Tehnici de programare - TP

Cursul 10 – Recursivitate

Ș.l. dr. ing. Cătălin Iapă

catalin.iapa@cs.upt.ro

Să ne amintim de la LSD: Recursivitatea

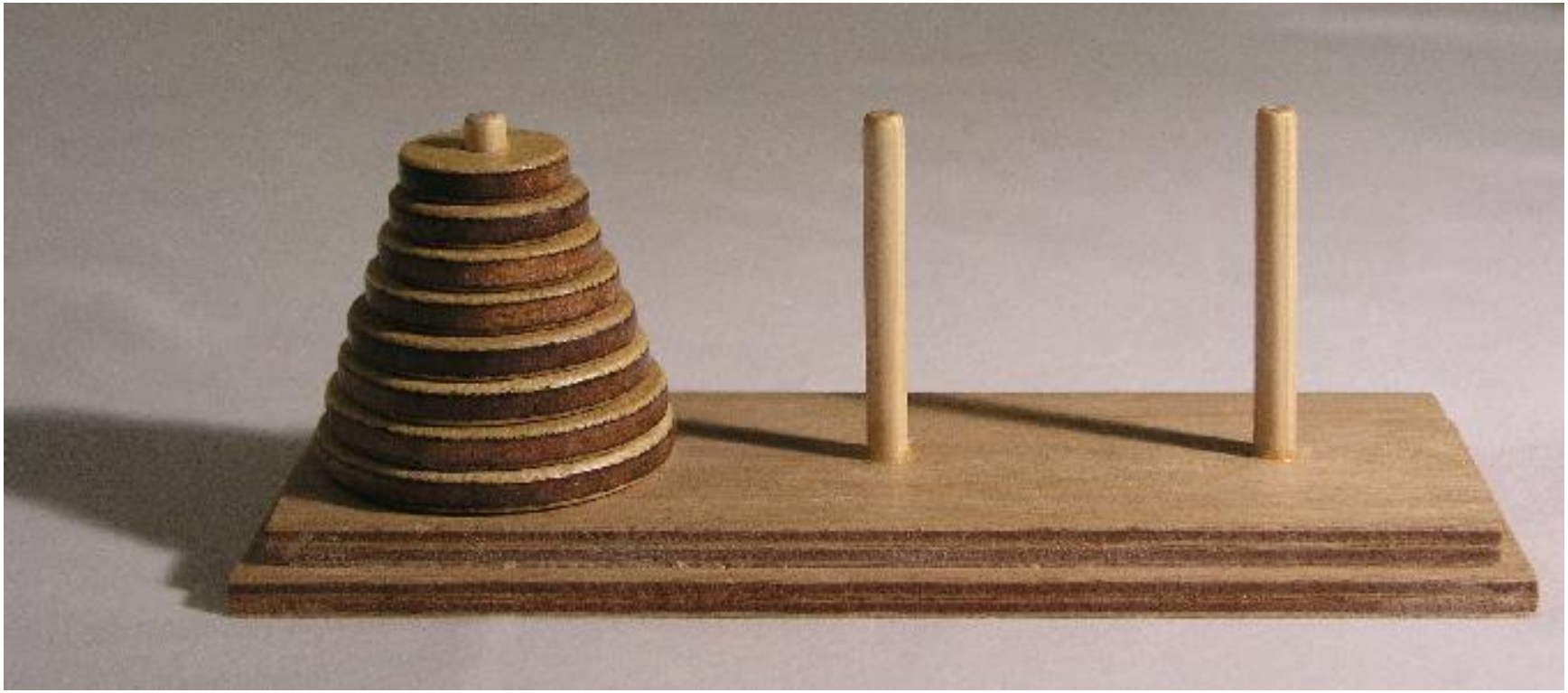
Să revenim la TP: Recursivitatea

Cum funcționează recursivitatea?

