Laboratorul 1 – Proiectarea algoritmilor
Seria CD

Mihai Nan – mihai.nan.cti@gmail.com

26 Februarie 2020

1 Divide et Impera

1.1 Prezentare generală

Mulți algoritmi utili au o structură recursivă: pentru a rezolva o problemă dată, aceștia sunt apelați de către ei înșiși o dată sau de mai multe ori pentru a rezolva subprobleme apropiate. Acești algoritmi folosesc, de obicei, o abordare de tipul ***Divide et Impera***: ei împart problema în mai multe probleme similare problemei inițiale, dar de dimensiune mai mică, le rezolvă în mod recursiv și apoi le combină pentru a crea o soluție a problemei inițiale.

Tehnica ***Divide et Impera*** implică trei pași la fiecare nivel de recursivitate:

1. **Divide** problema într-un număr de subprobleme.
2. **Stăpânește** subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sunt suficient de mici, rezolvă subproblemele în modul uzual, nerecursiv.
3. **Combină** soluțiile tuturor subproblemelor în soluția finală pentru problema inițială.

1.2 Probleme rezolvate

1. Se dau 3 tije simbolizate prin a, b, c. Pe tija a se găsesc discuri de diametre diferite, așezate în ordine descrescătoare a diametrelor privite de jos în sus. Se cere să se mute de pe tija a pe c, uzitând ca tijă intermediară tija b, respectând următoarele reguli:
   - la fiecare pas se mută un singur disc;
   - nu este permis să se așeze un disc cu diametrul mai mare peste un disc cu diametrul mai mic.

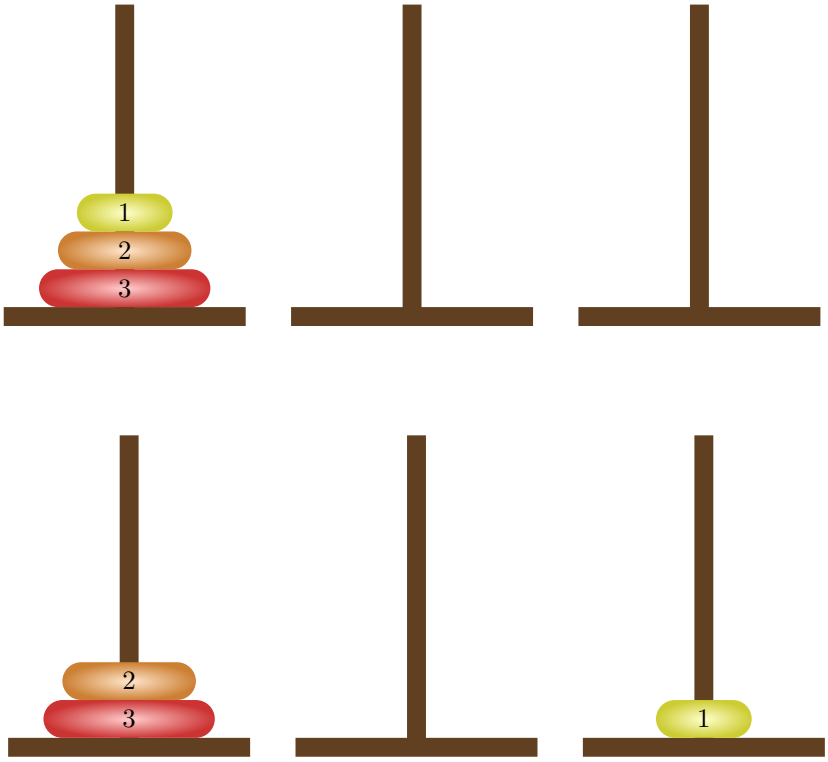
   Soluție: Notăm cu ab o mutare validă a unui disc de pe tija a pe tija b și cu n numărul de discuri.

   (a) Dacă $n = 1$ se face mutarea ac.

