Recursivitate

CAPITOLUL IX

Cuprins

Introducere

• **O definiţie recursivă** este acea definiţie care se referă la un obiect care se defineşte ca parte a propriei sale definiri.

• Desigur o definiţie de genul "o floare este o floare" încalcă principiul circularității și este greșită.

• O caracteristică foarte importantă a recursivităţii este aceea de a preciza o definiţie într-un sens evolutiv, care evită circularitatea.

• Spre exemplu o **definiţie recursivă** este următoarea: *"Un buchet de flori este: (1) fie o floare, (2) fie o floare adăugată buchetului "*.

- Afirmaţia (1) serveşte ca şi **condiţie iniţială**, indicând maniera de amorsare a definiţiei.
- Afirmaţia (2) precizează **definirea recursivă** (**evolutivă**) **propriu-zisă**.

Varianta iterativă a aceleaşi definiţii este *"Un buchet de flori constă fie dintr-o floare, fie din două, fie din 3 flori, fie ... etc"*.

• După cum se observă, definiţia recursivă este **simplă** şi **elegantă** dar oarecum indirectă, în schimb definiţia iterativă este directă dar greoaie şi lipsită de eleganţă.

Introducere

Despre un **obiect** se spune că este **recursiv** dacă el **constă** sau **este definit prin el însuşi.**

• Prin **definiţie** orice **obiect recursiv** implică **recursivitatea** ca şi proprietate **intrinsecă** a obiectului în cauză.

• **Recursivitatea** este utilizată cu multă eficienţă în **matematică**, spre exemplu în definirea numerelor naturale, a structurilor arbore sau a anumitor funcţii

• **Numerele naturale**:

(1) 0 este un număr natural.

(2) Succesorul unui număr natural este un număr natural.

(3) Nu sunt alte elemente în mulțimea numerelor naturale (între un număr și succesorul său).

• Funcţia **factorial** n! (definită pentru întregi pozitivi):

(1) 0!= 1

(2) Dacă n > 0 , atunci n!= n*(n-1)!

Introducere

• **Fractali**:

Un fractal este o figură geometrică ce poate fi divizată în mai multe părți, astfel încât fiecare dintre acestea să fie o copie în miniatură a originalului.

Câteva exemple naturale de fractali sunt ferigile, fulgii de zăpadă, cochiliile de melci și fulgerele.

https://infogenius.ro/fractali-p5js/

Introducere

Șiruri recurente ca de exemplu:

- progresie aritmetică:

$$\begin{cases} x_0 = b \\ x_n = x_{n-1} + r, pentru\ n > 0 \end{cases}$$

Exemplu: 1,4,7,10,13,... (b = 1, r = 3)

- progresie geometrică:

$$\begin{cases} x_0 = b \\ x_n = x_{n-1} * r, pentru\ n > 0 \end{cases}$$

Exemplu: 3,6,12,24,48,... (b = 3, r = 2)

Nu se calculează x_n direct, ci din aproape în aproape, folosind x_{n-1}.

Introducere

Recursivitatea este importantă în informatică pentru că reduce o problemă la **un caz mai simplu al aceleiași probleme**

$$\text{obiecte: un șir este} \begin{cases} \text{(un singur element)} \\ \text{un element urmat de un șir} \end{cases}$$

ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

Structura unui apel recursiv

Un exemplu clasic de funcție recursivă este ridicarea la o putere pozitivă, întreagă a unui număr real. Funcția poate fi definită matematic în felul următor:

$$x^n = \begin{cases} 1, dacă\ n = 0 \\ x * x^{n-1}, dacă\ n > 0 \end{cases}$$

O implementare C poate fi:

```
double pwr(double x, unsigned n)
{
 if (n == 0)
      return 1.0;
 return x*pwr(x, n - 1);
}
```

Structura unui apel recursiv

```
double pwr(double x, unsigned n)
{if (n == 0)      //caz de bază
     return 1.0;
return x*pwr(x, n - 1);}  //apel recursiv
```

La fel ca în cazul inducției matematice, în cazul unei funcții recursive avem două componente:

- Un caz de **bază**, ce corespunde pasului inițial din inducție
- Apelul **recursiv**, ce corespunde pasului inductiv

Verificarea programelor recursive se poate face formal, printr-o demonstrație inductivă a formulei de recurență.

Pentru exemplul anterior demonstrarea corectitudinii implică doi pași, pentru n==0, valoarea 1 este corectă, iar pentru n>0, presupunând că valoarea calculata pentru predecesorul lui n pwr(x,n-1), prin înmulțirea acesteia cu x, se obține valoarea corectă pentru pwr(x,n).

Mecanismul implementării recursivității

Structurile de program necesare şi suficiente pentru exprimarea recursivităţii sunt **funcţiile** care pot fi apelate prin nume și **stiva sistemului**.

Considerăm funcția de ridicare la putere ce poate fi definită matematic în felul următor:

$$x^n = \begin{cases} 1, dacă\ n = 0 \\ x * x^{n-1}, dacă\ n > 0 \end{cases}$$

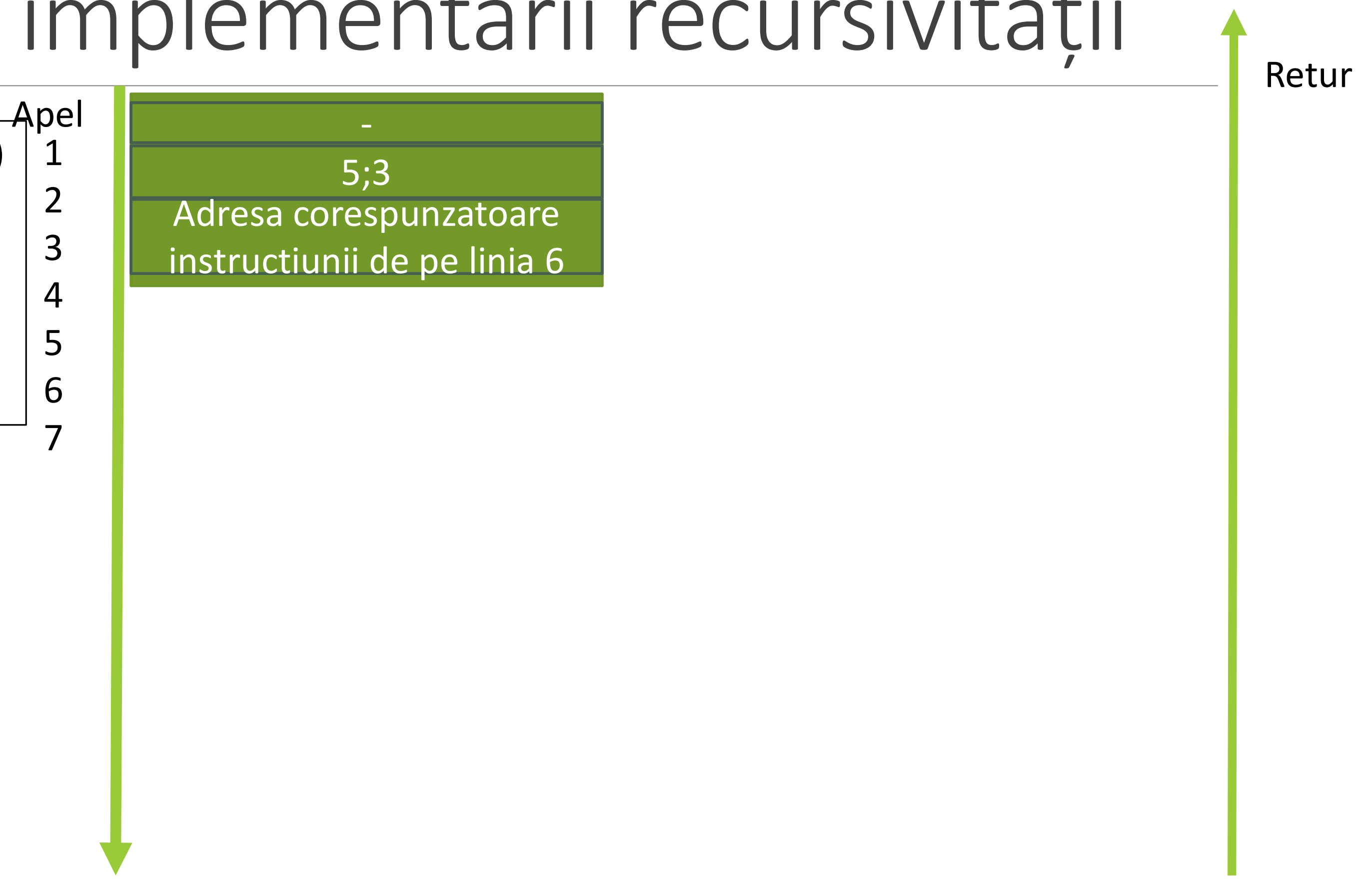
Folosind definiția anterioară, x^3 poate fi calculat în felul următor:

$$x^3 = x * x^2 = x * (x * x^1) = x * \left(x * (x * x^0)\right) = x * \left(x * (x * 1)\right) = x * \left(x * (x)\right) = x * (x * x) = x * x * x$$

În continuare exemplificăm starea stivei sistem pentru apelul unei funcții pwr care implementează ridicare la putere, pentru x=5 și n=3.

