CURS 6
TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

În acest curs vom prezenta câteva probleme clasice care se pot rezolva utilizând tehnica de programare Divide et Impera.

1. Problema căutării binare

Fie t un tablou unidimensional format din n numere întregi sortate crescător şi x un număr întreg. Să se verifice dacă valoarea x apare în tabloul t.

Evident, problema ar putea fi rezolvată printr-o simplă parcurgere a tabloului t (*căutare liniară*), obţinând un algoritm având complexitatea $\mathcal{O}(n)$, dar nu am utiliza deloc faptul că elementele tabloului sunt în ordine crescătoare. Pentru a efectua o căutare binară într-o secvenţa $t[st], t[st+1], \dots, t[dr]$ a tabloului t în care $st \leq dr$ vom folosi această ipoteză, comparând valoarea căutată x cu valoarea $t[mij]$ aflată în mijlocul secvenţei. Astfel, vom obţine următoarele 3 cazuri:

a) $x < t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[st], \dots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \dots, t[dr]$;
c) $x = t[mij] \Longrightarrow$ am găsit valoarea x, deci operaţia de căutare se încheie cu succes.

Dacă la un moment dat $st > dr$, înseamnă că nu mai există nicio secvenţă $t[st],, \dots, t[dr]$ în care să aibă sens să căutăm valoarea x, deci operaţia de căutare eşuează.

O implementare a căutării binare în limbajul C, sub forma unei funcţii care furnizează o poziţie pe care apare valoarea x în tabloul t sau valoarea -1 dacă x nu apare deloc în tablou, este următoarea:

```
int cautare_binara(int t[], int st, int dr, int x)
{
    int mij;

    if(st > dr) return -1;
    else
    {
        mij = (st+dr)/2;
        if(x == t[mij]) return mij;
        else
            if(x < t[mij])
                return cautare_binara(t, st, mij-1, x);
            else
                return cautare_binara(t, mij+1, dr, x);
    }
}
```

Se observă faptul că acest algoritm de tip Divide et Impera constă doar din etapa Divide, nemaifiind combinate soluţiilor subproblemelor (etapa Impera). Practic, la fiecare

pas, problema curentă se restrânge la una dintre cele două subprobleme, ci nu se rezolvă ambele subprobleme! Astfel, ţinând cont de faptul că etapa Divide are complexitatea $\mathcal{O}(1)$, relaţia de recurenţă asociată complexităţii algoritmului de căutare binară este următoarea:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Folosind atât iterarea directă a relaţiei de recurenţă, cât şi teorema master, demonstraţi faptul că $T(n) \in \mathcal{O}(\log_2 n)$!

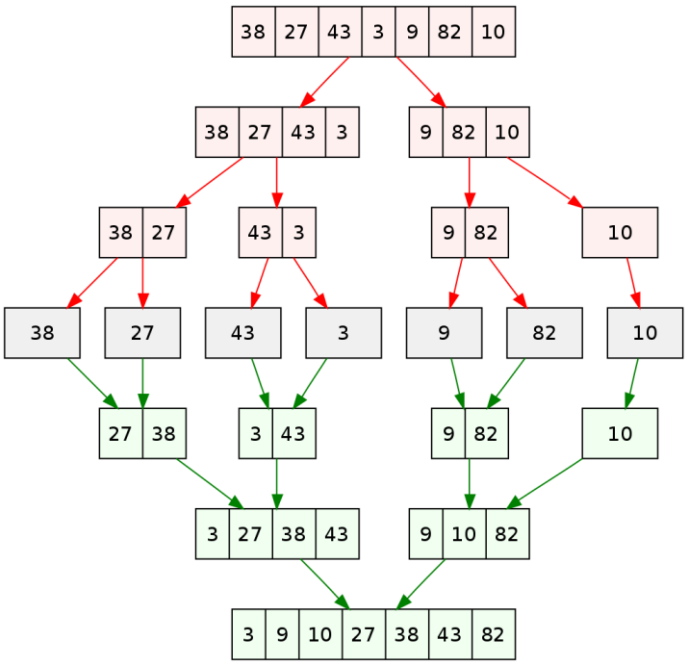
Observaţie importantă: Complexitatea $\mathcal{O}(\log_2 n)$ reprezintă strict complexitatea algoritmului de căutare binară, deci complexitatea unui program, chiar foarte simplu, în care se va utiliza acest algoritm va fi mai mare! De exemplu, un simplu program de test pentru funcţia de mai sus necesită citirea celor n elemente ale tabloului t şi afişarea valorii furnizate de funcţie, deci complexitatea sa va fi $\mathcal{O}(n) + \mathcal{O}(\log_2 n) + \mathcal{O}(1) \approx \mathcal{O}(n)$!