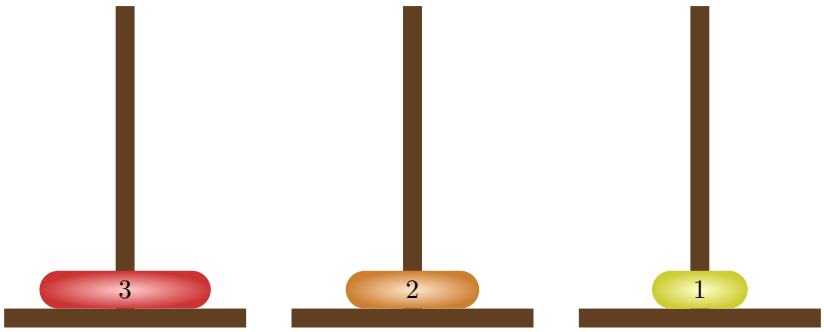
   (b) Dacă $n = 2$ se fac mutările ab, ac, bc.

   (c) Dacă $n > 2$ problema se complică. Notăm cu $H(n, a, b, c)$ șirul mutărilor celor n discuri de pe tija a pe tija c, utilizând ca tijă intermediară tija b. Conform strategiei ***Divide et Impera***, încercăm să descompunem problema în alte doua subprobleme de același tip, urmând apoi combinarea soluțiilor. În acest sens, observăm că mutarea celor n discuri de pe tija a pe tija c este echivalentă cu:
   - mutarea a $n-1$ discuri de pe tija a pe tija b, utilizând ca tijă intermediara tija c;
   - mutarea discului rămas pe tija c;
   - mutarea a $n-1$ discuri de pe tija b pe tija c, utilizând ca tijă intermediară tija a.

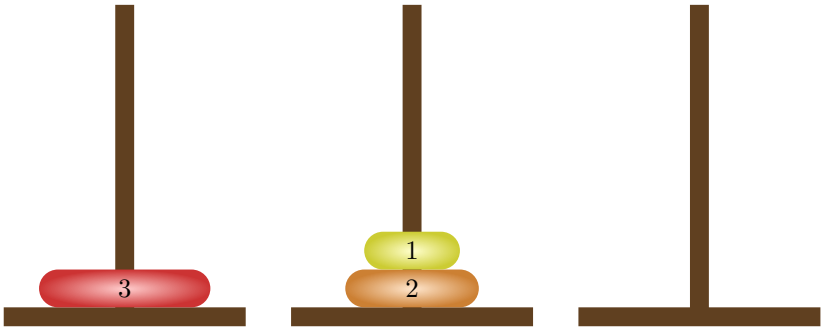
Exemplu



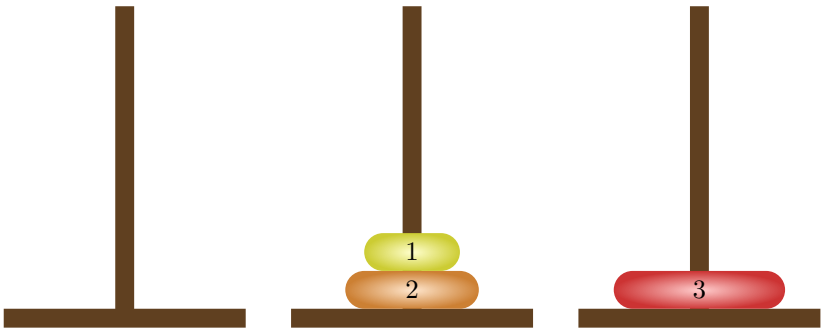
Mutarea discului de pe tija 1 pe tija 3.