Recursiv vs. Iterativ

Exerciții

Turnurile din Hanoi



https://upload.wikimedia.org/wikipedia/commons/thumb/0/07/Tower_of_Hanoi.jpeg/450px-Tower_of_Hanoi.jpeg

Turnurile din Hanoi

Scopul jocului este acela de a muta întreaga stivă de pe o tijă pe alta, respectând următoarele reguli:

Turnurile din Hanoi

Scopul jocului este acela de a muta întreaga stivă de pe o tijă pe alta, respectând următoarele reguli:

- Doar un singur disc poate fi mutat, la un moment dat.
- Fiecare mutare constă în luarea celui mai de sus disc de pe o tija și glisarea lui pe o altă tijă, chiar și deasupra altor discuri care sunt deja prezente pe acea tijă.
- Un disc mai mare nu poate fi poziționat deasupra unui disc mai mic.

Turnurile din Hanoi

Trebuie să găsim numărul minim de mutări a întregii stive de pe un disc pe altul în funcție de numărul de discuri inițial

p(n)

p(1) = 1

p(2) = 3

P(3) = 7

Putem găsii o regulă generală?

Turnurile din Hanoi

Pasul 1 – mutăm n-1 discuri
Pasul 2 – mutăm discul cel mai mare (1 disc)
Pasul 3 – mutăm n-1 discuri

Turnurile din Hanoi

Pasul 1 – mutăm n-1 discuri
Pasul 2 – mutăm discul cel mai mare (1 disc)
Pasul 3 – mutăm n-1 discuri

$p(n) = p(n-1) + 1 + p(n-1)$
$p(n) = 2 * p(n-1) + 1$

Problema numărului de profesori

La facultate avem o provocare cu numărul de cadre didactice. Avem nevoie de mai mulți *profesori* pentru că numărul de studenți tot crește.

Problema numărului de profesori

La facultate avem o provocare cu numărul de cadre didactice. Avem nevoie de mai mulți *profesori* pentru că numărul de studenți tot crește.

Avem o regulă care duce la creșterea numărului de profesori:

- În fiecare an, un profesor trebuie să aducă/ să formeze *un nou profesor*
- Excepție face doar *primul an* pentru fiecare profesor, an în care nu trebuie să aducă/ să formeze un profesor

Problema numărului de profesori

La facultate avem o provocare cu numărul de cadre didactice. Avem nevoie de mai mulți *profesori* pentru că numărul de studenți tot crește.

Avem o regulă care duce la creșterea numărului de profesori:

- În fiecare an, un profesor trebuie să aducă/ să formeze *un nou profesor*
- Excepție face doar *primul an* pentru fiecare profesor, an în care nu trebuie să aducă/ să formeze un profesor

Câți profesori o să aibă facultatea după 8 ani dacă aplicăm aceste reguli și, pentru a simplifica calculul, în anul 1 pornim de la 1 profesor?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0

f(1) = 1

f(2) = ?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0

f(1) = 1

f(2) = ? ... 1

f(3) = ?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0

f(1) = 1

f(2) = ? ... 1

f(3) = ? ... 2

f(4) = ?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0
f(1) = 1
f(2) = ? ... 1
f(3) = ? ... 2
f(4) = ? ... 3
f(5) = ?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0
f(1) = 1
f(2) = ? ... 1
f(3) = ? ... 2
f(4) = ? ... 3
f(5) = ? ... 5
f(6) = ?

Problema numărului de profesori

Definim funcția:

f(n) = #numărul de profesori după anul n

f(0) = 0
f(1) = 1
f(2) = ? ... 1
f(3) = ? ... 2
f(4) = ? ... 3
f(5) = ? ... 5
f(6) = ? ... 8

Problema numărului de profesori

Anul 1: 1 profesor

Anul 2: 1 profesor - doar profesorul de anul trecut

Anul 3: 2 profesori – cel de anul trecut + 1 nou

Anul 4: 3 profesori – cei 2 de anul trecut + 1 nou

Anul 5: 5 profesori – cei 3 de anul trecut + 2 noi

Anul 6: 8 profesori – cei 5 de anul trecut + 5 noi

Anul 7: 13 profesori – cei 8 de anul trecut + 5 noi

Anul 8: 21 profesori – cei 13 de anul trecut + 8 noi

Problema numărului de profesori

f(n) = #numărul de profesori după anul n

Problema numărului de profesori

f(n) = #numărul de profesori după anul n

- f(n+1) = f(n) + f(n-1)

  Profesorii de anul trecut — profesori noi

Recunoașteți această recurență?