2. Sortarea prin interclasare (Mergesort)

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător un tablou unidimensional de numere, astfel:

- se împarte secvenţa curentă $t[st], \dots, t[dr]$, în mod repetat, în două secvenţe $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$ până când se ajunge la secvenţe implicit sortate, adică secvenţe de lungime 1;
- în sens invers, se sortează secvenţa $t[st], \dots, t[dr]$ interclasând cele două secvenţe în care a fost descompusă, respectiv $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$, şi care au fost deja sortate la un pas anterior.

O reprezentară grafică a modului în care rulează această metodă de sortare se poate observa în următoarea imagine (sursa: https://en.wikipedia.org/wiki/Merge_algorithm):



Începem prezentarea detaliată a acestei metodei de sortare reamintind faptul că interclasarea este un algoritm care permite obținerea unui tablou sortat crescător din două tablouri care sunt, de asemenea, sortate crescător. Considerând dimensiunile tablourilor care vor fi interclasate ca fiind egale cu m și n, complexitatea algoritmului de interclasare este $\mathcal{O}(m+n)$.

În cazul sortării prin interclasare, se vor interclasa secvențele $t[st], \dots, t[mij]$ și $t[mij+1], \dots, t[dr]$ într-un tablou aux alocat dinamic de lungime $dr-st+1$, iar la sfârșit elementele acestuia se vor copia în secvența $t[st], \dots, t[dr]$:

```
void interclasare(int t[], int st, int mij, int dr)
{
	//parcurgem secvența t[st],…,t[mij] cu variabila i,
	//secvența t[mij+1],…,t[dr] cu variabila j,
	//iar tabloul aux cu variabila k
	int i = st, j = mij + 1, k = 0;

	int *aux = (int *)malloc((dr – st + 1)*sizeof(int));

	//cât timp nu am terminat de parcurs niciuna dintre cele
	//două secvențe, copiem în aux[k] minimul dintre
	//elementele curente t[i] și t[j]
	while(i <= mij && j <= dr)
		if(t[i] <= t[j])
			aux[k++] = t[i++];
		else
			aux[k++] = t[j++];

	//dacă au mai rămas în prima secvență elemente neparcurse,
	//le copiem în tabloul aux
	while(i <= mij)
		aux[k++] = t[i++];

	//dacă au mai rămas în a doua secvență elemente neparcurse,
	//le copiem în tabloul aux
	while(j <= dr)
		aux[k++] = t[j++];

	//copiem elementele tabloului aux în secvența t[st],…,t[dr]
	k = 0;
	for(i = st; i <= dr; i++)
		t[i] = aux[k++];

	free(aux);
}
```

Se observă ușor faptul că funcția `interclasare` are o complexitate egală cu $\mathcal{O}(dr-st+1) \leq \mathcal{O}(n)$.

Sortarea tabloului t utilizând se va efectua, folosind tehnica Divide et Impera, în cadrul următoarei funcţii:

```
void mergesort(int t[], int st, int dr)
{
      int mij;

      if(st < dr)
      {
            mij = (st + dr)/2;
            mergesort(t, st, mij);
            mergesort(t, mij+1, dr);
            interclasare(t, st, mij, dr);
      }
}
```

Se observă faptul că, în acest caz, funcţia `interclasare` are rolul funcţiei `soluție` din algoritmul generic Divide et Impera.