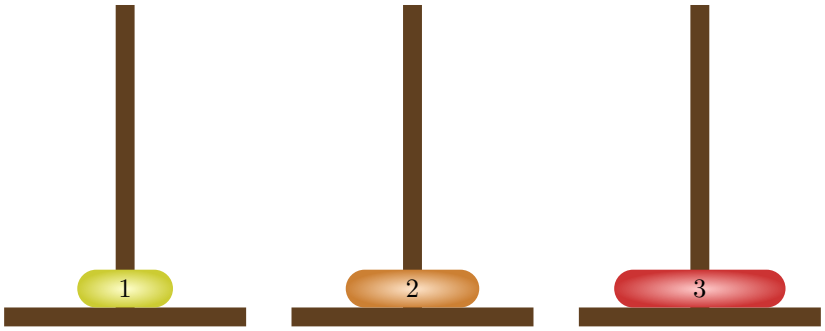
Mutarea discului de pe tija 1 pe tija 2.



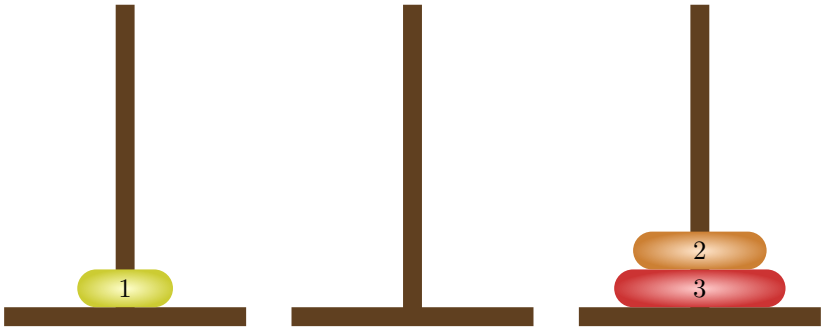
Mutarea discului de pe tija 3 pe tija 2.



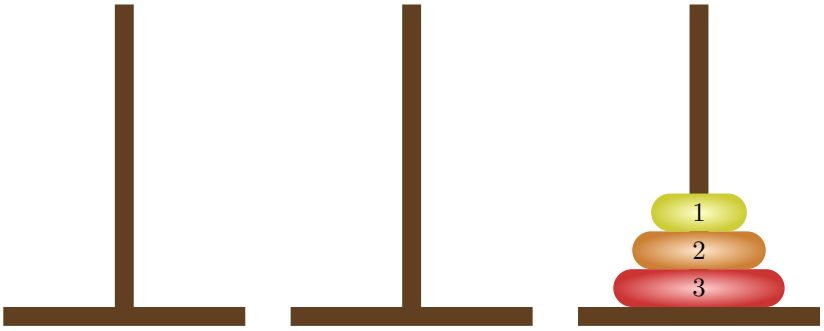
Mutarea discului de pe tija 1 pe tija 3.



Mutarea discului de pe tija 2 pe tija 1.



Mutarea discului de pe tija 2 pe tija 3.



Mutarea discului de pe tija 1 pe tija 3.


```
void tower_of_hanoi(int n, char x, char y, char z) {
        if (n > 0) {
                tower_of_hanoi(n - 1, x, z, y);
                printf("%c to %c\n", x, y);
                tower_of_hanoi(n - 1, z, y, x);
        }
}
```

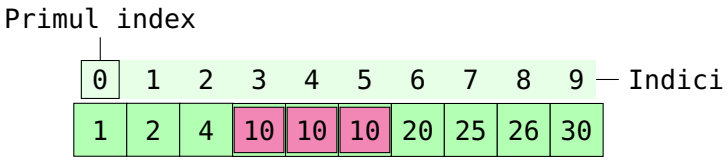
2. Implementați o funcție care primește ca argumente un vector cu ***nr*** elemente **sortate crescător** și o valoare ***x***. Funcția verifică dacă valoarea ***x*** se află în vectorul citit sau nu.

```
int binary_search(int *v, int start, int end, int x) {
        if(start > end) {
                return -1;
        }
        int mid = start + (end - start) / 2;
        if(v[mid] == x)  {
                return mid;
        } else {
                if(v[mid] > x) {
                        return binary_search(v, start, mid - 1, x);
                } else {
                        return binary_search(v, mid + 1, end, x);
                }
        }
}
```

3. Implementați o funcție care primește ca argumente un șir sortat crescător cu n elemente și o valoare x. Funcția determină numărul de elemente egale cu x din șir.