Mecanismul implementării recursivității

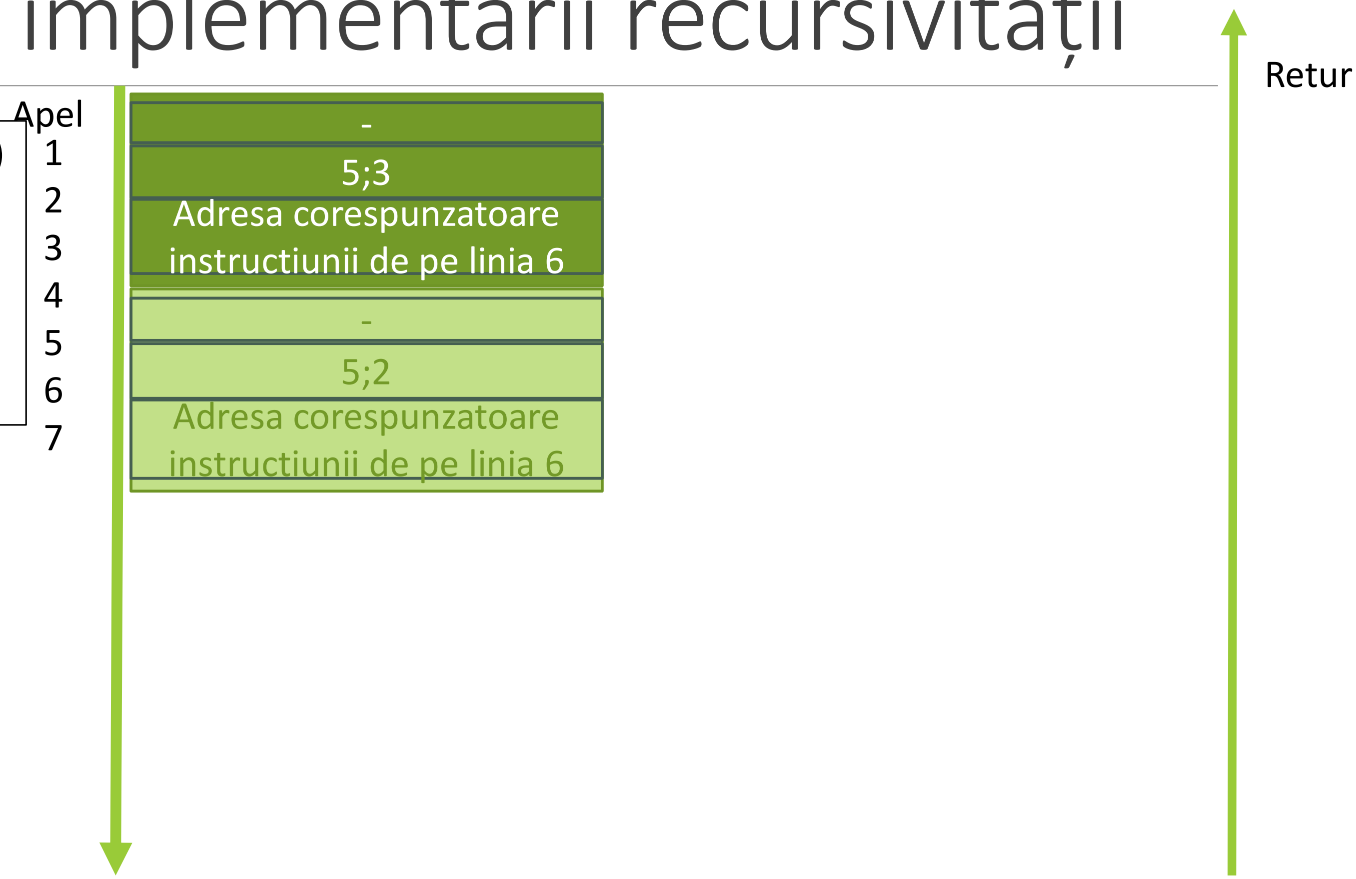
```
1  double pwr(double x, unsigned n)
2  {
3          if (n == 0)
4                  return 1.0;
5          return x*pwr(x, n - 1);
6  }
7
```



Exemplu apel pwr(5,3)

Mecanismul implementării recursivității

```
  double pwr(double x, unsigned n)      1
  {                                     2
      if (n == 0)                       3
          return 1.0;                   4
      return x*pwr(x, n - 1);           5
  }                                     6
                                        7
```

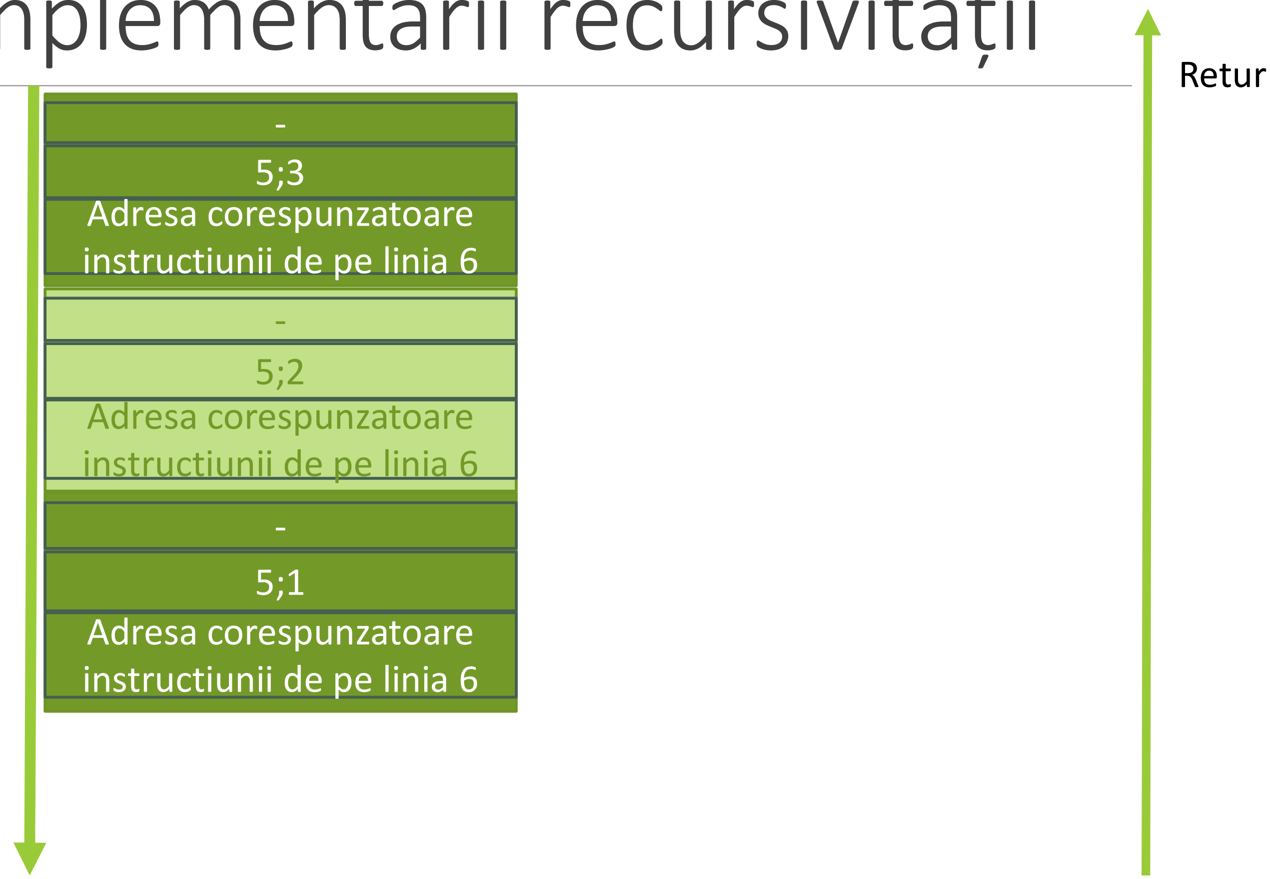


Exemplu apel pwr(5,3)

Mecanismul implementării recursivității

```
1  double pwr(double x, unsigned n)
2  {
3          if (n == 0)
4                  return 1.0;
5          return x*pwr(x, n - 1);
6  }
7
```

Exemplu apel pwr(5,3)

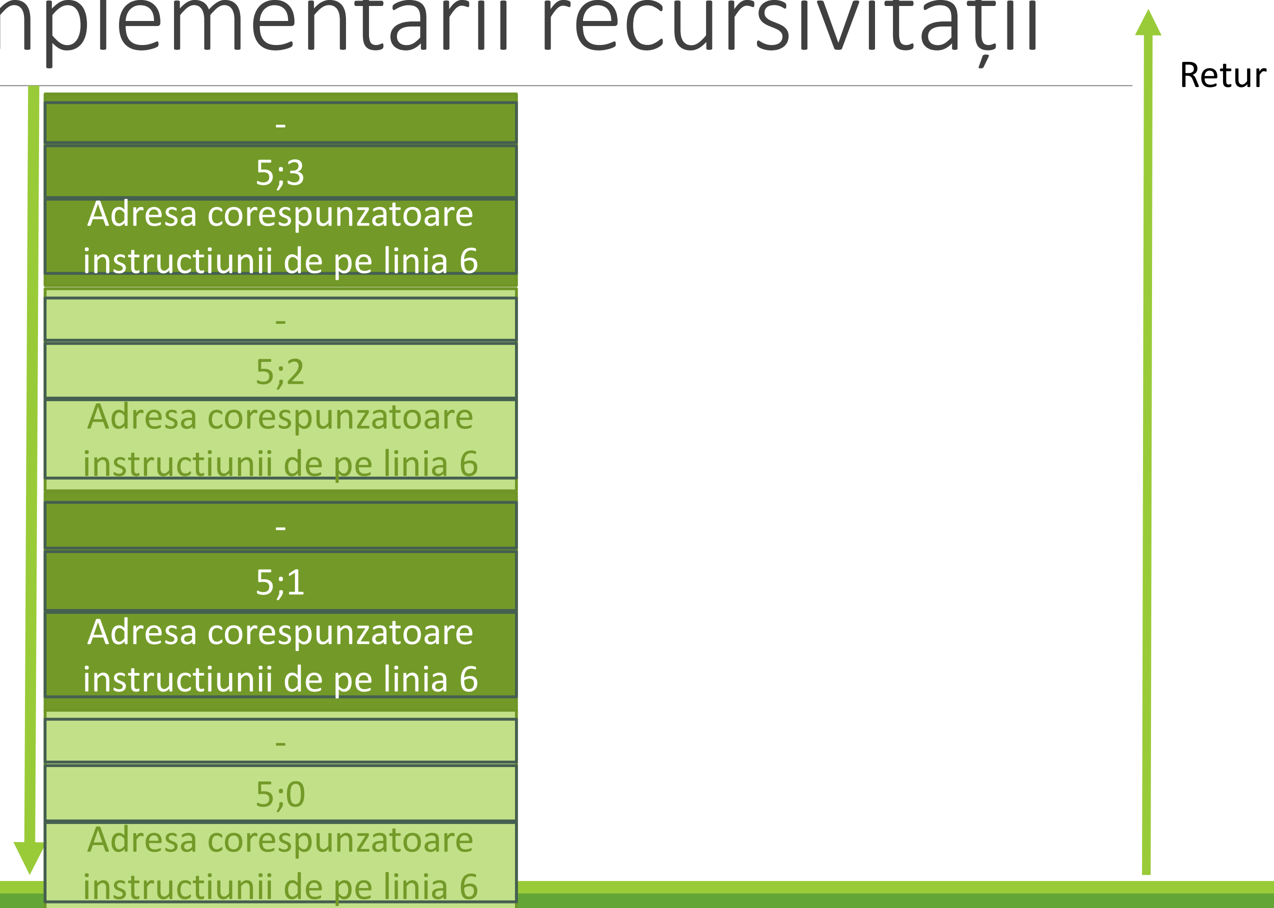


Mecanismul implementării recursivității

Apel

```
1 double pwr(double x, unsigned n)
2 {
3         if (n == 0)
4                 return 1.0;
5         return x*pwr(x, n - 1);
6 }
7
```

