



RECURSIVITATE

Șl. Dr. Ing. Șerban Radu

Definiții


- Recursivitate = proprietatea unei funcții de **a se putea apela pe ea însăși**
- O funcție este **direct recursivă** dacă conține în interiorul său un apel către ea însăși
- O funcție **P** este **indirect (mutual) recursivă** dacă conține în interiorul său un apel la o altă funcție **Q**, care la rândul ei apelează funcția **P**

Exemplu

```
# include <stdio.h>
# include <conio.h>
void recursiv(int i) {
    if (i < 10) {
        recursiv(i + 1);
        printf("%d ",i);
    }
}
int main(void) {
    recursiv(0);
    getch();
}
```

Rezultat

- 9 8 7 6 5 4 3 2 1 0
- Funcția **recursiv** este apelată pentru prima dată cu argumentul **0**
- Aceasta este prima activare a funcției **recursiv**
- Deoarece **0** este mai mic decât **10**, **recursiv** se apelează pe ea însăși, cu valoarea lui **i** (în acest caz **0**) plus **1**.




Aceasta este a doua activare a funcției **recursiv**, iar **i** este egal cu **1**

În continuare, **recursiv** este apelată din nou, folosind valoarea **2**

Procesul se repetă până când **recursiv** este apelată cu valoarea **10**

Cu această valoare, condiția din instrucțiunea **if** devine **false** și se revine în programul apelant

Deoarece revenirea se face la instrucțiunea de după apelare, se va executa **printf** din precedentă sa activare, adică se va afișa **9**




Se va reveni din nou în programul apelant la instrucțiunea ce urmează instrucțiunii de apelare, deci se va afișa cifra **8**

Procesul continuă până când fiecare din activările făcute revin în programul apelant și programul se termină

Observație – Nu au loc copieri multiple ale funcției recursive


Când este apelată o funcție, parametrii și datele locale sunt salvate pe **stivă**



Când o funcție este apelată recursiv, aceasta începe execuția cu un **nou set de parametri și variabile locale**, dar codurile care constituie funcția rămân aceleași. Fiecare funcție recursivă are o instrucțiune **if**, care controlează dacă funcția se va apela pe ea însăși din nou sau dacă va reveni în programul apelant. Fără o astfel de instrucțiune, o funcție recursivă va rula necontrolată, folosind toată memoria alocată stivei.

Exemplu – afișează 0 1 2 ... 8 9

```
# include <stdio.h>
# include <conio.h>
void recursiv(int i) {
    if (i < 10) {
        printf("%d ",i);
        recursiv(i + 1);
    }
}
int main(void) {
    recursiv(0);
    getch();
}
```





Deoarece apelarea funcției **printf** precede în acest caz apelarea recursivă a funcției **recursiv**, numerele sunt afișate în ordine crescătoare




Folosirea recursivității pentru a copia conținutul unui șir într-un alt șir

```
#include <stdio.h>
#include <conio.h>
void rcopy(char *s1, char *s2);
int main(void) {
    char str[80];
    rcopy(str, "Acesta este un exemplu");
    puts(str);
    getch();
}
```



```
/* Copiem s2 in s1 folosind recursivitatea */  
void rcopy(char *s1, char *s2) {  
    if (*s2) { /* daca nu este sfarsitul lui s2 */  
        *s1++ = *s2++;  
        rcopy(s1,s2);  
    }  
    else *s1 = '\0'; /* null incheie sirul */  
}
```



Programul atribuie caracterul punctat curent de **s2** unui caracter punctat de **s1** și apoi incrementează ambii pointeri

Acești pointeri sunt apoi folosiți pentru a apela recursiv funcția **rcopy**, până când **s2** punctează caracterul **null**, care încheie șirul

Recursivitate mutuală

- Apare când o funcție apelează o altă funcție, care apoi o apelează pe prima
- Program care afișează
- *****0 2 4 6 8 10 12 14 16 18 20
22 24 26 28 30



```
#include <stdio.h>
```

```
#include <conio.h>
```

```
void f2(int b);
```

```
void f1(int a);
```

```
int main(void) {
```

```
    f1(30);
```

```
    getch();
```

```
}
```

```
void f1(int a) {
```

```
    if (a) f2(a - 1);
```

```
    printf("%d ",a);
```

```
}
```

```
void f2(int b) {
```

```
    printf("*");
```

```
    if (b) f1(b - 1);
```


```
}
```



Rezultatul este produs de modul în care funcțiile **f1** și **f2** se apelează una pe cealaltă

De fiecare dată când **f1** este apelată, ea verifică dacă **a** este **0**


Dacă **a** nu este **0**, ea apelează funcția **f2** cu argumentul **a-1**



Funcția **f2** afișează mai întâi o steluță și apoi verifică dacă **b** este **0**

Dacă **b** nu este **0**, ea apelează funcția **f1** cu argumentul **b-1** și procesul se repetă

Dacă **b** este **0**, începe procesul de revenire în programul apelant, ceea ce face ca **f1** să afișeze numerele de la **0** la **30** din doi în doi




Program care afișează pe ecran un șir,
caracter cu caracter, folosind o funcție
recursivă

```
#include <stdio.h>
#include <conio.h>
void display(char *p);
int main(void) {
    display("acesta este un exemplu");
    getch();
}
void display(char *p) {
    if (*p) {
        printf("%c", *p);
        display(p+1);
    }
}
```

