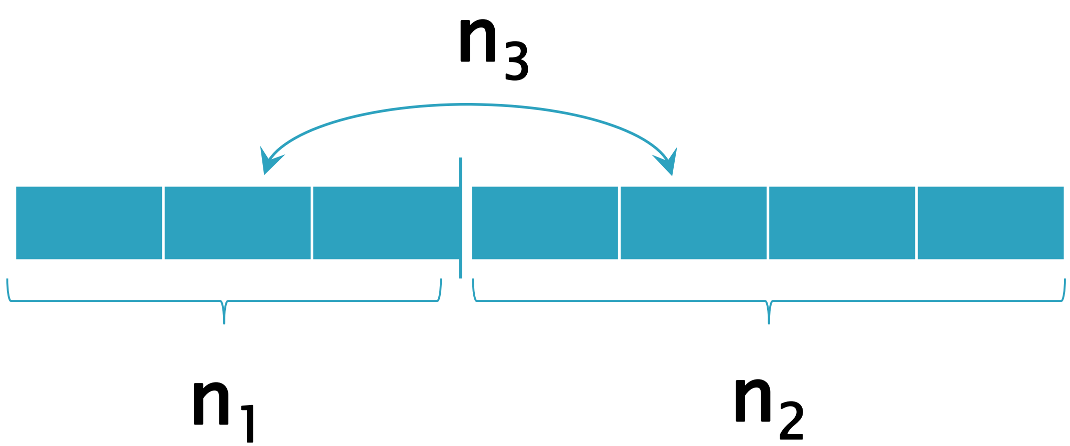
- **Încercare**: Considerăm fiecare pereche (i,j) cu i în subvectorul stâng și j în cel drept



Cum calculăm eficient n_3?

- **Încercare**: Considerăm fiecare pereche (i,j) cu i în subvectorul stâng și j în cel drept