Problema numărului de profesori

f(n) = #numărul de profesori după anul n

- f(n+1) = f(n) + f(n-1)

  Profesorii de anul trecut — profesori noi

Recunoașteți această recurență?

E modul de construire a renumitului

Șir al lui Fibonacci

Problema numărului de profesori

Șirul lui Fibonacci e prima recurență despre care se știe că s-a studiat matematic, dintre toate recurențele studiate

Problema numărului de profesori

Șirul lui Fibonacci e prima recurență despre care se știe că s-a studiat matematic, dintre toate recurențele studiate

A fost publicat pentru prima dată în tratatul *"Liber abaci"* de *Leonardo Fibonacci din Pisa* în anul *1202*. Acest tratat cuprindea tot ce se știa despre matematică la acele vremuri și a influențat dezvoltarea matematicii în anii următori

Problema numărului de profesori

Șirul lui Fibonacci e prima recurență despre care se știe că s-a studiat matematic, dintre toate recurențele studiate

A fost publicat pentru prima dată în tratatul *"Liber abaci"* de *Leonardo Fibonacci din Pisa* în anul *1202*. Acest tratat cuprindea tot ce se știa despre matematică la acele vremuri și a influențat dezvoltarea matematicii în anii următori

A studiat cu ajutorul lui o problemă reală a acelor vremuri, creșterea populației de iepuri, pe care a exprimat-o astfel:

- în fiecare lună, o pereche de iepuri vor naște în medie alți 2 iepuri, cu excepția primei luni de viață

Recursivitate în informatică

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

Recursivitate în informatică

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

Recursivitatea e fundamentală în informatică:

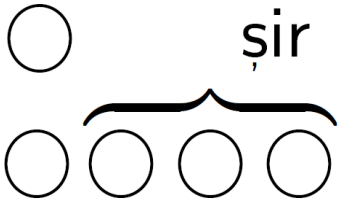
- dacă o problemă are soluție, *se poate rezolva recursiv*
- reducând problema la un caz mai simplu al *aceleiași probleme*

Înțelegând recursivitatea, putem rezolva orice problemă fezabilă

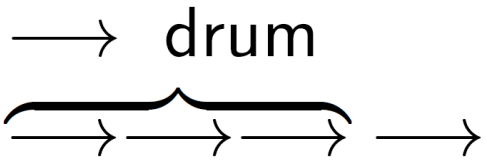
Recursivitate: exemple

Recursivitatea reduce o problemă la un caz mai simplu al aceleiași probleme

un *șir* e { un singur element / un element urmat de un *șir* }



un *drum* e { un pas / un *drum* urmat de un pas }



Șiruri recurente

Progresia aritmetică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0\text{)} \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

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

Progresia geometrică:

$$\begin{cases} x_0 = b & \text{(adică: } x_n = b \text{ pentru } n = 0\text{)} \\ x_n = x_{n-1} * \mathrm{r} & \text{pentru } n > 0 \end{cases}$$

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

Definițiile de mai sus nu calculeaă x_n *direct* ci *din aproape în aproape*, în funcție de x_{n-1}

Elementele unei definiții recursive

1. *Cazul de bază* este cel mai simplu caz pentru definiția dată, definit direct (termenul inițial dintr-un șir recurent: x_0)

Cazul de bază nu trebuie să lipsească

2. *Relația de recurență* propriu-zisă – definește noțiunea, folosind un caz mai simplu al aceleiași noțiuni

3. Demonstrația de *oprire a recursivității* după un număr finit de pași

Avantajele recursivității în programare

- Codul e scurt și ușor de urmărit, elegant, curat
- Problemele complexe pot fi împărțite în subprobleme mai simple și astfel mai ușor de rezolvat
- Generarea de șiruri se face mai simplu recursiv

Dezvantajele recursivității în programare

- E mai greu de urmărit pas cu pas logica din spatele unui cod scris recursiv
- Apelurile recursive repetate folosesc multă memorie
- Erorile care apar la funcțiile recursive sunt mai greu de corectat

Să ne amintim de la LSD: Recursivitatea

Să revenim la TP: Recursivitatea

Cum funcționează recursivitatea?