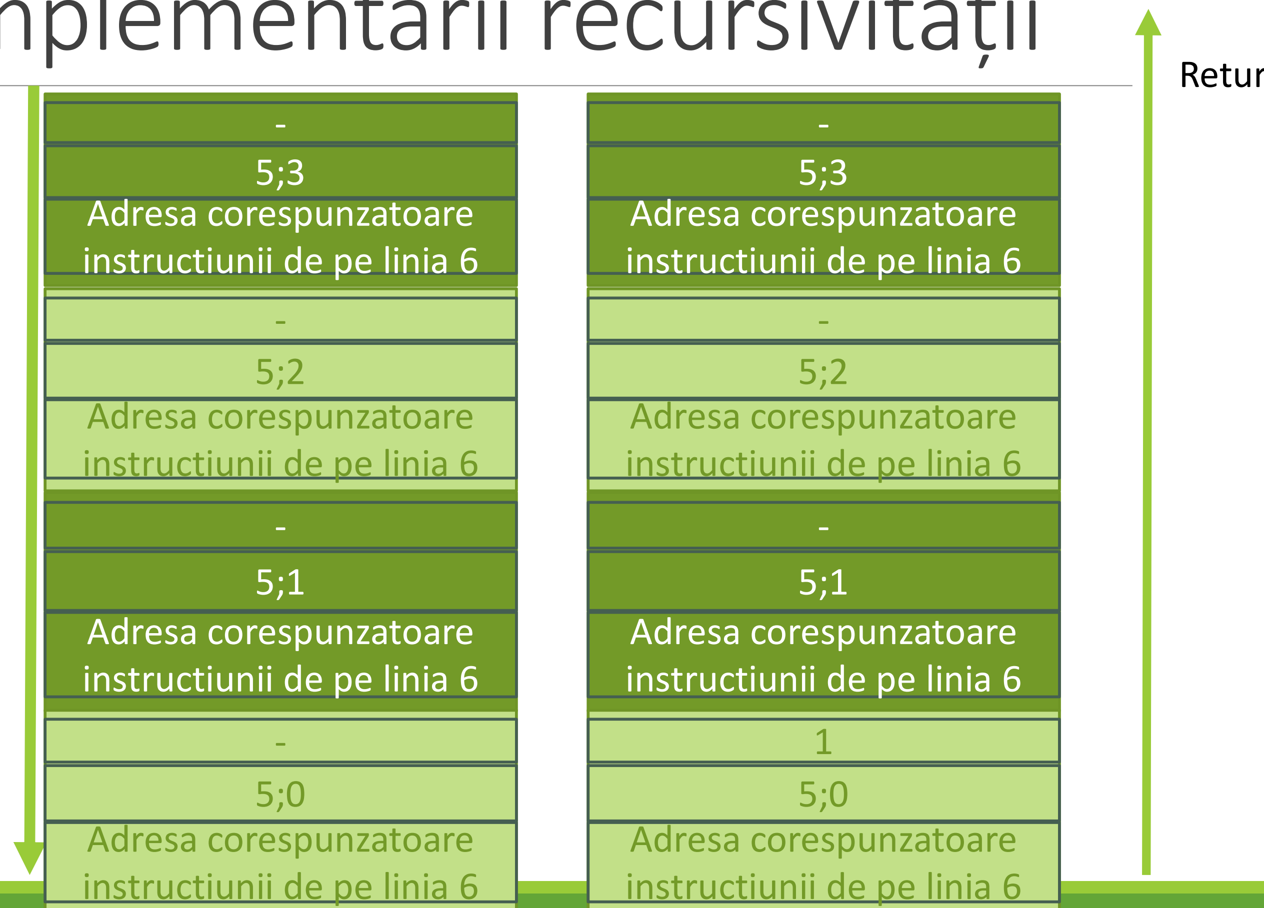
Exemplu apel pwr(5,3)



Mecanismul implementării recursivității

```
1  double pwr(double x, unsigned n)
2  {
3          if (n == 0)
4                  return 1.0;
5          return x*pwr(x, n - 1);
6  }
7
```

Apel

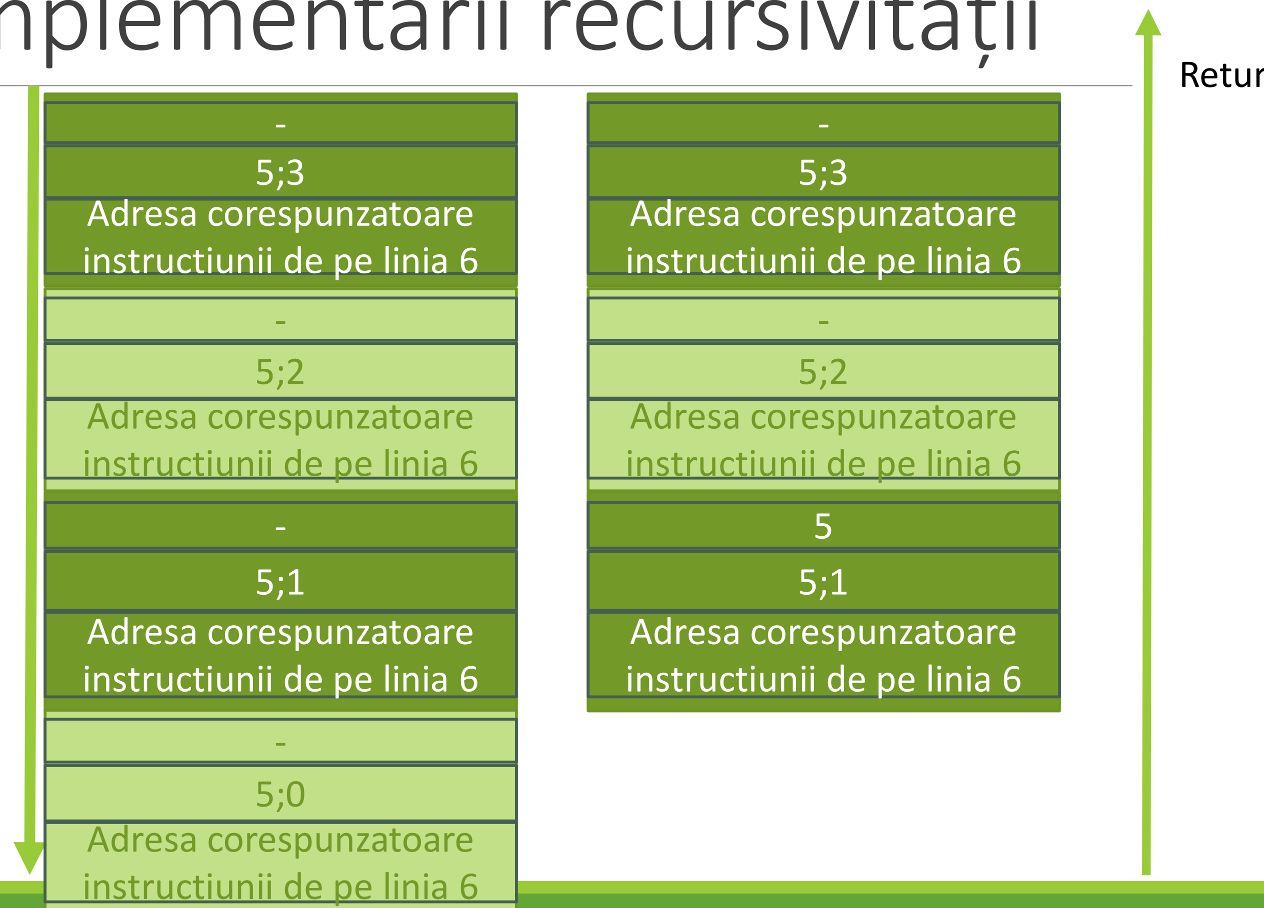


Exemplu apel pwr(5,3)

Mecanismul implementării recursivității

```
1  double pwr(double x, unsigned n)
2  {
3          if (n == 0)
4                  return 1.0;
5          return x*pwr(x, n - 1);
6  }
7
```