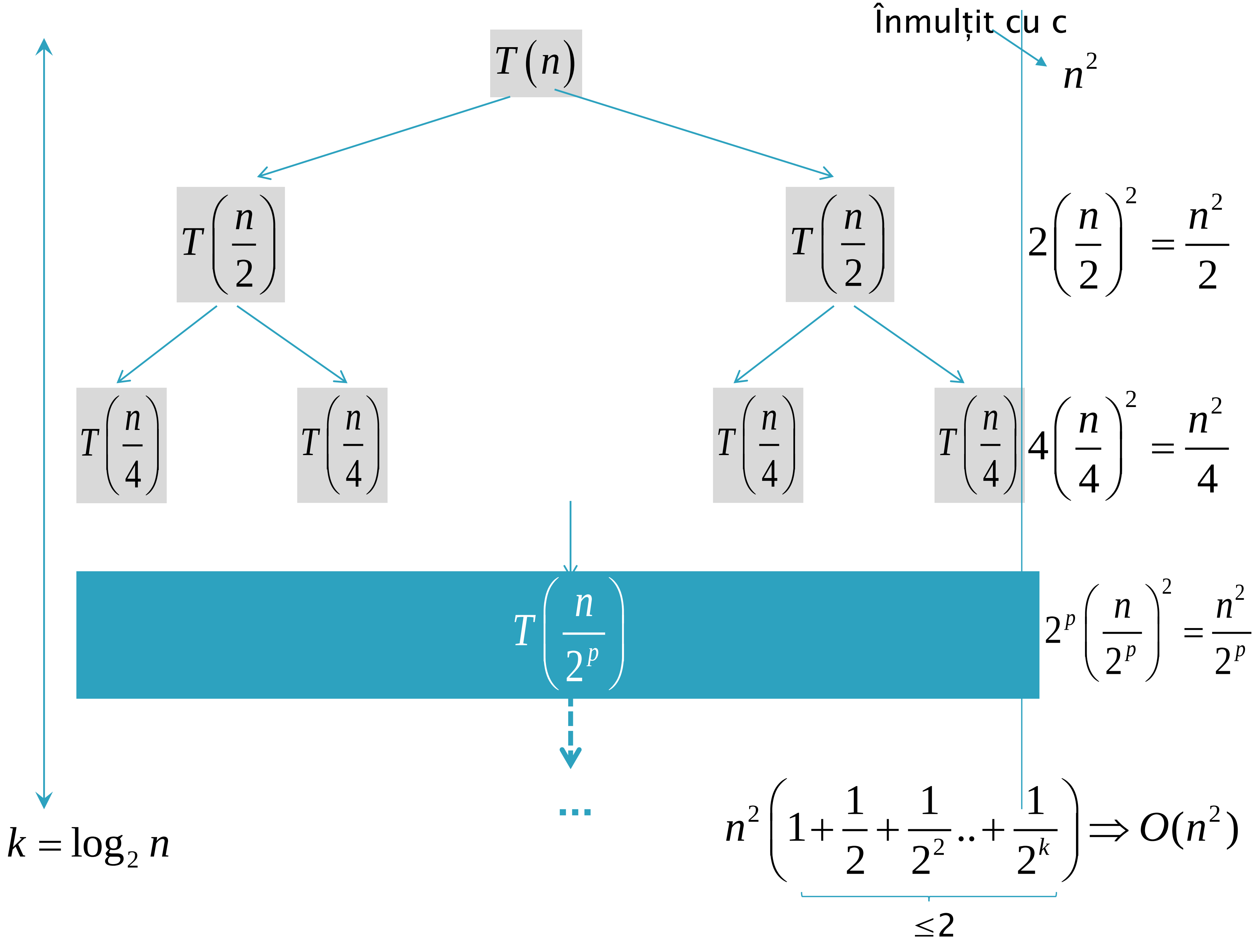
$$T(n) = 2T(n/2)+cn^2$$



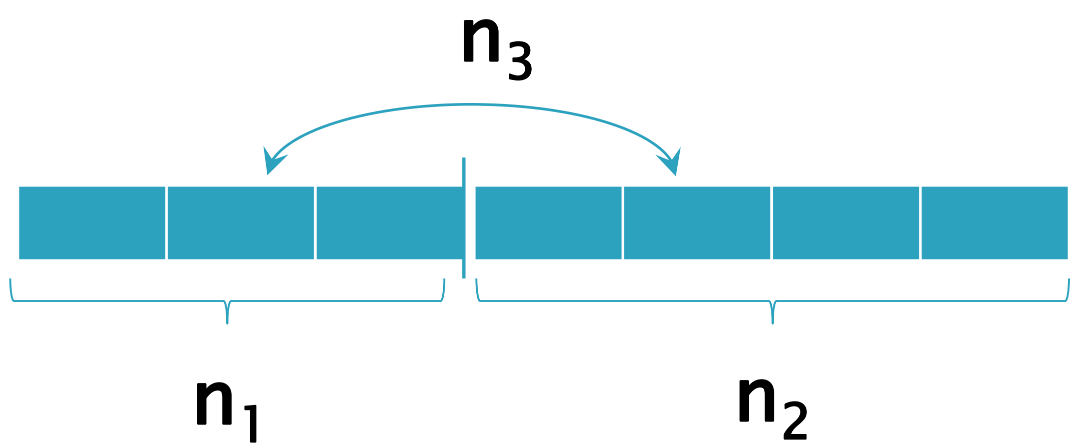
Cum calculăm eficient n_3?

- **Încercare**: Considerăm fiecare pereche (i,j) cu i în subvectorul stâng și j în cel drept

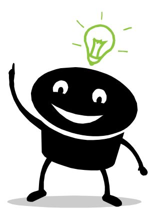
$$T(n) = 2T(n/2)+cn^2 \Rightarrow O(n^2)$$



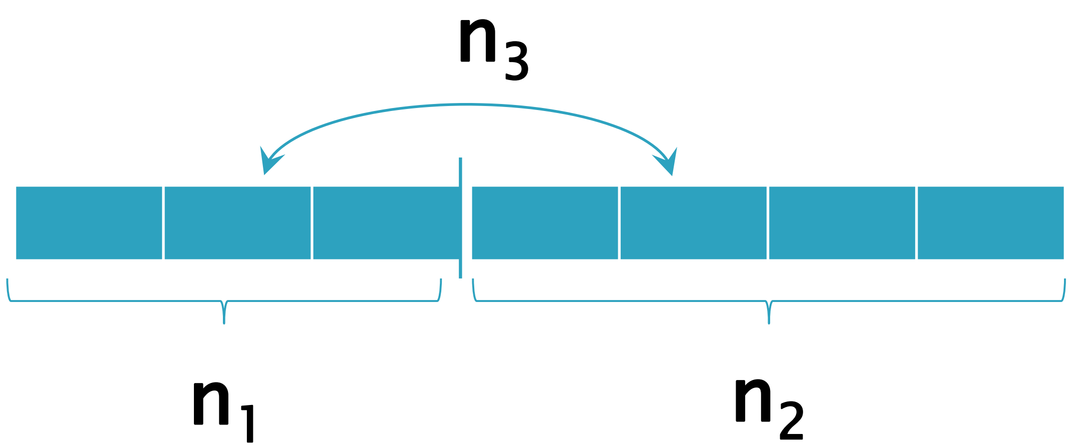
Înmulțit cu c
$T(n)$
n^2
$T\left(\frac{n}{2}\right)$
$T\left(\frac{n}{2}\right)$
$2\left(\frac{n}{2}\right)^2 = \frac{n^2}{2}$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$T\left(\frac{n}{4}\right)$
$4\left(\frac{n}{4}\right)^2 = \frac{n^2}{4}$
$T\left(\frac{n}{2^p}\right)$
$2^p\left(\frac{n}{2^p}\right)^2 = \frac{n^2}{2^p}$
...
$k = \log_2 n$
$n^2\left(\underbrace{1+\frac{1}{2}+\frac{1}{2^2}..+\frac{1}{2^k}}_{\leq 2}\right) \Rightarrow O(n^2)$



Cum calculăm eficient n_3?



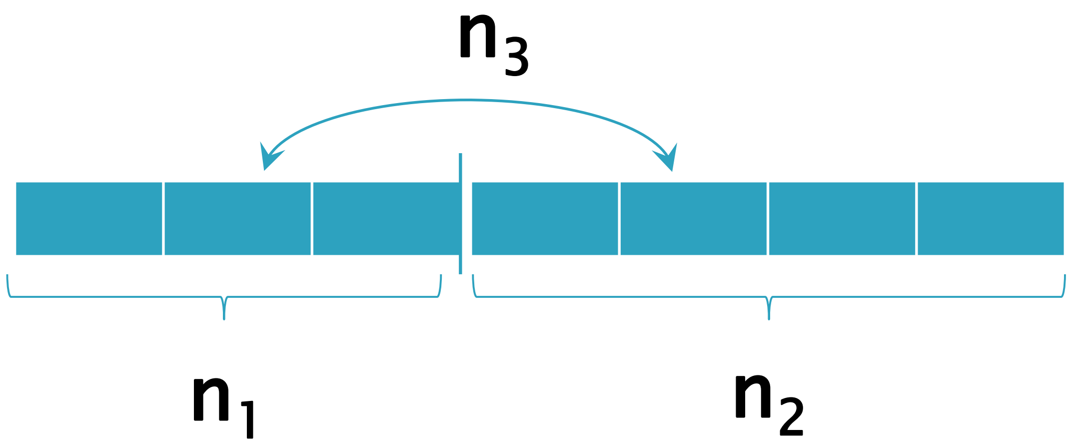
Dacă subvectorii stâng şi drept sunt **sortaţi crescător**, numărarea inversiunilor (i,j) date de elemente din subvectori diferiţi se poate face la interclasare



Cum calculăm eficient n_3?