Complexitatea funcţiei `mergesort` şi, implicit, complexitatea sortării prin interclasare, se obţine rezolvând următoarea relaţie de recurenţă:

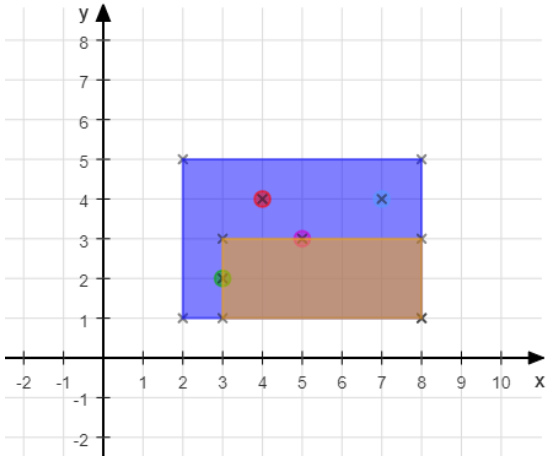
$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

În exemplul 2 de la teorema de master am demonstrat faptul că $T(n) \in \mathcal{O}(n \log_2 n)$, deci sortarea prin interclasare are complexitatea $\mathcal{O}(n \log_2 n)$.

3. Problema tăieturilor într-o placă

Într-o placă dreptunghiulară de lemn în care sunt date mai multe găuri putem efectua, folosind o maşină veche de debitat, doar tăieturi complete pe lungimea sau pe lăţimea plăcii şi paralele cu laturile sale. Să se determine o suprafaţă dreptunghiulară fără găuri cu arie maximă care se poate obţine efectuând doar tăieturi de tipul precizat.

Exemplu:



Placa de lemn este un dreptunghi având colţul stânga-jos de coordonate (2,1) şi colţul dreapta-sus de coordonate (8,5). În placă sunt date 4 găuri, la coordonatele (3,2), (4,4), (5,3) şi (7,4).

Dreptunghiul cu suprafaţa maximă (egală cu 10) şi care nu conţine nici un copac are coordonatele (3,1) pentru colţul stânga-jos şi (8,3) pentru colţul dreapta-sus şi se poate obţine, de exemplu, efectuând o tăietură orizontală completă de-a lungul dreptei $y = 3$ şi apoi a unei tăieturi verticale complete de-a lungul dreptei $x = 3$.

Algoritmul de tip Divide et Impera pentru rezolvarea acestei probleme este foarte simplu, respectiv verificăm dacă în dreptunghiul curent mai există cel puţin o gaură şi apoi tratăm cele două cazuri care pot să apară, astfel:

- dacă în dreptunghiul curent nu mai există nicio gaură, atunci comparăm aria sa cu aria maximă găsită până în acel moment şi, eventual, o actualizăm pe cea din urmă;
- dacă în dreptunghiul curent mai există cel puţin gaură, atunci efectuăm cele două tipuri de tăieturi şi reluăm algoritmul pentru fiecare dintre cele 4 dreptunghiuri care se formează.

Vom implementa în limbajul C algoritmul de mai sus, folosind o funcţie cu antetul:

```
void dreptunghiArieMaxima(int xst, int yst, int xdr, int ydr)
```

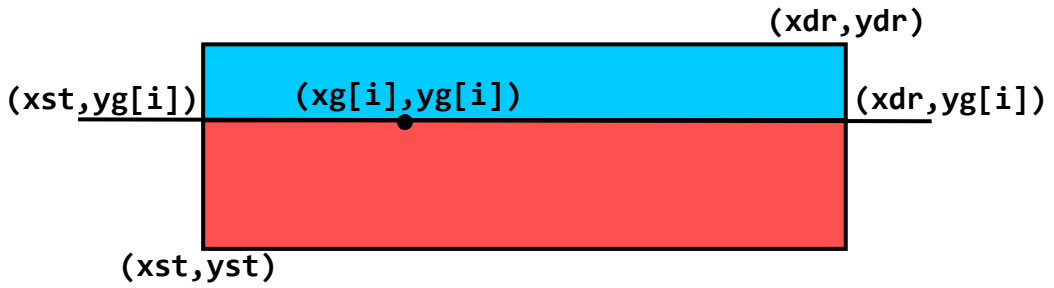
ai cărei parametrii `xst` şi `yst` sunt coordonatele colţului stânga-jos al dreptunghiului curent, iar `xdr` şi `ydr` sunt cele ale colţului dreapta-sus. De asemenea, vom utiliza următoarele variabile globale:

- `int ng`: numărul găurilor din placă (în exemplul dat, avem `ng=4`);
- `int xg[100], yg[100]`: abscisele şi ordonatele găurilor (în exemplul dat, avem `xg=(3, 4, 5, 7)` şi `yg=(2, 4, 3, 4)`);
- `int xst_init, yst_init, xdr_init, ydr_init`: coordonatele colţurilor stânga-jos şi dreapta-sus ale dreptunghiului iniţial (în exemplul dat, avem `xst_init=2, yst_init=1, xdr_init=8, ydr_init=5`) ;
- `int xst_max, yst_max, xdr_max, ydr_max`: coordonatele colţurilor stânga-jos şi dreapta-sus ale dreptunghiului cu arie maxim şi fără găuri (în exemplul dat, vom obţine `xst_max=3, yst_max=1, xdr_max=8, ydr_max=3`).

Dacă în dreptunghiul curent se găseşte o gaură cu coordonatele `xg[i]` şi `yg[i]`, atunci prin efectuarea celor două tipuri de tăieturi prin gaura respectivă se vor obţine următoarele dreptunghiuri:

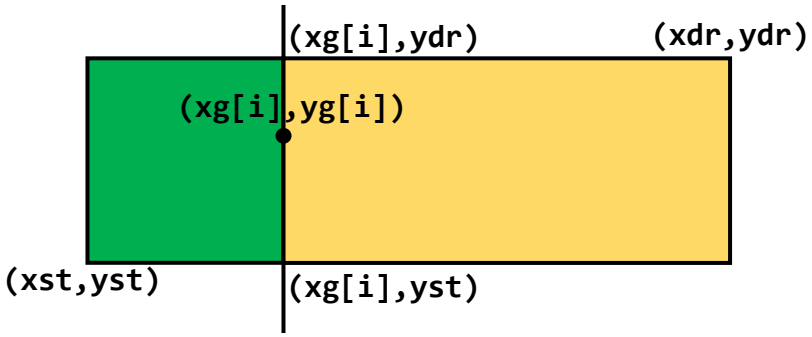
a) după o tăietură orizontală completă:

- *dreptunghiul superior* (albastru) având coordonatele colţului stânga-jos egale cu `xst` şi `yg[i]`, iar cele ale colţului dreapta-sus egale cu `xdr` şi `ydr`;
- *dreptunghiul inferior* (roşu) având coordonatele colţului stânga-jos egale cu `xst` şi `yst`, iar cele ale colţului dreapta-sus egale cu `xdr` şi `yg[i]`.



b) după o tăietură verticală completă:

- *dreptunghiul stâng* (verde) având coordonatele colțului stânga-jos egale cu `xst` și `yst`, iar cele ale colțului dreapta-sus egale cu `xg[i]` și `ydr`;
- *dreptunghiul drept* (galben) având coordonatele colțului stânga-jos egale cu `xg[i]` și `yst`, iar cele ale colțului dreapta-sus egale cu `xdr` și `ydr`.