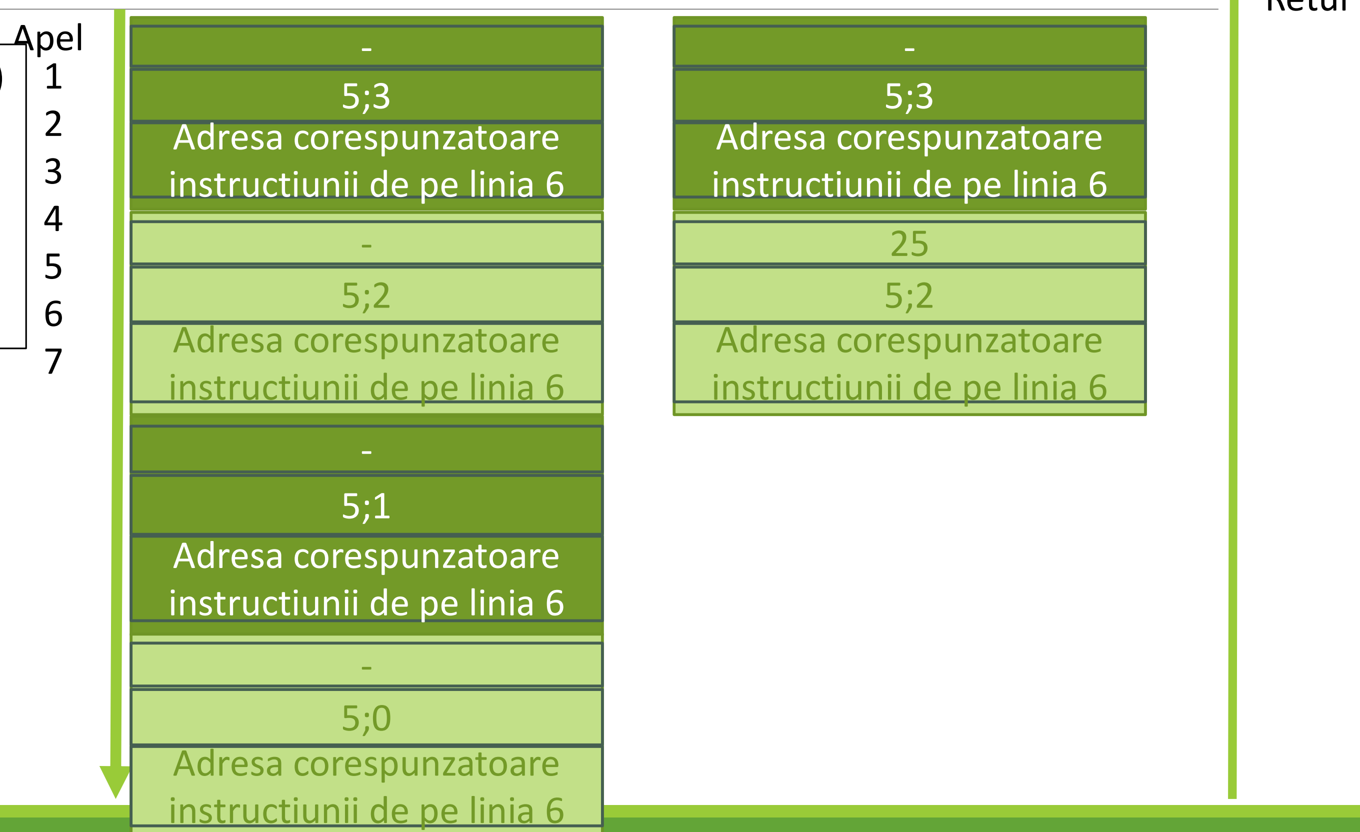
Exemplu apel pwr(5,3)



Mecanismul implementării recursivității

```
1 double pwr(double x, unsigned n)
2 {
3         if (n == 0)
4                 return 1.0;
5         return x*pwr(x, n - 1);
6 }
7
```

Exemplu apel pwr(5,3)



Mecanismul implementării recursivității

Apel

```
1 double pwr(double x, unsigned n)
2 {
3         if (n == 0)
4                 return 1.0;
5         return x*pwr(x, n - 1);
6 }
7
```

Exemplu apel pwr(5,3)

-
5;3
Adresa corespunzatoare instructiunii de pe linia 6
-
5;2
Adresa corespunzatoare instructiunii de pe linia 6
-
5;1
Adresa corespunzatoare instructiunii de pe linia 6
-
5;0
Adresa corespunzatoare instructiunii de pe linia 6

Retur

125
5;3
Adresa corespunzatoare instructiunii de pe linia 6

Mecanismul implementării recursivității

Ne amintim:

Ce se întâmplă în cazul apelului unei funcții?

Ce se salvează pe stiva sistem în cazul unui apel de funcție?

O funcție are parametri formali, care trebuie inițializați cu valorile transmise de parametri actuali. În plus sistemul trebuie să salveze adresa de revenire după apel. Spunem că atunci când facem un apel de funcție pe stivă se salvează ”starea sistemului”. Din ce constă aceasta?

Pe stivă se salvează valorile parametrilor, valorile variabilelor locale, valoarea returnată și adresa de retur. Starea fiecărei funcții, inclusiv main, este salvată pe stivă într-un spațiu dedicat

Mecanismul implementării recursivității

Exemplu de înregistrări pe stiva sistem pentru un apel din funcția main a funcției f1, care la rândul ei apelează funcția f2.

De retinut:
Pentru fiecare apel recursiv al unei funcții se creaza copii locale ale parametrilor transmiți prin valoare si ale variabilelor locale, ceea ce poate duce la risipa de memorie.

	Stiva sistem
	Parametri locali si variabile
Stiva functiei f2()	Referință la stiva funcției apelante
	Adresa de retur
	Valoarea returnată
	Parametri locali si variabile
Stiva functiei f1()	Referință la stiva funcției apelante
	Adresa de retur
	Valoarea returnată
Stiva functiei main()	

Tipuri de apeluri recursive

Recursivitatea se poate implementa în mai multe moduri:

- Dacă o funcție F se apelează pe ea însăşi, se spune că este **direct recursivă**.
- Dacă F apelează o altă funcție G care la rândul ei conţine un apel (direct sau indirect) la F, se spune că F este **recursivă indirect**.

Exemplu de recursivitate indirectă:

Se consideră aproximarea $\sin(x) \cong x - \frac{x^3}{6}$ și următoarele formule:

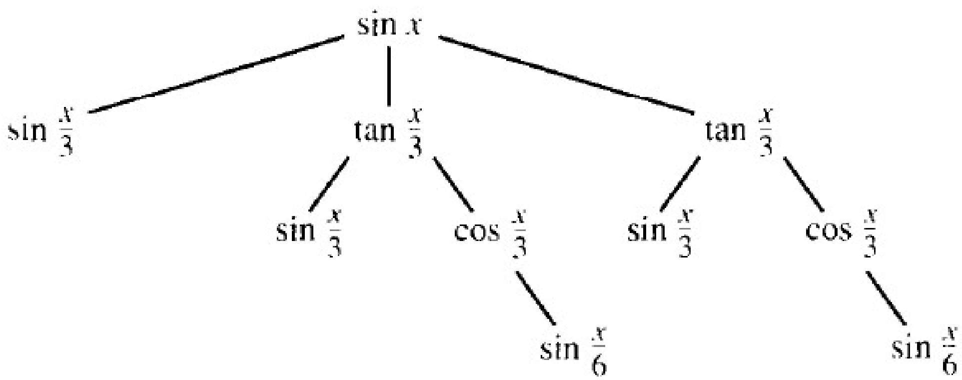
$$\sin(x) = \sin(\frac{x}{3}) \frac{(3-tan^2\left(\frac{x}{3}\right))}{(1+tan^2\left(\frac{x}{3}\right))}$$

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

$$\cos(x) = 1 - \sin(\frac{x}{2})$$

Tipuri de apeluri recursive

Pentru functiile anterioare un exemplu de graf de apeluri este



Tipuri de apeluri recursive

Recursivitate imbricată

Un exemplu mai complicat de recursivitate se găsește în definițiile în care o funcție nu este definită doar în termeni proprii, dar apare și ca parametru la ea însăși

Exemplu:

Funcția Ackermann

$$A(n,m) = \begin{cases} m+1, dacă\ n = 0 \\ A(n-1,1), dacă\ n > 0, m = 0 \\ A\big(n-1, A(n, m-1)\big), altfel \end{cases}$$

Pentru astfel de funcții o implementare nerecursivă este extrem de dificilă.

Tipuri de apeluri recursive

Recursivitate excesivă

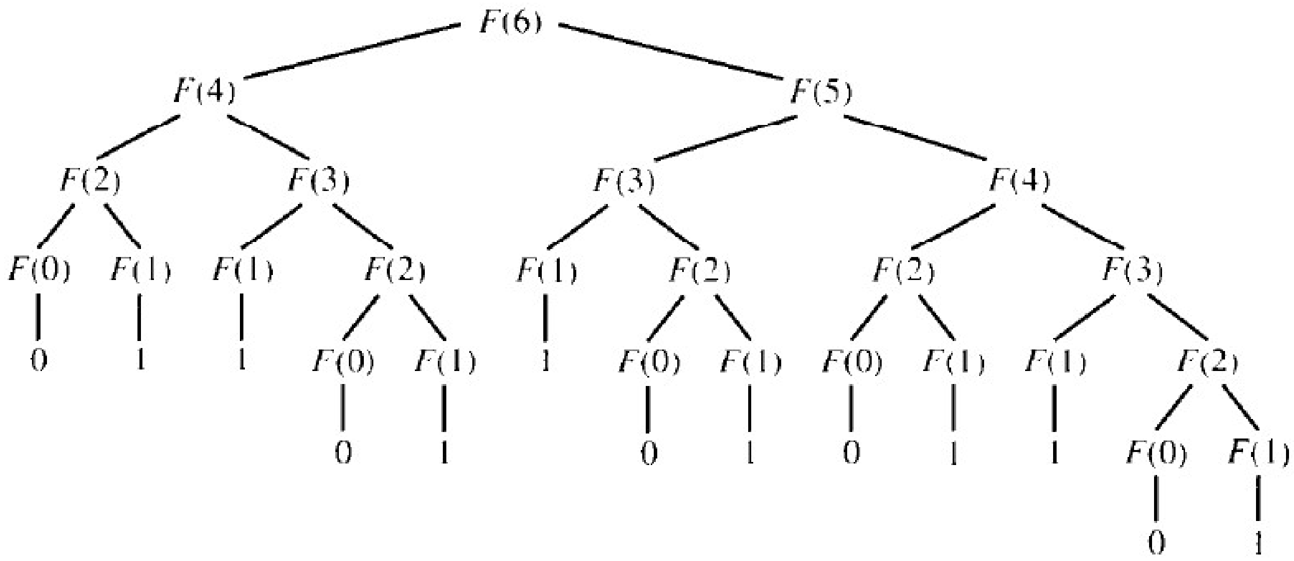
Din motive de simplitate și claritate, unele funcții sunt implementate în mod recursiv, deși creșterea numărului de apeluri în acest caz duce la o performanță extrem de scăzută atât din punct de vedere al timpului cât și al spațiului de memorie utilizat.