Dacă subvectorii stâng şi drept sunt **sortaţi crescător**, numărarea inversiunilor (i,j) date de elemente din subvectori diferiţi se poate face la interclasare

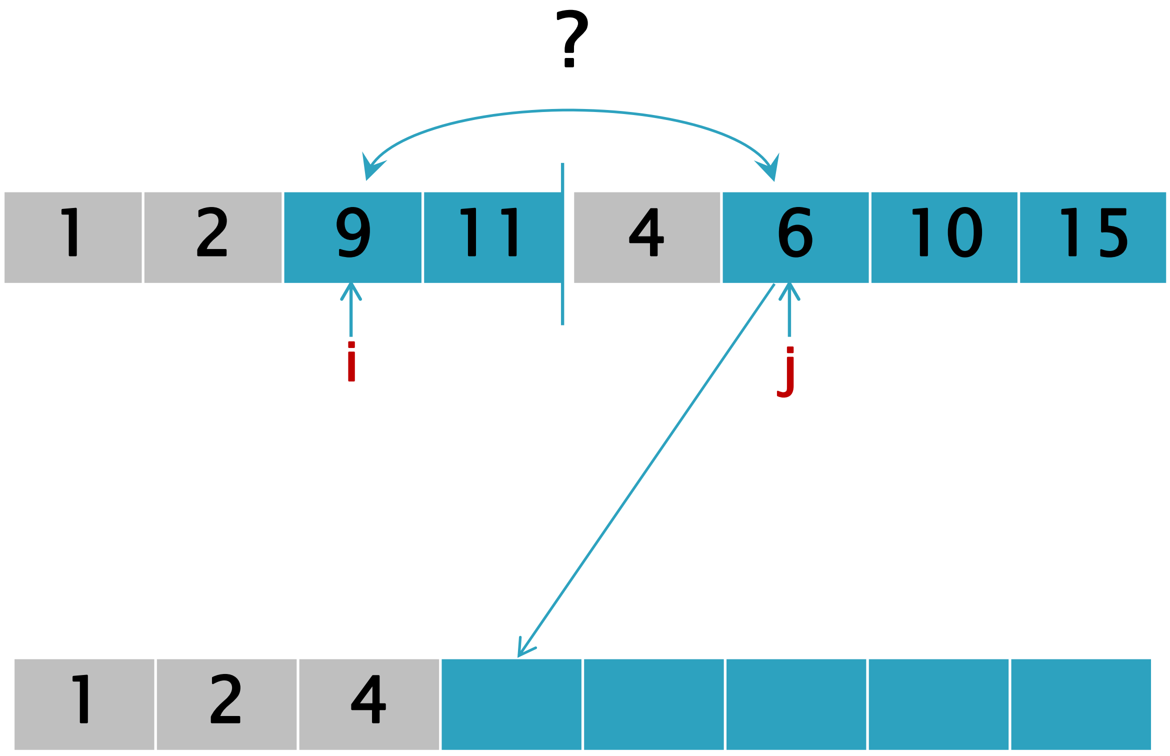
$T(n) = 2T(n/2)+cn \Rightarrow$

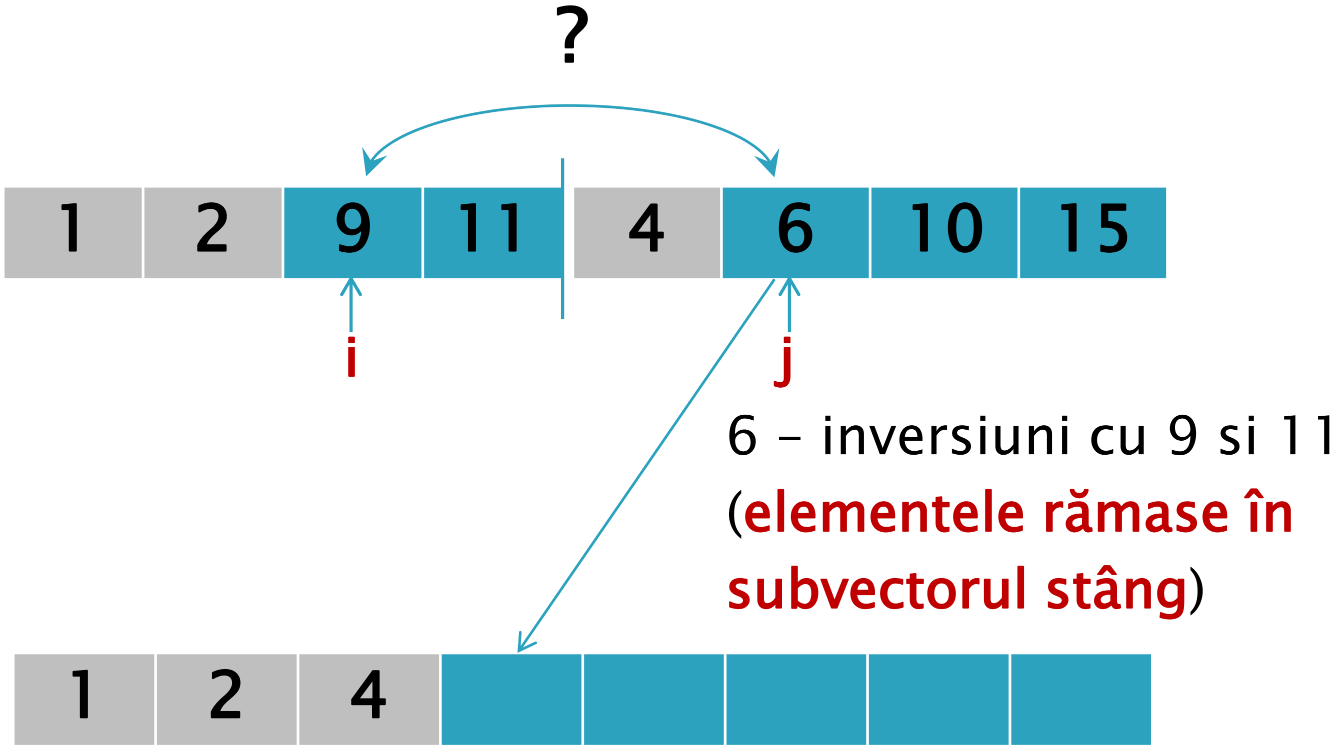


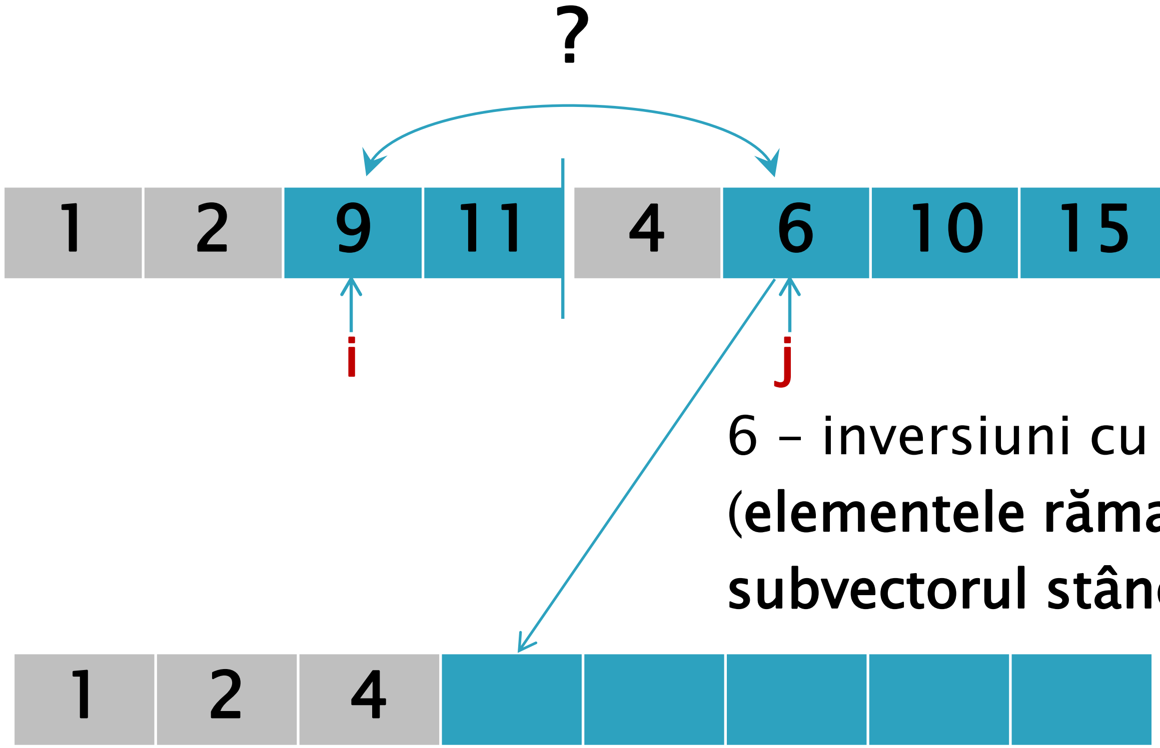
Cum calculăm eficient n_3?