Recursiv vs. Iterativ

Exerciții

Recursivitate

Recursivitatea reprezintă proprietatea unor noțiuni de a se defini prin ele însele.

Exemple:

- factorialul unui număr: $N!=N\cdot(N-1)!$;
- ridicarea la putere: $a^n=a\cdot a^{n-1}$;
- termenul unei progresii aritmetice: $a_n = a_{n-1} + r$;
- șirul lui Fibonacci: $F(n) = F(n-1) + F(n-2)$;
- păpușa Matrioska

Recursivitate

Recursivitate

Conceptul de recursivitate oferă posibilitatea definirii unei infinităţi de obiecte printr-un număr finit de relaţii.

Recursivitatea a fost introdusă in programare in 1960, in limbajul Algol.

O funcţie este recursivă atunci când executarea ei implică cel puţin încă un apel către ea însăşi.

Recursivitate directă – apelul recursiv se face chiar din funcţia invocată.

Recursivitate indirectă (mutuală) – apelul recursiv se realizează prin intermediul mai multor funcţii care se apelează circular.

Recursivitate

factorialul unui număr: $N!=N\cdot(N-1)!$

Observăm că această regulă nu se aplică întotdeauna (la infinit). De exemplu, pentru 3! am obține:

$3! = 3\cdot2!$, $2!=2\cdot1!$, $1!=1\cdot0!$, $0!=0\cdot(-1)!$

De aici am putea deduce că $0!=0$ și înlocuind în relațiile de mai sus obținem că $3!=0$ și generic $n!=0$, pentru orice număr natural n. Bineînțeles, nu este corect. De fapt, formula recursivă pentru n! se aplică numai pentru $n>0$, iar prin definiție $0!=1$.

Recursivitate

Astfel, identificăm următoarea definiție pentru n!, acum completă:

$$n! = \begin{cases} 1, & dacă\ n = 0 \\ n \cdot (n-1)!\,, & dacă\ n > 0 \end{cases}$$

Recursivitate

În C, recursivitatea se realizează prin intermediul funcțiilor, care se pot *autoapela*.

O funcție trebuie definită iar apoi se poate apela.

Recursivitatea constă în faptul că în definiția unei funcții apare apelul ei însăși. Acest apel, care apare în însăși definiția funcției, se numește autoapel.

Primul apel, făcut în altă funcție, se numește apel principal.

Recursivitate

Funcția factorial implementată iterativ (nerecursiv)

```
int fact(int n){
    int p = 1, i;
    for(i = 1 ; i <= n ; i ++)
            p = p * i;
    return p;
}
```

Funcția factorial implementată recursiv

```
int fact(int n){
    if(n == 0)
        return 1;
    else
        return n * fact(n-1);
}
```

Să ne amintim de la LSD: Recursivitatea

Să revenim la TP: Recursivitatea

Cum funcționează recursivitatea?

Recursiv vs. Iterativ

Exerciții

Cum funcționează recursivitatea?

Ne amintim (de la PC) că toate variabilele locale din definiția unei funcții precum și valorile parametrilor formali se memorează la apel în memoria de tip STIVĂ (STACK).

Cum funcționează recursivitatea?

Ne amintim (de la PC) că toate variabilele locale din definiția unei funcții precum și valorile parametrilor formali se memorează la apel în memoria de tip STIVĂ (STACK).

Pentru fiecare apel al unei funcții se adaugă pe stivă o zonă de memorie în care se memorează variabilele locale și parametrii pentru apelul curent.

Cum funcționează recursivitatea?

Ne amintim (de la PC) că toate variabilele locale din definiția unei funcții precum și valorile parametrilor formali se memorează la apel în memoria de tip STIVĂ (STACK).