Exemplu

Pentru $n = 10$, $x = 10$ și vectorul următor:



Răspuns: 3 (10 apare de 3 ori în șir)

Soluție

Pentru a determina numărul de apariții ale unui element în vector este suficient să determinăm indexul primei apariții în vector și indexul ultimei apariții. Pentru a putea determina acest lucru, vom folosi algoritmul de căutare binară. Va trebui să modificăm condiția de oprire (cazul de bază) astfel:

- pentru determinarea primei apariții - ne oprim atunci când elementul din mijlocul șirului analizat este egal cu cel căutat, iar în stânga lui există un element mai mic decât el sau el este chiar primul element din șir;

- pentru determinarea ultimei apariții - ne oprim atunci când elementul din mijlocul șirului analizat este egal cu cel căutat, iar în dreapta lui există un element mai mare decât el sau el este chiar ultimul element din șir.

```
int first(int *v, int start, int end, int key) {
        if (end >= start) {
                int mid = start + (end - start) / 2;
                if (v[mid] == key && (mid == 0 || key > v[mid - 1])) {
                        return mid;
                }
                if (v[mid] >= key) {
                        return first(v, start, mid - 1, key);
                }
                return first(v, mid + 1, end, key);
        }
        return -1;

}

int last(int *v, int start, int end, int key, int size) {
        if (end >= start) {
                int mid = start + (end - start) / 2;
                if (v[mid] == key && (mid == size - 1 || key < v[mid + 1])) {
                        return mid;
                }
                if (v[mid] > key) {
                        return last(v, start, mid - 1, key, size);
                }
                return last(v, mid + 1, end, key, size);
        }
        return -1;
}
```

4. Putem folosi ***Divide et Impera*** pentru a ridica mai rapid un număr la o putere.

Observație

Pentru calculul puterii naturale a unui număr întreg, se vor uzita formulele:

$$x^n = x^{n/2} * x^{n/2}, \text{n este par} \quad (1)$$

$$x^n = x * x^{(n-1)/2} * x^{(n-1)/2}, \text{n este impar} \quad (2)$$

```
long putere(long x, long n) {
        if (n == 0)
                return 1;
        long aux = putere(x, n / 2);
        if (n % 2 == 0)
                return aux * aux;
        else
                return x * aux * aux;
}
```

1.3 Probleme propuse

1. Se oferă doi vectori sortați crescător, între cei doi vectori existând o singură diferență. Primul vector conține un element în plus față de al doilea. Determinați indexul elementului care există în primul vector, dar nu și în al doilea.

```
/*
*   arr1 - primul vector sortat crescător
*   arr2 - al doilea vector sortat crescător
*   size - numărul de elemente comune din cei doi vectori
*/
int find_missing_element(int *arr1, int *arr2, int size);
```

2. Dându-se un vector de cuvinte, implementați o funcție, bazată pe Divide et Impera, pentru a determina cel mai lung prefix comun al acestora.

```
/*
*   arr - vectorul de cuvinte
*   size - dimensiunea vectorului de cuvinte
*/
char *longest_common_prefix(char **arr, int size);
```