Dacă subvectorii stâng şi drept sunt **sortaţi crescător**, numărarea inversiunilor (i,j) date de elemente din subvectori diferiţi se poate face la interclasare

$$T(n) = 2T(n/2)+cn \ \Rightarrow \ T(n) = O(n \log n)$$


?
1
2
9
11
4
6
10
15
i
j
1
2
4

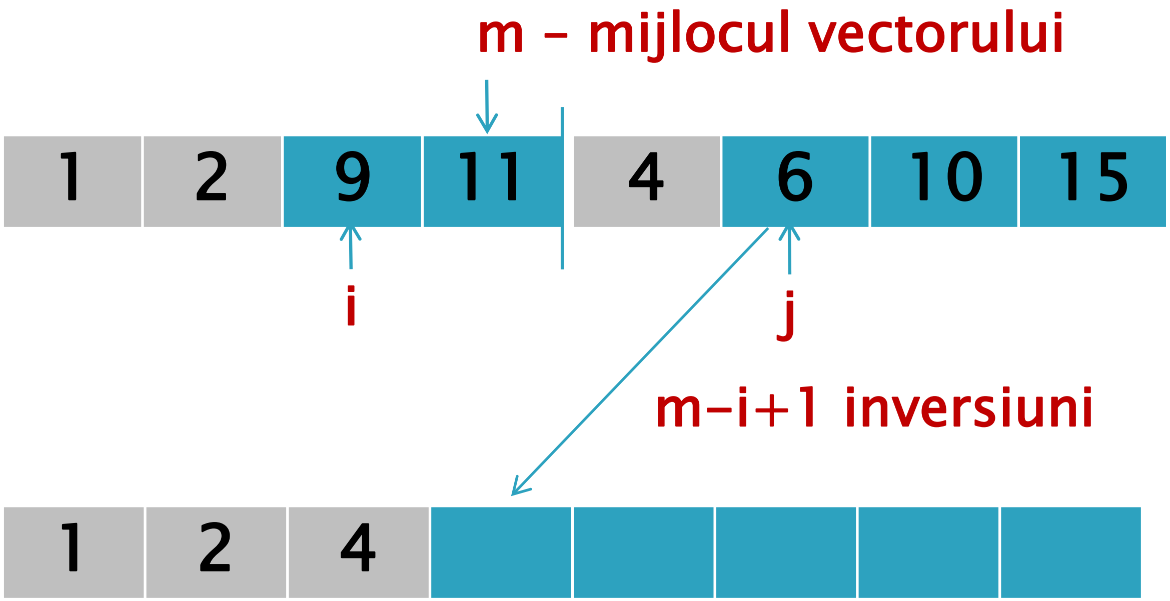

?
1
2
9
11
4
6
10
15
i
j
6 – inversiuni cu 9 si 11
(elementele rămase în
subvectorul stâng)
1
2
4