Pentru fiecare apel al unei funcții se adaugă pe stivă o zonă de memorie în care se memorează variabilele locale și parametrii pentru apelul curent.

Această zonă a stivei va exista până la finalul apelului, după care se va elibera.

Cum funcționează recursivitatea?

Ne amintim (de la PC) că toate variabilele locale din definiția unei funcții precum și valorile parametrilor formali se memorează la apel în memoria de tip STIVĂ (STACK).

Pentru fiecare apel al unei funcții se adaugă pe stivă o zonă de memorie în care se memorează variabilele locale și parametrii pentru apelul curent.

Această zonă a stivei va exista până la finalul apelului, după care se va elibera.

Dacă din apelul curent se face un alt apel, se adaugă pe stivă o nouă zonă de memorie, iar conținutul zonei anterioare este inaccesibil până la finalul acelui apel.

Cum funcționează recursivitatea?

Ne amintim (de la PC) că toate variabilele locale din definiția unei funcții precum și valorile parametrilor formali se memorează la apel în memoria de tip STIVĂ (STACK).

Pentru fiecare apel al unei funcții se adaugă pe stivă o zonă de memorie în care se memorează variabilele locale și parametrii pentru apelul curent.

Această zonă a stivei va exista până la finalul apelului, după care se va elibera.

Dacă din apelul curent se face un alt apel, se adaugă pe stivă o nouă zonă de memorie, iar conținutul zonei anterioare este inaccesibil până la finalul acelui apel.

Aceste operații se fac la fel și dacă al doilea apel este un autoapel al unei funcții recursive.

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

STIVA DE MEMORIE:

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

STIVA DE MEMORIE:

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

fact(2)

STIVA DE MEMORIE:

Zona 2: n = 2, f = 2*fact(1) = ?

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

fact(2)

fact(1)

STIVA DE MEMORIE:

Zona 3: n = 1, f = 1*fact(0) = ?

Zona 2: n = 2, f = 2*fact(1) = ?

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

fact(2)

fact(1)

fact(0)

STIVA DE MEMORIE:

Zona 4: n = 0, f = 1

Zona 3: n = 1, f = 1*fact(0) = ?

Zona 2: n = 2, f = 2*fact(1) = ?

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

fact(2)

fact(1)

fact(0)

STIVA DE MEMORIE:

Zona 4: n = 0, f = 1

Zona 3: n = 1, f = 1*fact(0) = 1*1=1

Zona 2: n = 2, f = 2*fact(1) = ?

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
   int f;
   if(n == 0)
      return 1;
   else
      f = fact(n - 1) * n;
   return f;
}
int main(){
   int x = fact(3);
   printf("fact(%d) = %d", 3, x);
   return 0;
}
```

main() int x

fact(3)

fact(2)

fact(1)

fact(0)

STIVA DE MEMORIE:

Zona 4: n = 0, f = 1

Zona 3: n = 1, f = 1*fact(0) = 1*1=1

Zona 2: n = 2, f = 2*fact(1) = 2*1=2

Zona 1: n = 3, f = 3*fact(2) = ?

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
    int f;
    if(n == 0)
        return 1;
    else
        f = fact(n - 1) * n;
    return f;
}
int main(){
    int x = fact(3);
    printf("fact(%d) = %d", 3, x);
    return 0;
}
```

main() int x

fact(3)

fact(2)

fact(1)

fact(0)

STIVA DE MEMORIE:

Zona 4: n = 0, f = 1

Zona 3: n = 1, f = 1*fact(0) = 1*1=1

Zona 2: n = 2, f = 2*fact(1) = 2*1=2

Zona 1: n = 3, f = 3*fact(2) = 3*2=6

Zona 0: x = ?