În continuare, prezentăm codul complet al funcției `dreptunghiArieMaxima`:

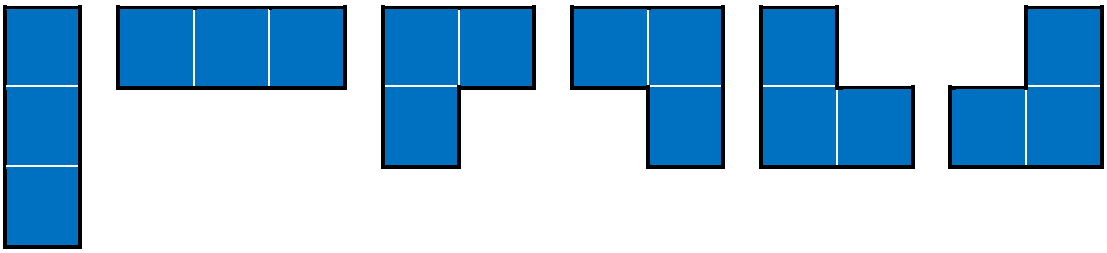
```
void dreptunghiArieMaxima(int xst, int yst, int xdr, int ydr)
{
    int i;

    //verificăm dacă există o gaură în dreptunghiul curent
    for(i = 0; i < ng; i++)
        if(xst < xg[i] && xg[i] < xdr && yst < yg[i] && yg[i] < ydr)
            break;

    //în caz afirmativ, reapelăm funcția pentru cele 4 dreptunghiuri
    //care rezultate în urma efectuării celor două tipuri de tăieturi
    if(i < ng)
    {
        dreptunghiArieMaxima(xst, yg[i], xdr, ydr);
        dreptunghiArieMaxima(xst, yst, xdr, yg[i]);
        dreptunghiArieMaxima(xst, yst, xg[i], ydr);
        dreptunghiArieMaxima(xg[i], yst, xdr, ydr);
    }
    //în caz negativ, comparăm aria dreptunghiului curent care nu
    //conține nicio gaură cu aria maximă a unui dreptunghi fără găuri
    //găsită până în acest moment
    else
        if((ydr-yst)*(xdr-xst)>(ydr_max-yst_max)*(xdr_max-xst_max))
        {
            xst_max = xst;
            yst_max = yst;
            xdr_max = xdr;
            ydr_max = ydr;
        }
}
```

Funcția va fi apelată prin `dreptunghiArieMaxima(xst_init, yst_init, xdr_init, ydr_init)`, iar rezultatul se va găsi în variabilele `xst_max`, `yst_max`, `xdr_max` și `ydr_max`.

Deoarece dimensiunile subproblemelor nu sunt aproximativ egale, nu putem determina complexitatea funcției utilizând cele două metode prezentate. Totuși, folosind teorema Akra-Bazzi, se poate demonstra faptul că funcția are complexitatea $\mathcal{O}(l * L)$, unde l și L reprezintă lățimea și lungimea dreptunghiului inițial, respectiv $l = ydr\_init - yst\_init$ și $L = xdr\_init - xst\_init$.

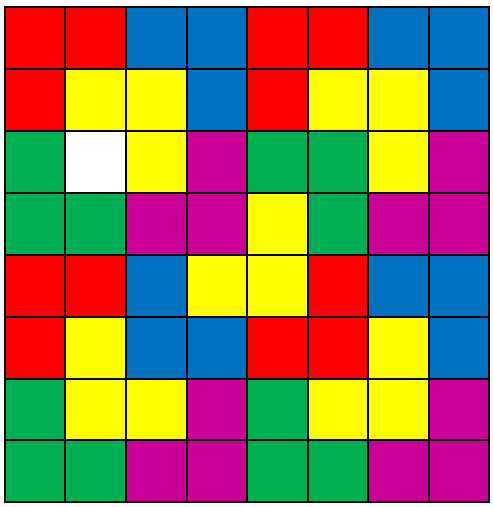
4. L-tromino

Un *tromino* este o figură geometrică plană care se obține prin alipirea a 3 pătrate cu laturi egale:



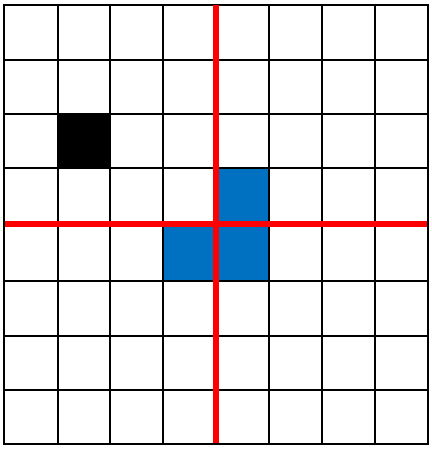
Se observă faptul că există 6 tromino-uri. Primele două se numesc *I-tromino*-uri, iar următoarele 4 se numesc *L-tromino*-uri.