6 – inversiuni cu 9 si 11 (**elementele rămase în subvectorul stâng**)

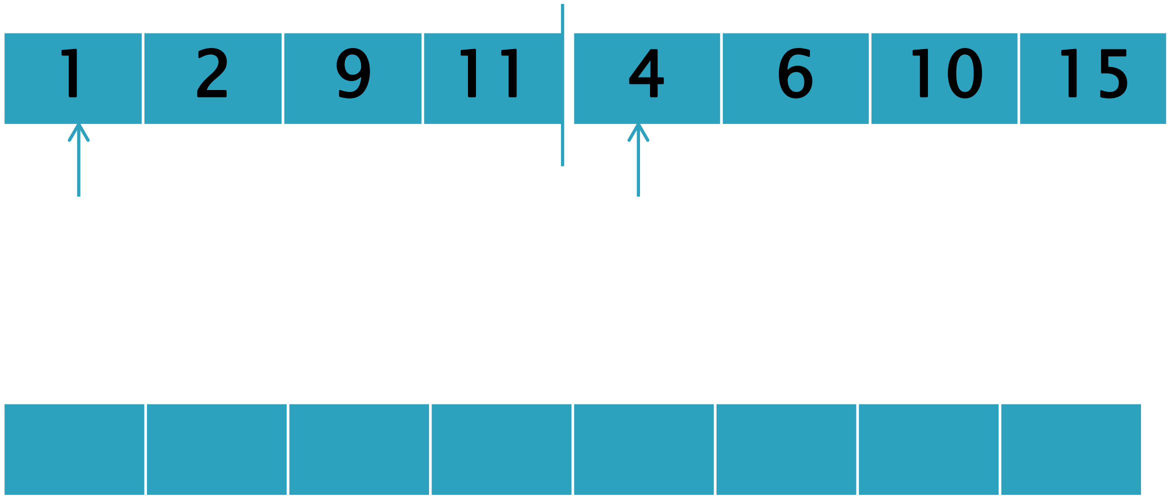
Când a[j] cu j > m este adăugat în vectorul rezultat, el este **mai mic** (**doar**) decât toate elementele din subvectorul stâng **neadăugate** încă în vectorul rezultat

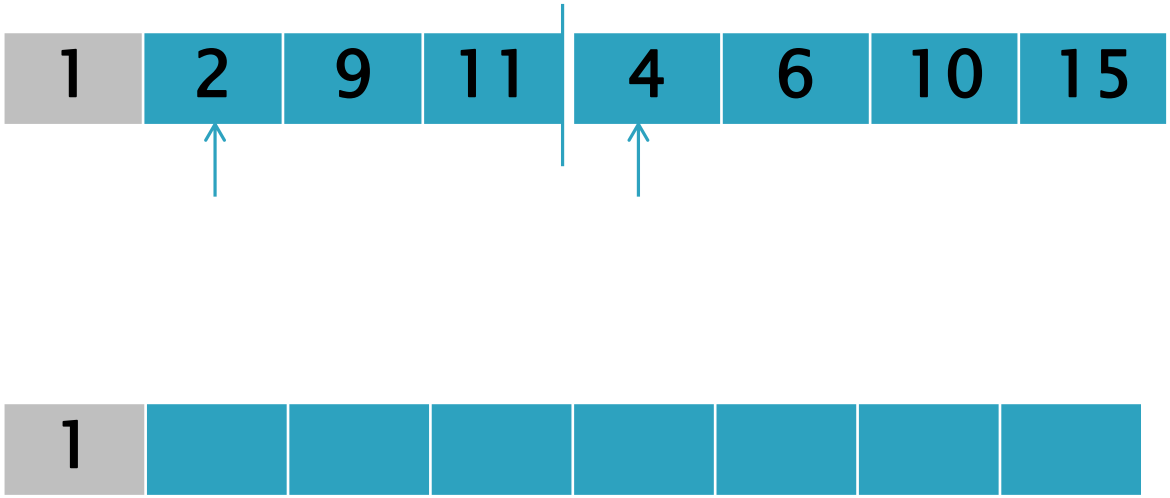
Câte inversiuni determină deci a[j] cu elementele din subvectorul stâng?