Cum funcționează recursivitatea?

```
int fact(int n){
    int f;
    if(n == 0)
        return 1;
    else
        f = fact(n - 1) * n;
    return f;
}
int main(){
    int x = fact(3);
    printf("fact(%d) = %d", 3, x);
    return 0;
}
```

main() printf: 6

fact(3)

fact(2)

fact(1)

fact(0)

STIVA DE MEMORIE:

Zona 4: n = 0, f = 1

Zona 3: n = 1, f = 1*fact(0) = 1*1=1

Zona 2: n = 2, f = 2*fact(1) = 2*1=2

Zona 1: n = 3, f = 3*fact(2) = 3*2=6

Zona 0: x = 6

Cum funcționează recursivitatea?

La fiecare apel al funcției fact avem variabilele n și f.

Ele însă sunt variabile diferite în fiecare apel de funcție, cu valori diferite, memorate în zone diferite ale stivei.

La un moment dat, se pot folosi numai variabilele din zona de memorie curentă, cea din "vârful" stivei.

Cum funcționează recursivitatea?

Este obligatoriu ca în definiția unei funcții recursive să apară cazul particular (cazul de bază, în care să nu aibă loc autoapelul). În caz contrar autoapelurile vor avea loc "la nesfârșit", iar în urma prea multor autoapeluri, stiva se va ocupa în totalitate și execuția programului se va întrerupe.

Cum funcționează recursivitatea?

Este obligatoriu ca în definiția unei funcții recursive să apară cazul particular (cazul de bază, în care să nu aibă loc autoapelul). În caz contrar autoapelurile vor avea loc "la nesfârșit", iar în urma prea multor autoapeluri, stiva se va ocupa în totalitate și execuția programului se va întrerupe.

De asemenea, este obligatoriu ca, pentru cazurile neelementare, valorile la autoapel a parametrilor să se apropie de valorile din cazul elementar. Altfel se va întâmpla situația descrisă mai sus: stiva se va ocupa în totalitate și programul se va opri, fără a determina rezultatele dorite.

Exercițiu cu funcții recursive

Exercițiu:

Se cere să se scrie un program care să citească un cuvant de la tastatură şi să-l afişeze atat normal cat şi in ordinea inversă a literelor. Cuvantul va fi urmat de spaţiu. Citirea se va efectua caracter cu caracter, intr-o singură variabilă de tip *char*.

INPUT: abcd(SPACE)(ENTER)

OUTPUT: abcd dcba

Exercițiu cu funcții recursive

```
void Invers( void ){
        char ch;
        ch = getchar( );
        printf("%c",ch);
        if (ch !=‘ ‘)
                Invers();
        printf("%c",ch);
        }

int main(void ){
        Invers( );
        return 0;
        }
```

Să ne amintim de la LSD: Recursivitatea

Să revenim la TP: Recursivitatea

Cum funcționează recursivitatea?

Recursiv vs. Iterativ

Exerciții

Comparație: recursivitate vs. iteraţie

Iteraţia

- execuţia repetată a unei secvenţe de instrucţiuni
- o nouă iteraţie se execută doar in urma evaluării unei condiţii (**la început sau sfârşit**)
- fiecare iteraţie se execută până la capăt şi apoi se trece, eventual, la o nouă iteraţie
- se recomandă atunci cand algoritmul de calcul este exprimat printr-o formulă iterativă

Recursivitatea

- execuţia repetată a unei intregi funcţii
- un nou apel recursiv se execută tot in urma evaluării unei condiţii (**pe parcurs**)
- funcţia recursivă se apelează din nou, inainte de terminarea apelului precedent
- se recomandă doar atunci cand problema este prin definiţie recursivă (recursivitatea consumă resurse in exces)

Comparație: recursivitate vs. iteraţie

Ce este mai indicat de utilizat, apelul recursiv sau calculul iterativ?