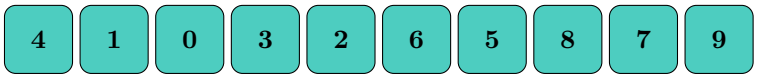
3. Implementați un program care să construiască un arbore binar de căutare, pornind de la șirurile rezultate în urma parcurgerii sale în preordine și inordine.

```
/*
*   inOrder - vectorul care conține parcurgerea în inordine
*   preOrder - vectorul care  conține parcurgerea în preordine
*   inLeft - indicele primului element din vectorul inOrder
*   inRight - indicele ultimului element din vectorul inOrder
*   preIndex - indicele elementului din vectorul preOrder care urmează să fie adăugat
*/
Tree make_binary_tree(int *inOrder, int *preOrder, int inLeft, int inRight, int *preIndex);
```

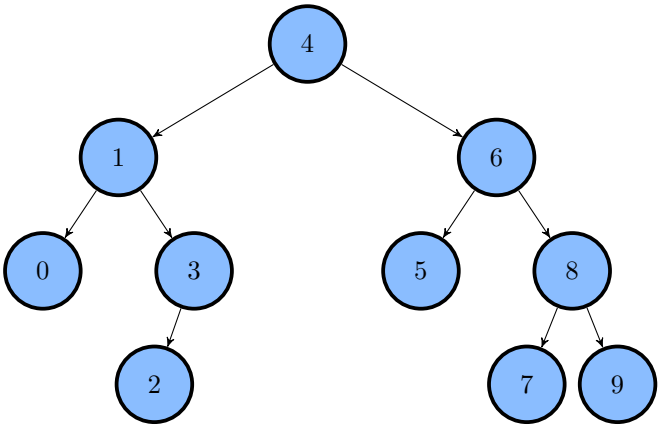
Exemplu

Parcurgerea în inordine

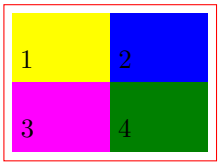
Parcurgerea în preordine



Arborele rezultat

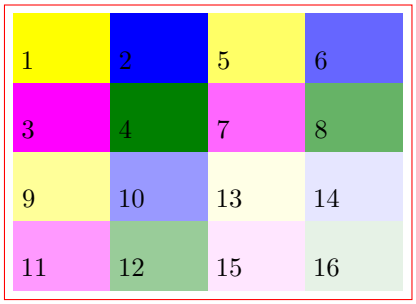


4. Petrică are o tablă pătratică de dimensiuni $2^N \times 2^N$. Acesta ar vrea să scrie pe pătrățelele tablei numere naturale cuprinse între 1 și $2^N \times 2^N$ conform unei parcurgeri mai deosebite pe care o numește Z-parcurgere. O Z-parcurgere vizitează recursiv cele patru cadrane ale tablei în ordinea: stânga-sus, dreapta-sus, stânga-jos, dreapta-jos. De exemplu, dacă $N = 1$, ordinea vizitării pătrățelelor de pe tablă este în formă de Z, ca în figura următoare:




```
/*
*   (x, y) - coordonatele celulei pentru care dorim să aflăm valoarea
*/
int ZParcurgere(int n, int x, int y);
```

Dacă $n = 2$, Petrică va traversa pătrățelele în ordinea:



1.4 Probleme bonus

1. Un șir magic este o permutare definită pe mulțimea $\{1, 2, \ldots, n\}$ care îndeplinește următoarea condiție:

   Pentru orice i, j cu $i < j$, nu există k cu $i < k < j$ astfel încât $A[k] * 2 = A[i] + A[j]$.

   Scrieți o funcție, bazată pe Divide et Impera, care primește ca argument un număr natural n și determină un șir magic de dimensiune n.

   ```
   /*
   *   n - dimensiunea șirului magic pe care dorim să îl determinăm
   */
   int *beautiful_array(int n);
   ```

2. Implementați o funcție care primește ca parametru un șir cu n numere întregi, având tipul unui munte, și determină elementul din vârful muntelui. Dacă vom considera a_i elementul din vârful muntelui, atunci vom avea următoarea relație $a_0 < a_1 < \cdots < a_i > a_{i+1} > \cdots > a_{n-1}$.