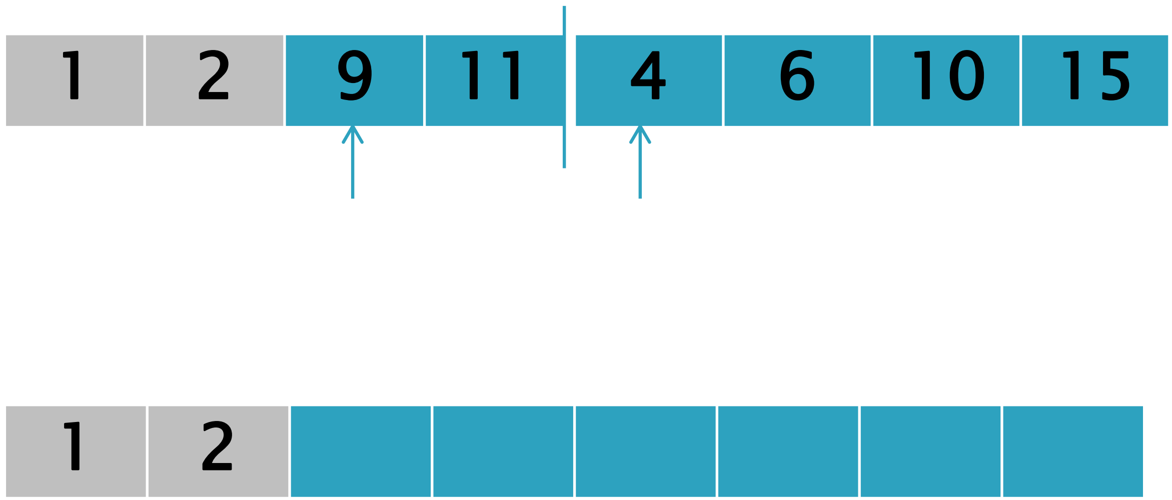


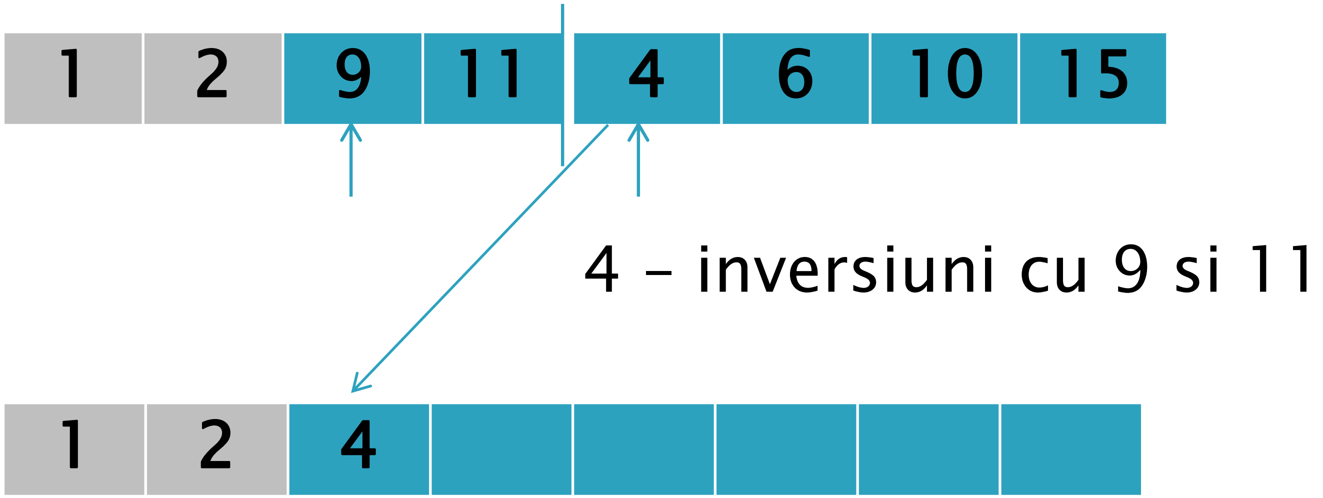
- a[j] determină m – i + 1 inversiuni

Exemplu – numărarea inversiunilor la interclasare

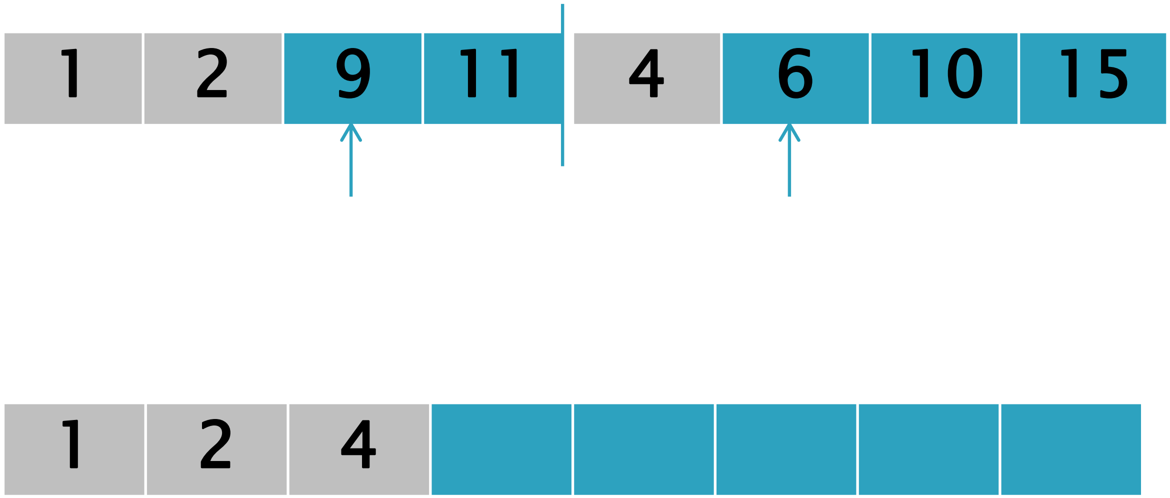

1
2
9
11
4
6
10
15


1
2
9
11
4
6
10
15
1

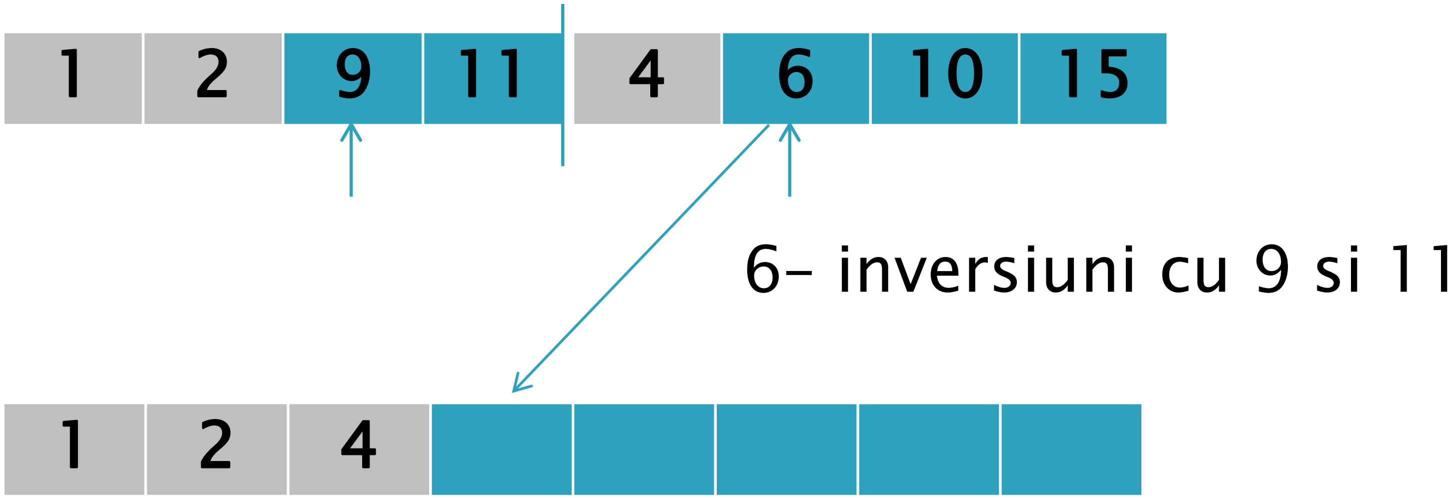

1 2 9 11 4 6 10 15
1 2



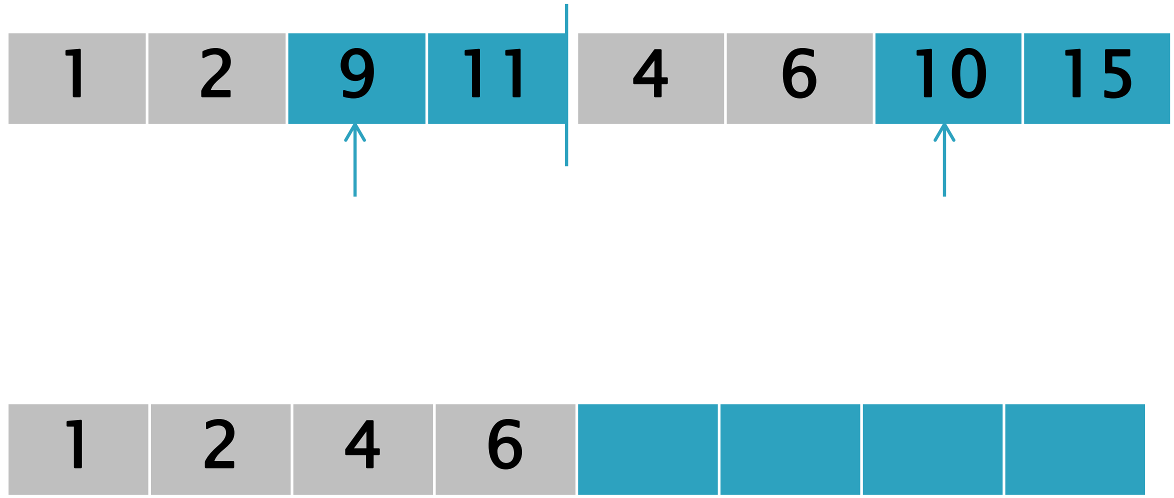
Inversiuni = 2



Inversiuni = 2



Inversiuni = 2 + 2

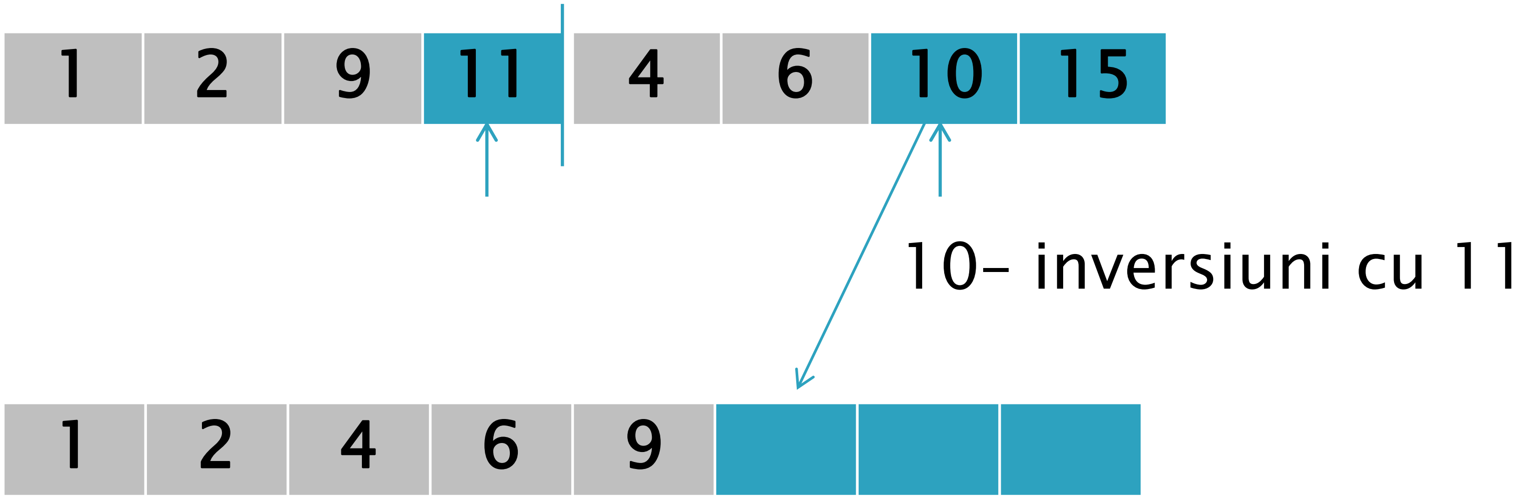


Inversiuni = 2 + 2

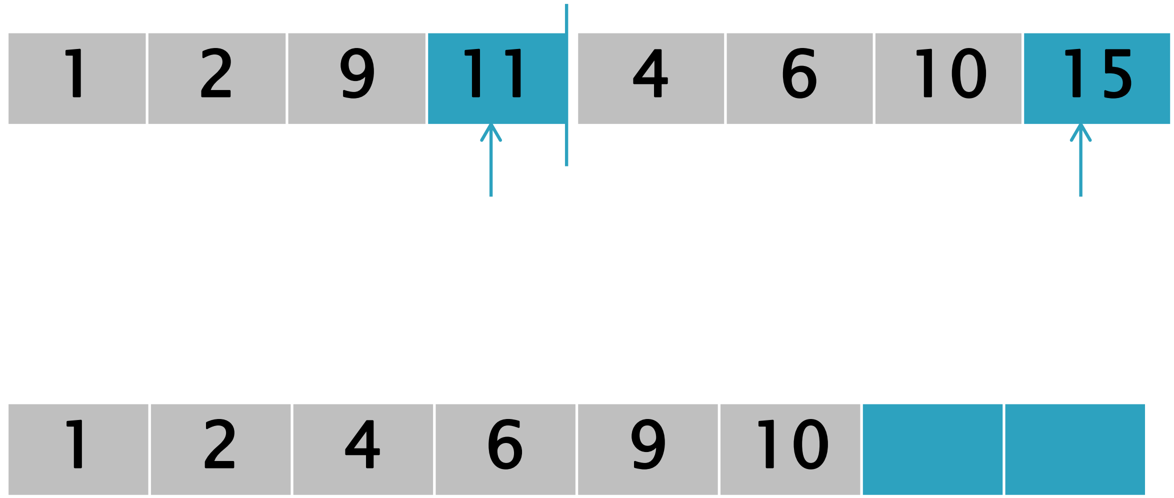