Un astfel de exemplu este șirul lui Fibonacci, definit pe numere naturale

$$Fib(n) = \begin{cases} n, dacă\ n < 2 \\ Fib(n-2) + Fib(n-1), altfel \end{cases}$$

```
unsigned int Fib(unsigned int n)
{
 if (n < 2)
    return n;
 else
    return Fib(n - 2) + Fib(n - 1);
}
```

Tipuri de apeluri recursive



Tipuri de apeluri recursive

Apel recusiv la sfârșit (tail recursion)

- Acest tip de recursivitate este caracterizat printr-un singur apel recursiv la sfârșitul funcției
- Cu alte cuvinte, după apelul recursiv nu urmează alte instrucțiuni

Exemplu:

```
double pwr(double x, unsigned n)
{if (n == 0)
      return 1.0;
 return x*pwr(x, n - 1);
}
```

În acest caz funcția recursivă **poate fi înlocuită cu o buclă**.

```
double pwr(double x, unsigned n)
{ double p=1.0;
 while (n != 0)
 {    p=p*x; n--;
 }
 return p;
}
```

Tipuri de apeluri recursive

```
double pwr(double x, unsigned n)
{if (n == 0)
        return 1.0;
 return x*pwr(x, n - 1);
}
```

Pasul 1: T(1) = c, caz de bază

Pasul 2: T(n-1) -> T(n),(daca T(n-1) =(n-1)c => T(n)=nc)

T(1)=c, T(2)=c+c=2c, …T(k)=kc =>

T(n-1) = (n-1)c

T(n) = T(n-1)+c, (din definitie)

T(n) = (n-1)c + c = n·c => O(n)

Tipuri de apeluri recursive

Caz general

Teoreme:

1. În cazul în care T(n) poate fi definit ca având proprietatea

$$T(n) = \begin{cases} c, daca\ n \leq 1 \\ aT(n-b) + f(n), daca\ n > 2 \end{cases}, pentru\ c, a > 0, b \geq 0, k \geq 0$$

Daca f(n) este $O(n^k), atunci$:

$$T(n) = \begin{cases} O(n^k), daca\ a < 1 \\ O(n^{k+1}), daca\ a = 1 \\ O(n^k a^{\frac{n}{b}}), daca\ a > 1 \end{cases}$$

2. În cazul în care T(n) = T(α n) + T((1 – α)n) + βn, unde 0 < α < 1 si β > 0 sunt constante =>T(n)= O(n log n)

Tipuri de apeluri recursive

Apel recursiv în interiorul funcției (nontail recursion)

Datorită stivei sistem, recursivitatea se poate folosi și pentru a prelucra o serie de date în ordine inversă.

Exemplu:

```
void citeste_c_rec(char car)
{
      if (car == '\n')
             return;
      else
      {
      scanf("%c",&car);
      citeste_c_rec(car);
      }
      printf("%c",car);
}
```

Pentru aceste cazuri, eliminarea recursivității constă de obicei în implementarea și utilizarea explicită a unei stive

Tehnici de programare bazate pe recursivitate

Cazul general de utilizare a recursivităţii

• Algoritmii recursivi sunt potriviţi a fi utilizaţi atunci când problema care trebuie rezolvată sau datele care trebuiesc prelucrate sunt definite în termeni recursivi.

• Cu toate acestea, un astfel de mod de definire nu justifică întotdeauna faptul că utilizarea unui algoritm recursiv reprezintă cea mai bună alegere.

• Mai mult, utilizarea recursivităţii în anumite situaţii nepotrivite, coroborată cu regia relativ ridicată a implementării şi execuţiei unor astfel de algoritmi, a generat în timp un curent de opinie potrivnic destul de vehement.

• Cu toate acestea recursivitatea rămâne o tehnică de programare fundamentală cu un domeniu de aplicabilitate foarte bine delimitat.

• În continuare se prezintă un exemplu de construcţie a unui program recursiv pornind de la modelul generic recursiv

$$P \equiv \textbf{IF}\ c\ \textbf{THEN}\ P[S_i,P]$$

Tehnici de programare bazate pe recursivitate

Acest model este foarte potrivit în cazurile în care se cere calculul unor valori care se definesc cu ajutorul unor **relaţii recurente**.

• Un exemplu clasic în acest sens îl reprezintă **numerele factoriale** $f_n=n!$

După cum s-a precizat, algoritmii recursivi se recomandă a fi utilizaţi cu precădere pentru implementarea unor probleme care se pot defini în **manieră recursivă**.

Ne reamintim de formulele de recurență din capitolul 3 al acestui curs.

• Cu toate acestea, **recursivitatea** poate fi utilizată şi în cazul unor probleme **de natură nerecursivă**.

Tehnici de programare bazate pe recursivitate

Ne amintim exemplele din capitolul 3

```
//Varianta 1
void printAsc(int start, int end)
{
   if (start == end)
         printf("%d ", start);  //cazul de baza
   else
   {
         printAsc(start, end - 1);   //ipoteza/apelul recursiv
         printf("%d ", end);
   }
}
```

Tehnici de programare bazate pe recursivitate

Ne amintim exemplele din capitolul 3

```
//Varianta 2
void printAsc(int start, int end)
{
   if (start == end)
         printf("%d ", end);  //cazul de baza
   else
   {
         printf("%d ", start);
         printAsc(start+1, end);   //ipoteza/apelul recursiv
   }
}
```

Tehnici de programare bazate pe recursivitate

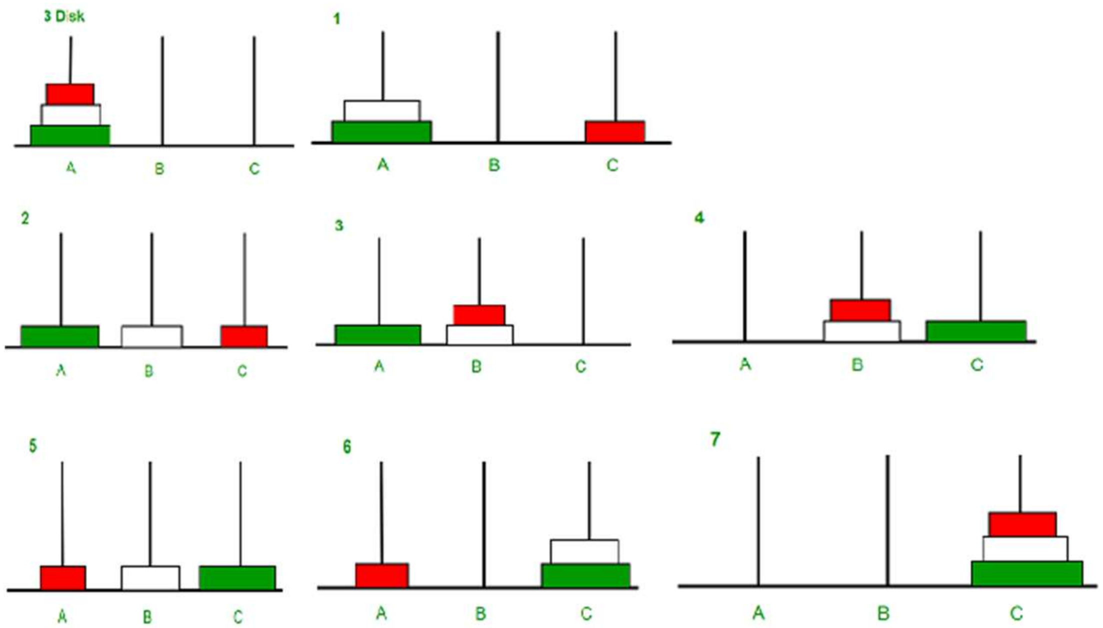
Algoritmi de reducere – Turnurile din Hanoi

Descriere problemă:

Fie 3 turnuri, si un număr de discuri pe primul turn, de mărimi diferite, așezate în ordine descrescătoare, de la cel mai mare ocupând poziția cea mai de jos la cel mai mic, ocupând poziția cea mai de sus. Se cere să se mute toate discurile pe ultimul turn respectând următoarele reguli:

- doar un disc este mutat la un moment dat

- niciun disc nu poate fi poziționat pe un disc mai mic decât el



Tehnici de programare bazate pe recursivitate

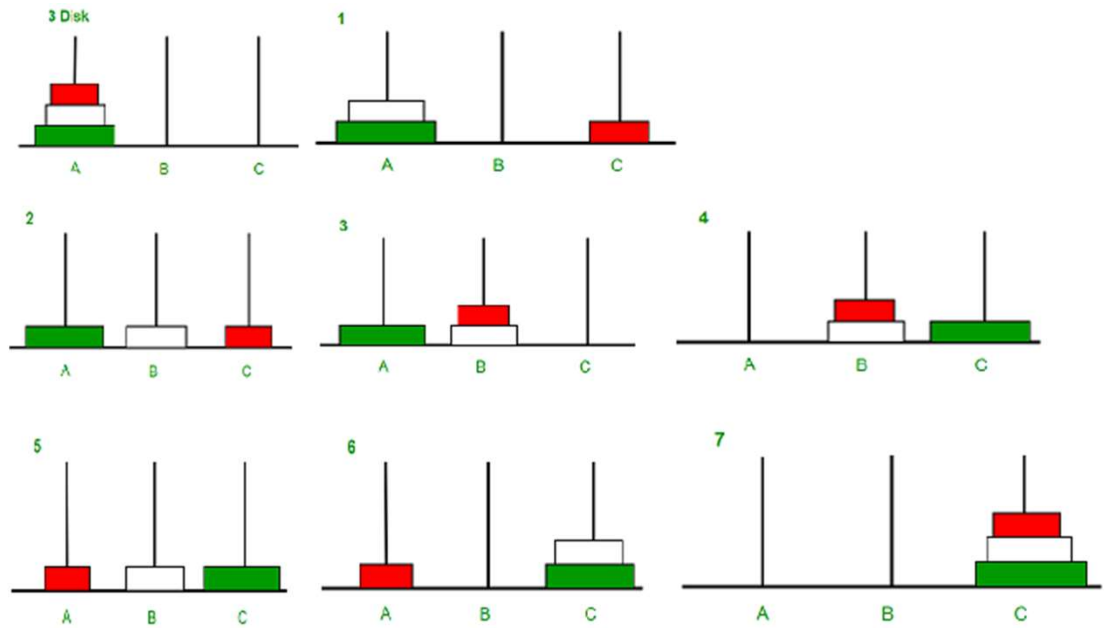
Algoritm

Se mută, în mod recursiv, primele n-1 discuri, de pe turnul sursă pe cel auxiliar

Se mută discul n, de pe turnul sursă pe cel destinație

Se mută cele n-1 discuri, în mod recursiv, de pe turnul auxiliar pe turnul destinație

Observație: Acest ultim pas poate fi văzut ca o altă instanță a aceleași probleme



Tehnici de programare bazate pe recursivitate

Algoritmi de reducere

• O categorie de algoritmi care se pretează pentru o **abordare recursivă** o reprezintă **algoritmii de reducere**.