În 1954, matematicianul american Solomon W. Golomb a demonstrat următoarea teoremă: "O rețea formată din $2^n \times 2^n$ pătrate cu laturi egale din care eliminăm un pătrat (îl considerăm ca fiind inaccesibil) poate fi acoperită cu L-tromino-uri care nu se suprapun." (https://www.tandfonline.com/doi/abs/10.1080/00029890.1954.11988548). De exemplu, pentru $n = 3$, o posibilă soluție este următoarea (pătratul alb este cel inaccesibil):



Demonstrația teoremei este foarte simplă și folosește inducția matematică:

- pentru $n = 1$ teorema este evident adevărată, deoarece o rețea de dimensiune 2×2 din care eliminăm un pătrat este chiar un L-tromino;
- presupunem că o rețea de dimensiune $2^n \times 2^n$ din care eliminăm un pătrat poate fi acoperită cu L-tromino-uri care nu se suprapun și demonstrăm că o rețea de dimensiune $2^{n+1} \times 2^{n+1}$ din care eliminăm un pătrat poate fi acoperită cu L-tromino-uri care nu se suprapun. Pentru a demonstra acest lucru, împărțim rețeaua de dimensiune $2^{n+1} \times 2^{n+1}$ în 4 sub-rețele de dimensiune $2^n \times 2^n$ (am marcat cu negru pătratul inaccesibil) și adăugăm în colțurile celor 3 sub-rețele de dimensiune $2^n \times 2^n$ care nu conțin pătratul inaccesibil un L-tromino, astfel:



În acest mod, am obținut 4 sub-rețele de dimensiune $2^n \times 2^n$ care conțin, fiecare, câte un pătrat inaccesibil, deci, conform ipotezei de inducție, toate cele 4 sub-rețele pot fi acoperite cu L-tromino-uri care nu se suprapun. Astfel, am demonstrat că rețeaua de dimensiune $2^{n+1} \times 2^{n+1}$ din care eliminăm un pătrat poate fi acoperită cu L-tromino-uri care nu se suprapun, ceea ce încheie demonstrația teoremei.

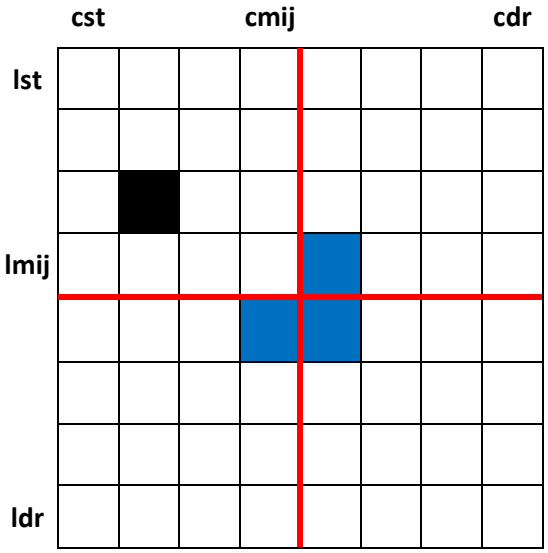
Utilizând ideea din demonstrația teoremei lui Golomb, putem elabora un algoritm de tip Divide et Impera care să acopere o rețea de dimensiune $2^n \times 2^n$ din care eliminăm un pătrat cu L-tromino-uri care nu se suprapun. În acest scop, vom utiliza o matrice pătratică `t` ale cărei elemente le vom inițializa cu 0, iar pătratul inaccesibil îl vom marca prin valoarea -1. De asemenea, vom folosi o variabilă globală `crt` care va reține numărul de ordine al L-tromino-ului curent, deci o vom inițializa cu 1. Pentru exemplul de mai sus, după terminarea algoritmului, matricea `t` va fi următoarea (pătratul inaccesibil este cel alb):