1	2	9	11	4	6	10	15

1	2	4	6	9			

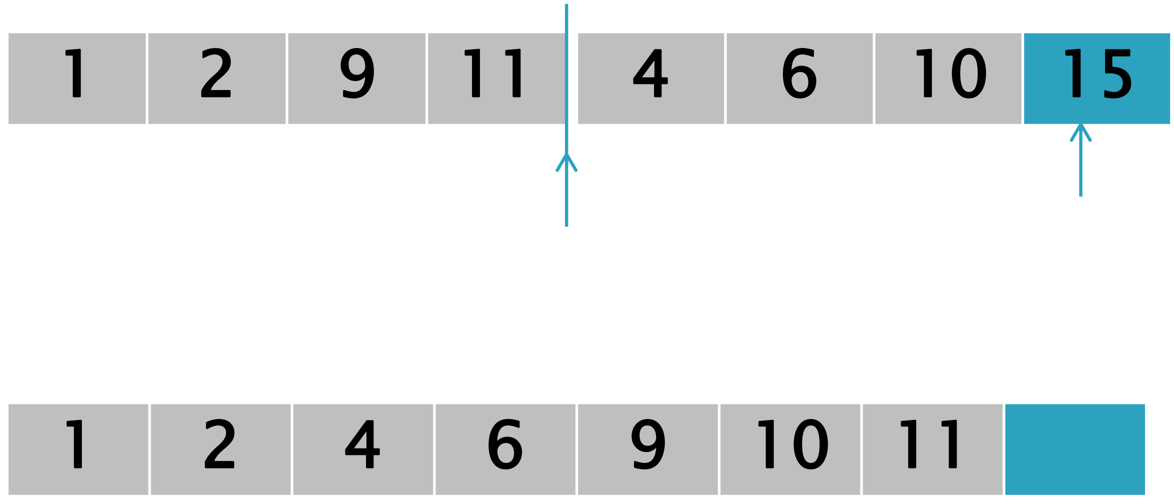
Inversiuni = 2 + 2



Inversiuni = 2 + 2 + 1



Inversiuni = 2 + 2 + 1



Inversiuni = 2 + 2 + 1

1	2	9	11	4	6	10	15

15- nicio inversiune

1	2	4	6	9	10	11	

Inversiuni = 2 + 2 + 1 + 0

1	2	9	11	4	6	10	15

1	2	4	6	9	10	11	15

Inversiuni = 2 + 2 + 1 + 0 = 5

```python
def nr_inversiuni(v, p, u):
    if p==u:
        return 0
    else:
        m = (p+u)//2
        n1 = nr_inversiuni(v, p, m)
        n2 = nr_inversiuni(v, m+1, u)
        return n1+n2+interclaseaza(v, p, m, u)
```

- **Apel:** `x = nr_inversiuni(v,0, len(v)-1)`

```
def interclaseaza(a, p, m, u):
   b = [None]*(u-p+1)
   nr = 0
   i = p;  j = m + 1; k = 0
   while (i<=m) and (j <= u):
         if a[i] <= a[j]:
            b[k] = a[i]; i += 1
         else:
            b[k] = a[j]; j += 1; nr += (m-i+1)
         k+=1

   while i<=m:
      b[k] = a[i]; k += 1; i += 1

   while j<=u:
      b[k] = a[j]; k += 1; j += 1

   for i in range(p,u+1):
      a[i] = b[i-p]

   return nr
```