• Aceşti algoritmi se bazează pe **reducerea** în manieră **recursivă** a **gradului de dificultate** al problemei, pas cu pas, până în momentul în care aceasta devine banală.

• În continuare se **revine** în acceeaşi manieră **recursivă** şi se **asamblează** soluţia integrală.

• În această categorie pot fi încadraţi algoritmul pentru **calculul factorialului** şi algoritmul pentru **rezolvarea problemei Turnurilor din Hanoi**.

• Se atrage atenţia că spre deosebire de **algoritmii cu revenire**, în acest caz **nu** se pune problema **revenirii** în caz de nereuşită, element care încadrează acest tip de algoritmi într-o categorie separată.

Divide et impera (Divide and Conquer)

Algoritmi de divizare

Tehnica divizării (”Divide and Conquer”)

• Una dintre **metodele fundamentale de proiectare a algoritmilor** se bazează pe **tehnica divizării** ("**devide and conquer**").

• **Principiul de bază** al acestei tehnici este următorul:

(1) Se **descompune** (divide) problema de rezolvat în mai multe subprobleme a căror rezolvare este mai simplă şi din soluţiile cărora se poate **asambla** simplu soluţia problemei iniţiale.

(2) Se repetă **recursiv** pasul (1) până când subproblemele devin banale iar soluţiile lor evidente.

Divide et impera (Divide and Conquer)

```
-------------------------------------------------------------
/* Tehnica divizării – soluţia recursivă - C */
void rezolva(x);
{
if (*x este divizibil în subprobleme)
{
*divide pe x în mai multe părţi: x1,x2,..,xk;
rezolva(x1);
rezolva(x2);
...
rezolva(xk);
*combină cele k soluţii parţiale într-o soluţie
pentru x
}/*if*/
else
*rezolvă pe x direct;
};/*rezolva*/
/*-------------------------------------------------------*/
```

Divide et impera (Divide and Conquer)

Un prim exemplu de **algoritm de divizare** îl constituie **metoda de sortare Quicksort** deja prezentată în capitolul de sortări avansate.

• În acest caz problema de rezolvat se divide de fiecare dată în **două subprobleme**, rezultând un **arbore binar de apeluri**.

• **Combinarea soluţiilor** parţiale **nu** este necesară deoarece scopul este atins prin modificările care se realizează chiar de către rezolvările parţiale

Alte exemple studiate în acest curs sunt: Shellsort, Mergesort și Cătutarea binară.

Divide et impera (Divide and Conquer)

Teorema de bază pentru recursivitatea de tip Divide et Impera (Divide and Conquer):

Pentru o problemă de tip recursiv, primul pas constă din a determina relația de recursivitate, așa cum s-a văzut în exemplele anterioare.

Dacă relația are următoarea formă generală:

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^k log^p n\right), unde\ a \geq 1, b > 1, k \geq 0, si\ p\ numar\ real, atunci:$$

$$1. daca\ a > b^k, atunci\ T(n) = \Theta\left(n^{log_b a}\right)$$

$$2. daca\ a = b^k, atunci$$

a. Daca p>-1, atunci $T(n) = \Theta\left(n^{log_b a} log^{p+1} n\right)$

b. Daca p=-1, atunci $T(n) = \Theta\left(n^{log_b a} \log log\, n\right)$

c. Daca p<-1, atunci $T(n) = \Theta\left(n^{log_b a}\right)$

Divide et impera (Divide and Conquer)

(Continuare) Dacă relația are următoarea formă general:

$T(n) = aT\left(\frac{n}{b}\right) + \Theta\left(n^k log^p n\right), unde\ a \geq 1, b > 1, k \geq 0, si\ p\ numar\ real, atunci$:

$3. Daca\ a < b^k$

a. Daca p ≥ 0, atunci $T(n) = \Theta\left(n^k log^p n\right)$

b. Daca p < 0, atunci $T(n) = \Theta\left(n^k\right)$

Exemple:

T(n) = 16T(n/4) + n => Solutie: T(n) = 16T (n/4) + n => T(n) = Θ(n^2) (Cazul 1)

T(n) = 4T (n/2) + n^2 => Solutie: T(n) = 4T (n/2) + n^2 => T (n) = Θ(n^2log n) (Cazul 2.a)

T(n) = 3T (n/2) + n^2 => Solutie: T(n) = 3T (n/2) + n^2 => T (n) =Θ(n^2) (Cazul 3.a)

Backtracking

Unul din subiectele de mare interes ale programării se referă la rezolvarea unor probleme cu caracter general.

- Ideea este de a concepe algoritmi generali pentru găsirea soluţiilor unor probleme specifice, care să nu se bazeze pe un set fix de reguli de calcul, ci pe încercări repetate şi reveniri în caz de nereuşită.
- Modalitatea comună de realizare a acestei tehnici constă în descompunerea obiectivului (taskului) în obiective parţiale (taskuri parţiale).
- De regulă această descompunere este exprimată în mod natural în termeni recursivi şi constă în explorarea unui număr finit de subtaskuri.
- În general, întregul proces poate fi privit ca un proces de încercare sau căutare care construieşte în mod gradat soluţia şi parcurge în acelaşi timp un arbore de subprobleme.

Backtracking

• Obţinerea unor **soluţii parţiale** sau **finale** care **nu** satisfac, provoacă **revenirea recursivă** în cadrul procesului de căutare şi **reluarea** acestuia până la obţinerea **soluţiei dorite**.

• Din acest motiv, astfel de algoritmi se numesc **algoritmi cu revenire** ("**backtracking algorithms**").

• În multe cazuri arborii de căutare cresc foarte rapid, de obicei **exponenţial**, iar efortul de căutare creşte în aceeaşi măsură.

• În mod obişnuit, arborii de căutare pot fi simplificaţi numai cu ajutorul **euristicilor**, simplificare care se reflectă de fapt în restrângerea volumului de calcul şi încadrarea sa în **limite acceptabile**.

Backtracking

- Metoda se aplica problemelor în care soluția se poate reprezenta sub forma unui vector $x=(x_1,x_2,\ldots x_n) \in S=S_1 \times S_2 \times\ldots\times S_n$, unde multimile S_i sunt finite, S numindu-se spațiul soluțiilor posibile.
- In particular, S_i sunt identice având același numar M de elemente. Pentru fiecare problemă concretă sunt date anumite relații între componentele vectorului x, numite condiții interne.
- Determinarea tuturor soluțiilor rezultat se poate face generând toate soluțiile posibile și verificând apoi care satisfac condițiile interne. Dar timpul de calcul ar fi foarte mare (dacă mulțimile S_i ar avea numai câte 2 elemente, timpul ar fi proportional cu 2^n).
- Metoda backtracking urmarește evitarea generării tuturor soluțiilor. Elementele vectorului x primesc valori pe rând, lui x_i i se atribuie valori, doar daca $x_1,x_2,\ldots,x_{i-1}$ au primit valori deja. Valorile atribuite trebuind să verifice condițiile de continuitate referitoare la $x_1,x_2,\ldots,x_i$. Doar apoi se trece la calculul lui x_{i+1}. În cazul neindeplinirii condițiilor de continuitate, se alege următoarea valoare posibilă pentru x_i. Dacă S_i a fost epuizat, se micsoreaza i, incercând o altă alegere pentru x_{i-1}.

Backtracking

```
backtracking( variabila i  de tip integer)
// {gaseste valoarea lui xi}
        variabila posibilitate de tip integer;
// {pentru toate valorile posibile ale lui xi}
{
   for posibilitate de la 0 la M-1
   {
      if acceptabila
      {
        inregisteaza_posibilitatea;
        if (i < n)
           backtracking(i+1);
        else afiseaza_solutia;
        sterge_inregistrarea;
      }
   }
}
```

Pe acestă metodă se bazează rezolvarea unor probleme clasice ca: "opt regine", a "relațiilor stabile", etc.

Backtracking

```
back2(variabila k de tip int);
{
        if solutie_completa
                afiseaza
        else
        {
          for j de la 0 la M-1 //parcurge pe rand posibilitatile
          {//alege posibilitatea posibilitate_k
           if acceptabila(posibilitate_k)
           {
                inregistreaza;
                back2(k+1);
                sterge_inregistrarea;
           }
          }
        }
}
```

Backtracking

Exemplu de implementare a permutarilor, care respecta structura anterioară:

```
int a[nmax];
int pos[nmax];  //tabloul de
posibilitati
void initializare(int n)
{//initializarea posibilitatilor
        int i;
        for (i = 0; i<n; i++)
              pos[i] = i + 1;
}
void afiseaza_solutia(int n)
{      int i;
        for (i = 0; i<n; i++)
           printf("%d ", a[i]);
        printf("\n"); }
```

```
int acceptabil(int pos, int k)
{       /*elementele din solutie
trebuie sa fie unice*/
        int i;
        for (i = 0; i < k;i++)
        if (a[i] == pos)
              return 0;
        return 1;
}
int solutie(int k, int n)
{
        return (k==n);  //solutia
cuprinde n elemente
}
```

Backtracking

```
void permuta(int k, int n) //k pasul, n -nr de elemente
{       int i,aux;
        if (solutie (k,n))  //solutie completa (avem n elemente)
              afiseaza_solutia(n);
        else
        {     for (i = 0; i <n; i++)  // parcurgem pe rand posibilitatile
              {
                     aux = pos[i];
                     if (acceptabil(aux, k)) {
                     //daca posibilitatea  e acceptabila
                            a[k ] = pos[i];
                            permuta(k + 1, n);   //back1(posibilitate_k+1)
                     }
//sterge_inregistrarea lipseste din aceasta implementare;
              }
        }
}    /*permuta*/
```