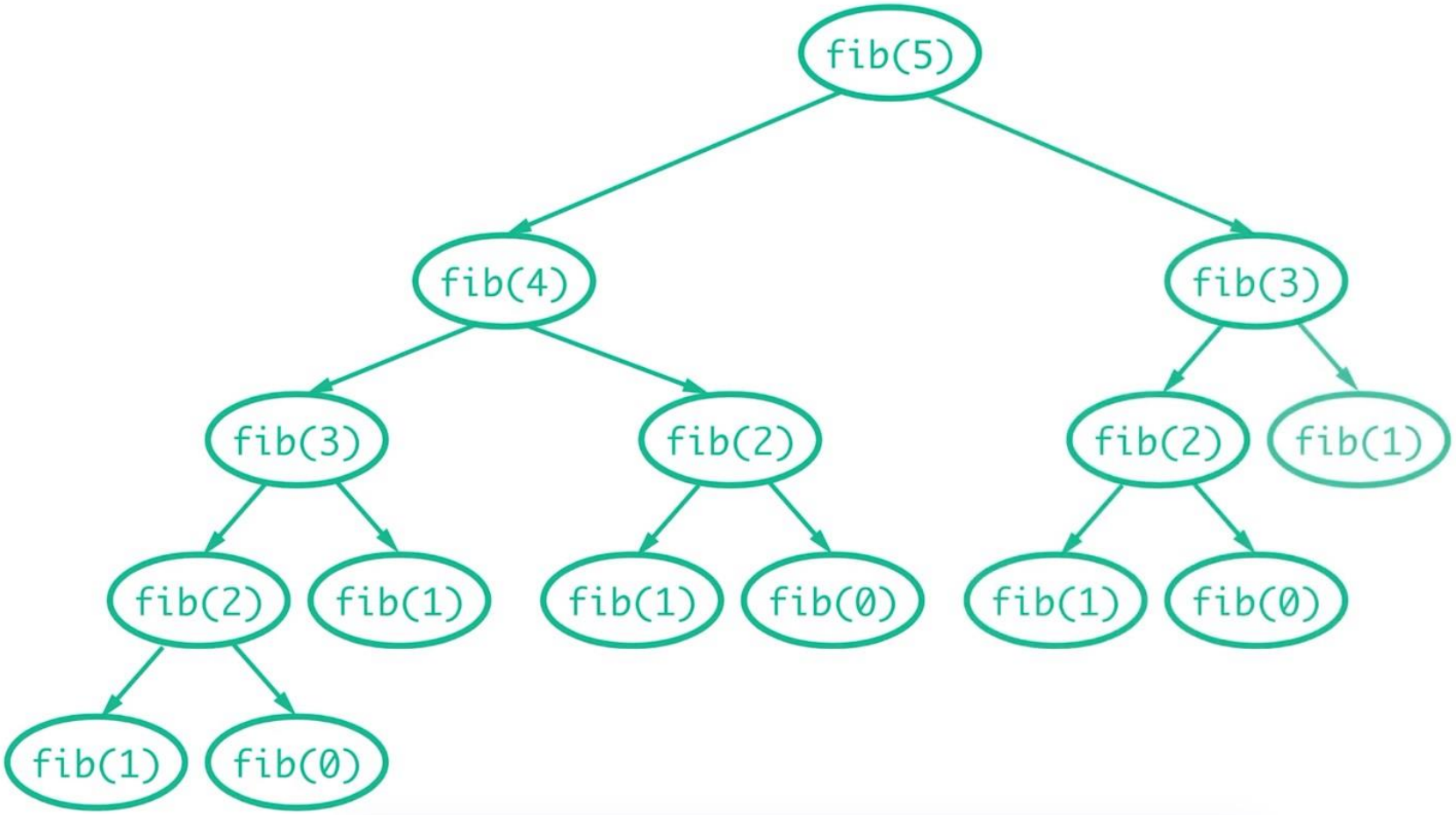
Deşi, aparent, cele două variante sunt echivalente, se impune totuşi următoarea remarcă generală: dacă algoritmului de calcul ii corespunde o formulă iterativă este de preferat să se folosească aceasta in locul apelului recursiv, intrucat viteza de prelucrarea este mai mare şi necesarul de memorie mai mic.

De exemplu, calculul recursiv al numerelor lui Fibonacci este complet ineficient intrucat numărul de apeluri creşte exponenţial odată cu *n:*

pentru *n* = 5 sunt necesare 15 apeluri

pentru n = 10 sunt necesare 177 apeluri

Comparație: recursivitate vs. iterație



Img: https://ro.khanacademy.org/

Comparație: recursivitate vs. iteraţie

In principiu, orice algoritm recursiv poate fi transformat intr-unul iterativ.