3	3	4	4	8	8	9	9
3	2	2	4	8	7	7	9
6	-1	2	5	11	11	7	10
6	6	5	5	1	11	10	10
18	18	19	1	1	13	14	14
18	17	19	19	13	13	12	14
21	17	17	20	16	12	12	15
21	21	20	20	16	16	15	15

Algoritmul de tip Divide et Impera va fi implementat utilizând o funcţie cu următorul antet:

```
void tromino(int lst, int cst, int ldr, int cdr, int lpi, int cpi)
```

ai cărei parametrii au următoarele semnificaţii: `lst` şi `cst` reprezintă linia şi coloana pătratului din colţul stânga-sus al reţelei curente, `ldr` şi `cdr` reprezintă linia şi coloana pătratului din colţul dreapta-jos al reţelei curente, iar `lpi` şi `cpi` reprezintă linia şi coloana pătratului inaccesibil. În cadrul funcţiei vom utiliza două variabile locale `lmij` şi `cmij` care vor conţine valorile liniei şi coloanei mediane din reţeaua curentă, valori pe care le vom utiliza la divizarea reţelei curente în 4 sub-reţele având lungimea laturii egală cu jumătate din lungimea laturii reţelei iniţiale:



Codul sursă complet al funcţiei `tromino` este următorul:

```
void tromino(int lst, int cst, int ldr, int cdr, int lpi, int cpi)
{
    int lmij, cmij;

    //dacă sub-rețeaua curentă este de dimensiune 2x2, atunci ea este
    //un L-tromino, deci îi marcăm pătratele componente cu numărul
    //său de ordine crt și actualizăm valoarea lui crt
    if (ldr - lst == 1 && cdr - cst == 1)
    {
        if(t[lst][cst] == 0) t[lst][cst] = crt;
        if(t[lst][cdr] == 0) t[lst][cdr] = crt;
        if(t[ldr][cst] == 0) t[ldr][cst] = crt;
        if(t[ldr][cdr] == 0) t[ldr][cdr] = crt;
        crt++;
    }
    else
    {
         //calculăm linia și coloana mediană a rețelei curente
```

```
lmij = (lst+ldr)/2;
cmij = (cst+cdr)/2;

 //pătratul inaccesibil se găsește în sub-rețeaua stânga-sus
if(lpi <= lmij && cpi <= cmij)
{
    //adăugăm un L-tromino în colțurile celorlalte 3
    //sub-rețele care nu conțin pătratul inaccesibil
    t[lmij][cmij+1] = crt;
    t[lmij+1][cmij] = crt;
    t[lmij+1][cmij+1] = crt;
    crt++;

    //reapelăm funcția pentru fiecare dintre cele 4 sub-rețele
    //care conțin, fiecare, câte un pătrat inaccesibil

    //sub-rețeaua stânga-sus
    tromino(lst, cst, lmij, cmij, lpi, cpi);
    //sub-rețeaua dreapta-sus
    tromino(lst, cmij+1, lmij, cdr, lmij, cmij+1);
    //sub-rețeaua dreapta-jos
    tromino(lmij+1, cmij+1, ldr, cdr, lmij+1, cmij+1);
    //sub-rețeaua stânga-jos
    tromino(lmij+1, cst, ldr, cmij, lmij+1, cmij);
}

//pătratul inaccesibil se găsește în sub-rețeaua dreapta-sus
if(lpi <= lmij && cpi > cmij)
{
    //adăugăm un L-tromino în colțurile celorlalte 3
    //sub-rețele care nu conțin pătratul inaccesibil
    t[lmij][cmij] = crt;
    t[lmij+1][cmij] = crt;
    t[lmij+1][cmij+1] = crt;
    crt++;

    //reapelăm funcția pentru fiecare dintre cele 4 sub-rețele
    //care acum conțin, fiecare, câte un pătrat inaccesibil

    //sub-rețeaua stânga-sus
    tromino(lst, cst, lmij, cmij, lmij, cmij);
    //sub-rețeaua dreapta-sus