Backtracking

Exemplu:

Algoritm pentru determinarea tuturor drumurilor de ieşire dintr-un labirint

- Algoritmul care va fi prezentat în continuare presupune un labirint.
- Labirintul este descris cu ajutorul unui tablou bidimensional de caractere de dimensiuni n*n.
- Valoarea 1 reprezintă pereţii labirintului.
- Valoarea 0 (spaţiu liber) reprezintă culoarele.
- Punctul de start este dat, de un punct c, de coordonate x (coordonata pe orizontală) și y(coordonata pe verticală).

```
 void initializare() //int pos[] variabila globala
 {     /*initializarea posibilitatilor de deplasare Posibilitățile de
deplasare sunt Nord, Est, Sud, Vestn */
      pos[0].x = 0; // Nord
      pos[0].y = -1;// y scade spre Nord
      pos[1].x = 1; // Est - x creste spre Est
      pos[1].y = 0;
      pos[2].x = 0; // Sud
      pos[2].y = 1; // y creste spre Sud
      pos[3].x = -1;// Vest - x scade spre Vest
      pos[3].y = 0;
 }
```

Backtracking

Exemplul unei implementări a procedurii de backtracking pentru determinarea traseului de ieșire din labirint

```
void labirint(int k)  //k pasul, c coordonata curenta
{                 int i; coordonate aux;
      if (solutie(k, a[k - 1]))        //solutie completa
             afiseaza_solutia(k);
      else
      {
             for (i = 0; i < 4; i++)  // parcurgem pe rand posibilitatile
             {
                    aux.x = a[k - 1].x + pos[i].x;
                    aux.y = a[k - 1].y + pos[i].y;
                    if (acceptabil(aux)) { //daca posibilitatea  e acceptabila
                       a[k] = aux;
                       matrice[a[k].y][a[k].x] = 2; //marcheaza ca vizitat
                       labirint(k + 1);     // back1(posibilitate_k+1)
                       matrice[a[k].y][a[k].x] = 0; //sterge marcajul
                    }
             }
      }
}   /*labirint*/
```

Backtracking

Dacă considerăm linia și coloana din matrice, în loc de x și y, programul devine:

```
#define N 4
typedef struct coordonate
{
 int linie, coloana;
}coordonate;
coordonate pos[4];
coordonate a[N * N];
int matrice[N][N] = { 1,1,1,1,
   1,0,1,1,
   1,0,0,0,
   1,0,1,0 };
```

Backtracking

```
void initializare() //int pos[] variabila globala
{/*initializarea posibilitatilor de deplasare Posibilitățile de
deplasare sunt Nord, Est, Sud, Vest*/
pos[0].coloana = 0; // Nord
pos[0].linie = -1;  // y scade spre Nord
pos[1].coloana = 1; // Est - x creste spre Est
pos[1].linie = 0;
pos[2].coloana = 0;  // Sud
pos[2].linie = 1;    // y creste spre Sud
pos[3].coloana = -1; // Vest - x scade spre Vest
pos[3].linie = 0;
}
```

Backtracking

```
int solutie(int k, coordonate c)
{
if (c.coloana == 0 || c.linie == 0 || c.coloana == N - 1 ||
c.linie == N - 1)
//daca am ajuns la margine
     return 1;
else
     return 0;
}
int acceptabil(coordonate c)
{
if (matrice[c.linie][c.coloana] == 0 && c.linie >= 0 &&
c.coloana >= 0 && c.linie < N && c.coloana < N)
//daca este liber si coordonata se afla in dimensiunile matricei
     return 1;
else return 0;
}
```

Backtracking

```
void labirint(int k)  //k pasul, c coordonata curenta
{
int i; coordonate aux;
if (solutie(k, a[k - 1]))        //solutie completa
      afiseaza_solutia(k);
else
{
      for (i = 0; i < 4; i++)  // parcurgem pe rand posibilitatile
{
aux.coloana = a[k - 1].coloana + pos[i].coloana;
aux.linie = a[k - 1].linie + pos[i].linie;
if (acceptabil(aux)) { //daca posibilitatea  e acceptabila
      a[k] = aux;
      matrice[a[k].linie][a[k].coloana] = 2; //marcheaza ca vizitat
      labirint(k + 1);     // back1(posibilitate_k+1)
      matrice[a[k].linie][a[k].coloana] = 0; //sterge marcajul ca vizitat
}}
}
}   /*labirint*/
```

Backtracking

```
int main(void)
{
initializare();
printf("dati linia initiala ");
scanf("%d", &a[0].linie);
printf("dati coloana initiala ");
scanf("%d", &a[0].coloana);
matrice[a[0].linie][a[0].coloana] = 2; //marcham ca vizitat
labirint(1);  //pasul 0 il consituie originea
return 0;
}
```

Backtracking

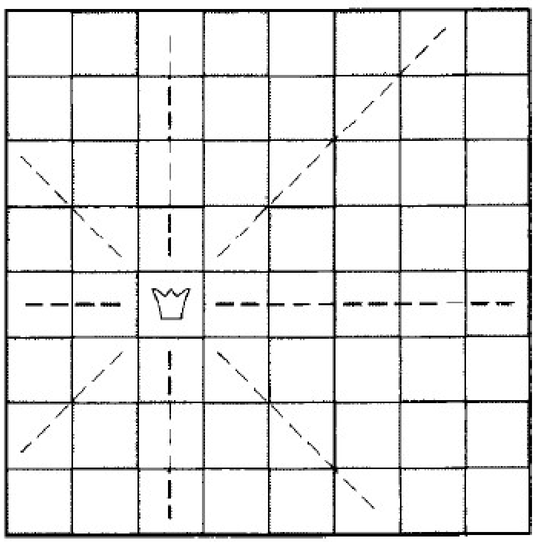
Exemplu de
execuție
Se marchează cu
valoarea 2
celulele vizitate

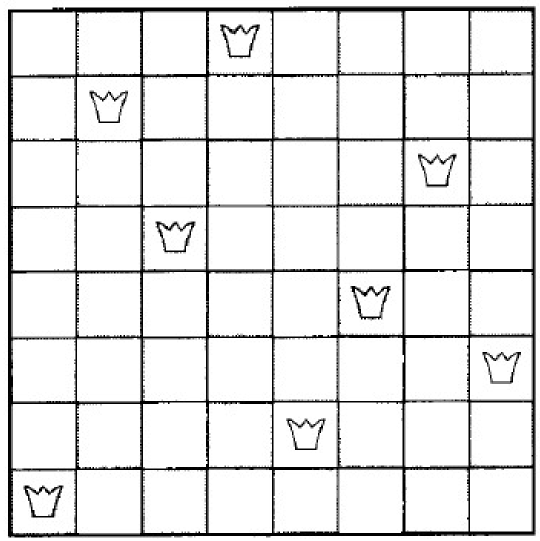
1111111111	1111111111	1111111111
1000100001	1222122221	1222100001
1110101101	1112121121	1112101101
1110101101	1112121121	1112101101
1110000101	1112220121	1112222101
1111110101	1111110121	1111112101
1100000101	1100000121	1122222101
1101111101	1101111121	1121111101
1100000000	1100000022	1122222222
1111111111	1111111111	1111111111

Backtracking

Un exemplu clasic de problemă care se rezolvă cu algoritmi cu revenire (backtracking) este problema celor 8 regine pe o tablă de șah.

Se cere să se așeze 8 regine pe o tablă de 8*8, astfel încât nicio regină să nu atace o alta (adică să nu fie pe aceeași linie, aceeași diagonală sau aceeași coloană 2 regine).





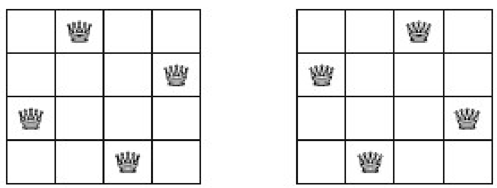
Backtracking

Algoritmul în pseudocod, pentru o tablă de dimensiune NxN este:

```
PuneRegina(rând)
    pentru fiecare coloană a aceluași rând
    dacă poziția coloană și cele două diagonale sunt libere
              pune regina pe poziția coloană
              dacă (rând < N)
                        puneRegina(rând+1)
              altfel
                    Succes
              înlătură regina de pe poziția coloană
```

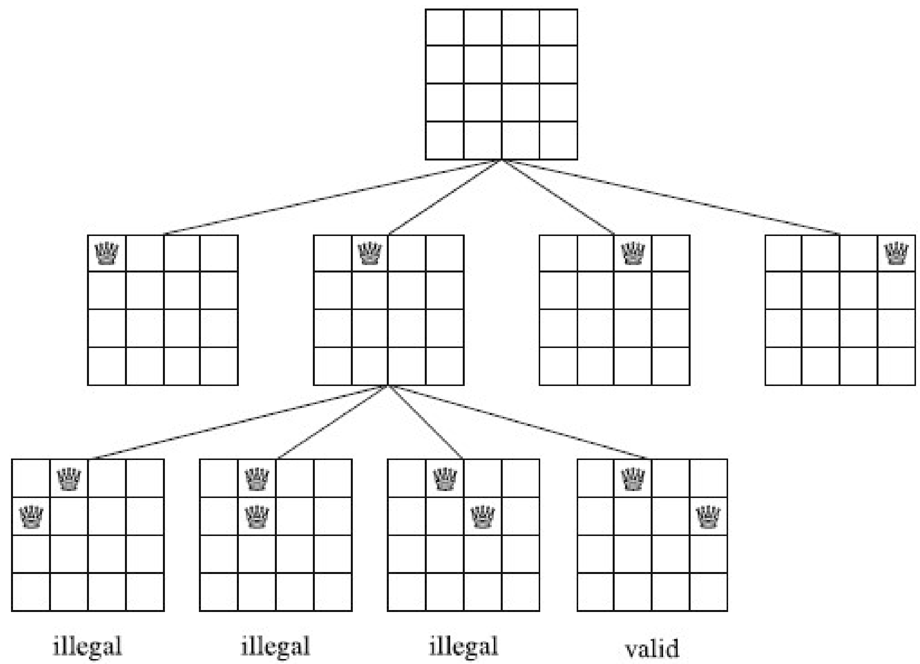
Backtracking

Astfel pentru N=4, avem următoarele posibilități



Pentru fiecare coloană a primului rând,
dacă poziția coloană este liberă
și nu este atacată de o altă regină
pune regina pe poziția coloană,
astfel doar soluția parțială din dreapta jos
este validă:

O soluție parțială:



Tehnici de eliminare a recursivității

De multe ori, soluția unei probleme poate fi elaborată mult mai ușor, mai clar și mai simplu de verificat, printr-un algoritm recursiv.