Această transformare se poate insă realiza mai uşor sau mai greu.

Sunt situaţii in care, in varianta iterativă, programatorul trebuie să gestioneze, in mod explicit, stiva apelurilor, salvandu-şi singur valorile variabilelor de la un ciclu la altul, ceea ce este dificil şi ingreunează mult inţelegerea algoritmului (ex. funcţia *Invers*).

Comparație: recursivitate vs. iteraţie

Pornind de la performanţele calculatoarelor actuale şi de la faptul că principala resursă a devenit timpul programatorului, recursivitatea are astăzi domeniile ei bine definite, in care se aplică cu succes.

In general se apreciază că algoritmii a căror natură este recursivă trebuie formulaţi ca funcţii recursive: prelucrarea structurilor de date definite recursiv (liste, arbori), descrierea proceselor şi fenomenelor in mod intrinsec recursive.

Comparație: recursivitate vs. iteraţie

S-au elaborat de asemenea metode recursive cu mare grad de generalitate in rezolvarea problemelor de programare (exemplu: *backtracking)* pe care programatorii experimentaţi le aplică cu multă uşurinţă, ca nişte scheme, aproape in mod automat.

Să ne amintim de la LSD: Recursivitatea

Să revenim la TP: Recursivitatea

Cum funcționează recursivitatea?

Recursiv vs. Iterativ

Exerciții

Căutarea binară (Binary Search)

Exercițiu:

Să se scrie un program care caută un element într-un şir *x=(x1, x2, ..., xn)* ordonat crescător. Se va folosi metoda căutării binare (Binary Search).

Căutarea binară (Binary Search)

Exercițiu:

Să se scrie un program care caută un element într-un şir *x=(x1, x2, ..., xn)* ordonat crescător. Se va folosi metoda căutării binare (Binary Search).

Rezolvare:

Se începe căutarea cu elementul din mijlocul şirului, continându-se apoi căutarea, dacă este nevoie, cu elementele din jumătatea stângă sau jumătatea dreaptă a şirului, aplicând același principiu, până nu mai avem elemente de căutat în șir.

Căutarea binară (Binary Search)

```
int CautareBinara(vector a, int elem, int inceput, int sfarsit) {
if (inceput<sfarsit) {
        int mijloc=(inceput+sfarsit)/2;
        if (elem==a[mijloc])
                return mijloc;
        else if (elem<a[mijloc])
                return CautareBinara(a,elem,inceput,mijloc-1);
              else
                return CautareBinara(a,elem,mijloc+1,sfarsit);
                }
else
        return -1;
}
```

Exerciții

Exercițiu:

Să se calculeze recursiv $\sqrt{x}$.

Rezolvare:

$\sqrt{x}$: a(0)=1, a(n+1) = (a(n)+x / a(n)) / 2

Cand s-a atins precizia dorită (|a(n+1)-a(n)|<ε), $\sqrt{x}$ este an.

Exerciții

```
double rad(double x, double an) {
if(x<0)
        return 0;
else
        if(fabs((x/an-an)/2)<0.0001 )
                return an;
        else
                return rad(x, (an+x/an)/2);
}
int main(void) {
printf("radical din 7 este %f \n", rad(7.0, 1.0));
return 0;
}
```

Exerciții

Care este rezultatul la apelul lui f(15, 2)?

```
1  void f (int n, int x){
2      if(x>n)
3          printf(""%d"",0);
4      else if(x%4<=1)
5          f(n,x+1);
6      else{
7          f(n,x+3);
8          printf(""%d"",1);
9      }
10 }
```

Exerciții

Care este rezultatul apelului lui f(3, 17)?

```
1  void f( int a, int b){
2      if(a<=b){
3          f(a+1,b-2);
4          printf(""%c"",'*');
5      }
6      else
7          printf(""%d"",b);
8  }
```

Vă mulțumesc!