    tromino(lst, cmij+1, lmij, cdr, lpi, cpi+1);
    //sub-rețeaua dreapta-jos
    tromino(lmij+1, cmij+1, ldr, cdr, lmij+1, cmij+1);
    //sub-rețeaua stânga-jos
    tromino(lmij+1, cst, ldr, cmij, lmij+1, cmij);
}
```

```
        //pătratul inaccesibil se găsește în sub-rețeaua dreapta-jos
        if(lpi > lmij && cpi > cmij)
        {
            //adăugăm un L-tromino în colțurile celorlalte 3
            //sub-rețele care nu conțin pătratul inaccesibil
            t[lmij][cmij] = crt;
            t[lmij][cmij+1] = crt;
            t[lmij+1][cmij] = crt;
            crt++;
            //reapelăm funcția pentru fiecare dintre cele 4 sub-rețele
            //care acum conțin, fiecare, câte un pătrat inaccesibil

            //sub-rețeaua stânga-sus
            tromino(lst, cst, lmij, cmij, lmij, cmij);
            //sub-rețeaua drepta-sus
            tromino(lst, cmij+1, lmij, cdr, lmij, cmij+1);
            //sub-rețeaua dreapta-jos
            tromino(lmij+1, cmij+1, ldr, cdr, lpi, cpi);
            //sub-rețeaua stânga-sus
            tromino(lmij+1, cst, ldr, cmij, lmij+1, cmij);
        }

        //pătratul inaccesibil se găsește în sub-rețeaua stânga-jos
        if(lpi > lmij && cpi <= cmij)
        {
            //adăugăm un L-tromino în colțurile celorlalte 3
            //sub-rețele care nu conțin pătratul inaccesibil
            t[lmij][cmij] = crt;
            t[lmij][cmij+1] = crt;
            t[lmij+1][cmij+1] = crt;
            crt++;

            //reapelăm funcția pentru fiecare dintre cele 4 sub-rețele
            //care acum conțin, fiecare, câte un pătrat inaccesibil

            //sub-rețeaua stânga-sus
            tromino(lst, cst, lmij, cmij, lmij, cmij);
            //sub-rețeaua drepta-sus
            tromino(lst, cmij+1, lmij, cdr, lmij, cmij+1);
            //sub-rețeaua drepta-jos
            tromino(lmij+1, cmij+1, ldr, cdr, lmij+1, cmij+1);
            //sub-rețeaua stânga-sus
            tromino(lmij+1, cst, ldr, cmij, lpi, cpi);
        }
    }
}
```

Dacă notăm cu $L = 2^n$ lungimea laturii rețelei inițiale (în limbajul C, valoarea `L` se poate calcula foarte simplu, astfel: `L = 1<<n`), atunci apelul inițial al funcției `tromino`

este `tromino(0, 0, L-1, L-1, lpi, cpi)`, unde `lpi` și `cpi` reprezintă linia și coloana pătratului inaccesibil.

Ținând cont de faptul că o rețea pătratică cu latura L se împarte în 4 sub-rețele cu laturile de $\frac{L}{2}$, putem determina complexitatea funcției `tromino` folosind următoarea relație de recurență (etapa Divide este realizată de instrucțiunile subordonate ramurii `else` a primei instrucțiuni `if` din cadrul funcției, iar etapa Impera lipsește):

$$T(L) = 4T\left(\frac{L}{2}\right) + 1$$

Aplicând primul caz al Teoremei master, obținem că $T(L) \in \mathcal{O}(L^2) \Rightarrow T(n) \in \mathcal{O}(2^{2n})$, deci funcția are o complexitate exponențială! Justificați complexitatea algoritmului calculând numărul de L-tromino-uri necesare pentru acoperirea rețelei!