Dar, cel mai mare dezavantaj al recursivității este folosirea stivei, care se poate umple foarte rapid.

=> **Atenție:** Numeroase platforme de calcul (îndeosebi cele încorporate) **nu** suportă recursivitatea.

Alternativă:

Transformarea recursivității în iterații.

Orice program recursiv poate fi transformat în unul iterativ!

Tehnici de eliminare a recursivității

Recursivitate

- fiecare apel creează noi copii de parametri cu alte valori
- se execută din nou aceeași funcție cu alte date

Iterație

- control direct al iterațiilor
- se modifică prin atribuire valorile variabilelor
- se reia execuția acelorlași instrucțiuni cu alte valori

Tehnici de eliminare a recursivității

Recursivitate

Avantaje
- Ușor de implementat – nu este necesară o structură de date (se folosește stiva implicită)
- Cod lizibil și scurt

Dezavantaje
- Consum mare (chiar excesiv) de memorie
- Ieșirea din ciclu poate fi dificilă, dacă nu sunt bune condițiile
- Optimizarea memoriei presupune, în general, folosirea de variabile globale

Iterație

Avantaje
- Consum redus de memorie – necesar în cazul dispozitivelor cu memorie puțină
- Ieșirea din ciclu se poate face facil, folosind instrucțiunea break

Dezavantaje
- Cod lung
- Poate necesita structuri de date auxiliare

Tehnici de eliminare a recursivității

Recursivitatea reprezintă o **facilitate** excelentă de programare care contribuie la exprimarea simplă, concisă şi elegantă a algoritmilor de natură recursivă.

• Cu toate acestea, ori de câte ori problemele eficienţei şi performanţei se pun cu preponderenţă, se recomandă **evitarea utilizării recursivităţii.**

• De asemenea se recomandă evitarea recursivităţii ori de câte ori stă la dispoziţie o **rezolvare evidentă** bazată pe **iteraţie**.

• De fapt, implementarea recursivităţii pe echipamente nerecursive dovedeşte faptul că **orice** algoritm **recursiv** poate fi transformat într-unul **iterativ**.

Tehnici de eliminare a recursivităţii

În continuare se abordează teoretic **problema conversiei** unui **algoritm recursiv** într-unul **iterativ**.

• În abordarea acestei chestiuni se disting **două cazuri**:

(1) Cazul în care **apelul recursiv** al funcției apare la **sfârşitul** ei, drept ultimă instrucţiune a funcției ("**tail recursion**").

• În această situaţie, **recursivitatea** poate fi înlocuită cu o **buclă** simplă de **iteraţie**.

• Acest lucru este posibil, deoarece în acest caz, **revenirea** dintr-un apel încuibat presupune şi **terminarea** instanţei respective a funcției, motiv pentru care contextul apelului **nu** mai trebuie restaurat.

• Astfel dacă o funcție F(x) conţine ca şi **ultim pas** al său un apel la ea însăşi de forma F(y):

• Acest apel poate fi înlocuit cu o instrucţiune de atribuire x=y , urmată de reluarea (salt la începutul) codului lui F (observație y poate fi şi o expresie).

Tehnici de eliminare a recursivității

Exemplu:

În cazul recursivității cu rezultat parțial:

- Valoarea inițială a rezultatului rămâne aceeași
- Testul de oprire al iterației poate fi același cu cazul de bază al recursivității
- Valoarea rezultatului final este obținută tot prin acumulare

```
#include <stdio.h>
double pwr(double x, int n) {  //recursiva
        if (n == 0)
                return 1;
        return x*pwr(x, n - 1);
}
double pwr_iterativ(double x, int n){  //iterativa
        long r = 1; //cazul de baza, valoare initiala
        while (n>0)   //conditia de continuare
        {
                r = x*r;  //acumulare
                n = n-1;
        }
        return r;
}
int main() {
        int n, x;
        printf("Introduceti valoarea pentru x (reala):");
        scanf("%d", &x);
        printf("Introduceti valoarea pentru n (>=0):");
        scanf("%d", &n);
        printf("x la puterea n este %lf\n", pwr_iterativ(x, n));
        return 0;
}
```

Tehnici de eliminare a recursivității

```
#define nmax 100
typedef struct{ char corp_stiva[nmax]; int varf_stiva;} stiva;
// o stiva de dimensiune maxima nmax
stiva st;                    //variabila de tip stiva
void push(char car)       //pune in stiva un caracter
{
      st.corp_stiva[st.varf_stiva]=car; //se adauga un caracter
in varf
      st.varf_stiva++;                          //se marcheaza
adaugarea acestuia
}
char pop()         //ia din stiva un caracter
{
      st.varf_stiva--;
      return st.corp_stiva[st.varf_stiva];    //se ia din varf un
caracter
}
```

```
#include <stdio.h>
void citeste_c_rec(char car)
{
               if (car == '\n') return;
               else
               { scanf("%c",&car);
                 citeste_c_rec(car);
               }
               printf("%c",car);
}
void prel_car_nerec()
{      char car; st.varf_stiva=0;    //initializare stiva
  do
      {    scanf("%c",&car);   //citesc un caracter
            push(car);              //il pun in stiva
      }
      while (car!='\n');
      while (st.varf_stiva!=0)   //cat timp stiva nu e goala
      {
                 car=pop();        //iau din stiva un element
                 printf("%c",car);
      }
}
```

Tehnici de eliminare a recursivității

Pentru cazul precedent tehnica de eliminare a recursivității implică implementarea și utilizarea unei stive.

Astfel apelul recursiv de înlocuiește cu o buclă în care în locul apelului recursiv avem o depunere pe stivă a datelor utile și în locul revenirii din apel avem o scoatere de pe stivă a acestora.

Astfel push(car) se execută după citire (acolo unde aveam apelul recursiv în varianta recursivă), iar car=pop() se execută înainte de afișare.

Structuri de date recursive

Structuri de date recursive

• În cadrul paragrafelor acestui capitol, până în prezent, recursivitatea a fost prezentată ca o proprietate specifică algoritmilor, implementată în forma unor proceduri sau funcţii care se apelează pe ele însele.

• În cadrul acestui paragraf se va prezenta extinderea acestei proprietăţi şi asupra tipurilor de date în forma aşa-numitelor "structuri de date recursive".

• Prin analogie cu algoritmii, prin structură de date recursivă se înţelege o structură care are cel puţin o componentă de acelaşi tip ca şi structura însăşi.

• Şi în acest caz, definiţiile unor astfel de tipuri de date pot fi recursive în mod direct sau în mod indirect

• Un exemplu simplu de structură recursivă îl reprezintă lista înlănţuită a cărei definire formală, poate fi: o listă poate fi fie vidă, fie formată dintr-o altă listă la care se adaugă un nod.

Structuri de date recursive

Un alt exemplu de **structură recursivă** îl reprezintă **expresiile aritmetice**.

• În acest caz **recursivitatea** rezultă din faptul că orice **expresie aritmetică** poate conţine un **operand** care este la rândul său, **o expresie aritmetică** închisă între paranteze.

• În exemplul care urmează se va considera cazul simplu în care prin expresie aritmetică se înţelege fie:

• Un operand simplu care este reprezentat ca un identificator format dintr-o singură literă.

• O expresie aritmetică (operand) urmată de un operator, urmat de o expresie aritmetică (operand).

Concluzii

Recursivitatea, ca orice altă tehnică de programare, trebuie folosită cu chibzuință.

Nu există regulă generală pentru când ar trebui să fie folosită și când nu.

De obicei recursivitatea nu este la fel de eficientă ca versiunea iterativă, dar aduce o mai bună claritate a codului și o simplificare a acestuia.

Chiar dacă orice funcție recursivă poate fi transformtată într-una iterativă, transformarea nu este întotdeauna trivială. În particular, aceasta poate implica manupularea unor stive.

În sistemele timp real se recomandă a nu se folosi niciodată recursivitatea, din motive de timp de răspuns și determinism.

Pentru determinarea complexității se caută formulele de recurență fie pornind de la teoreme cunoscute, fie pe baza cazurilor mai simple, a căror corectitudine trebuie apoi demonstrate folosind inducția matematică.

Pentru demonstrarea complexității se folosește în general tehnica inducției matematice.

Exerciții

Ex1. Scrieți o funcție recursivă care numără nodurile dintr-o listă simplu înlănțuită
Ex2. Să se scrie un program pentru formatarea unui program C. Programul va ține cont de indentarea corectă a codului, de numărul de spații dintre elemente și de situațiile când ar trebui să se continue pe rând nou. Programul primește ca parametru de intrare un fișier cod .c, îl procesează și furnizează un nou fisier .c formatat
De exemplu codul următor

```
if ( n==1 ) { n = 2 * m;
if ( m < 10 )
f( n, m-1 ); else f( n,m-2 );} else n = 3 * m;
va deveni:
if ( n == 1)
{
     n = 2 * m;
     if ( m < 10 )
          f ( n, m-1);
     else f (n, m-2);
}
else n = 3 * m;
```

Exerciții

Ex3. Să se scrie un program care să determine lungimea maximă de celule cu valoarea 1 interconectate (pe orizontală sau verticală) într-o matrice dată:
De exemplu pentru matricea de mai jos:
1 1 0 0 0
0 1 1 0 0
0 0 1 0 1
1 0 0 0 1
0 1 0 1 1
Răspunsul este 5.

Bibliografie selectivă

- Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.
- Shaffer, C. A. (2012). Data structures and algorithm analysis.
- Crețu, V. Structuri de date și algorimi, Editura Orizonturi Universitare Timișoara, 2011