Descoperiți greșeala

```
#include <stdio.h>
void f();
int main(void) {
    f();
}
void f() {
    int i;
    printf("In functia f\n");
    /*apelarea functiei de 10 ori*/
    for(i = 0; i < 10; i++)
        f();
}
```



Funcția se va apela pe ea însăși până când va fi epuizată toată memoria, deoarece nu conține condiția care să verifice dacă funcția trebuie să se apeleze din nou



Program care folosește o funcție recursivă
pentru a afișa literele alfabetului

```
#include <stdio.h>
#include <conio.h>
void alfabet(char ch);
int main(void) {
    alfabet('A');
    getch();
}
void alfabet(char ch) {
    printf("%c ", ch);
    if (ch < 'Z') alfabet(ch + 1);
}
```

Program cu o funcție recursivă pentru a
calcula lungimea unui șir

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
#include <string.h>
```

```
int rstrlen(char *p);
```

```
int main(void) {
```

```
    printf("%d", rstrlen("recursivitate"));
```

```
    getch(); }
```

```
int rstrlen(char *p) {
```


```
    if (*p) { p++;
```

```
        return 1 + rstrlen(p);
```

```
    }
```


```
    else return 0;
```

```
}
```



Program cu o funcție recursivă pentru
a calcula numărul de zerouri dintr-un număr
natural

```
#include <stdio.h>
#include <conio.h>
int numarDeZerouri(int n);
int main(void) {
    int i, contor;
    printf("Introduceti un numar natual ");
    scanf("%d", &i);
    printf("Numarul %d are %d cifre de zero ", i,
        numarDeZerouri(i));
    getch();
}
```




```
int numarDeZerouri(int n) {  
    if (n == 0) return 1;  
    else if (n < 10) return 0;  
    else if (n % 10 == 0) return  
        (numarDeZerouri(n/10) + 1);  
    else return (numarDeZerouri(n/10));  
}
```


Algoritmul recursiv de căutare binară

- Scopul algoritmului este de a găsi o anumită valoare dintr-un șir de numere ordonate crescător (sau descrescător), folosind un număr minim de comparații
- Se împarte vectorul de numere în două părți, observând în care din cele două jumătăți este valoarea căutată, iar apoi se împarte acea jumătate în două ș.a.m.d.
- Vezi demonstrația `OrderedArray`

```
#include <stdio.h>
#include <conio.h>
int cautarebinara(int elemCautat, int left, int right);
int a[] = {3, 5, 7, 11, 19, 23, 27, 30, 36, 39, 42, 47, 50,
           75, 100};
int main(void) {
    int elemCautat, left, right;
    printf("Introduceti valoarea elementului cautat = ");
    scanf("%d", &elemCautat);
    left = 0;
    right = 14;
    printf("Indexul elementului cautat este %d",
           cautarebinara(elemCautat, left, right));
    getch();
}
```



```
int cautarebinara(int elemCautat, int left, int right)
{
    int mijloc = (left + right) / 2;
    if (a[mijloc] == elemCautat) return mijloc;
    else if (left > right) return -1;
    else {
        if (a[mijloc] < elemCautat) return
cautarebinara(elemCautat, mijloc + 1, right);
        else return cautarebinara(elemCautat,
left, mijloc - 1);
    }
}
```



Algoritmi “Divide et Impera”

- Căutarea binară recursivă este un exemplu de aplicare a metodei “Divide et Impera”
- Metoda presupune **împărțirea unei probleme mari** în două (sau mai multe) probleme mai mici și rezolvarea separată a fiecăreia dintre acestea



Algoritmi “Divide et Impera”

- Rezolvarea fiecăreia dintre subprobleme decurge similar: acestea se împart la rândul lor în subprobleme, care trebuie rezolvate separat
- Procesul continuă până când se întâlnește **situația de punct fix**, care se poate rezolva cu ușurință, fără o nouă divizare în subprobleme



Algoritmi “Divide et Impera”

- “Divide et Impera” este implementată printr-o funcție care conține două apeluri recursive, câte unul pentru fiecare jumătate a problemei
- În cazul căutării binare, deși există două astfel de apeluri, numai unul se execută efectiv.




Algoritmi “Divide et Impera”


- Sortarea prin interclasare execută efectiv ambele apeluri recursive, pentru a sorta ambele jumătăți ale unui vector
- Vezi demonstrația Towers

Problema turnurilor din Hanoi

- Soluția problemei se poate exprima recursiv
- Vrem să deplasăm n discuri de pe tija **A** pe tija **C**, folosind tija intermediară **B**
- Deplasăm $n-1$ discuri de pe **A** pe **B**
- Deplasăm **1** disc de pe **A** pe **C**
- Deplasăm $n-1$ discuri de pe **B** pe **C**



```
#include <stdio.h>
#include <conio.h>
int n;
void hanoi(int n, char a, char b, char c);
int main(void) {
    printf("Introduceti numarul de discuri n = ");
    scanf("%d", &n);
    hanoi(n, 'a', 'b', 'c');
    getch();
}
```



```
void hanoi(int n, char a, char b, char c) {  
    if (n == 1)  
        printf("Mut discul de pe tija %c pe tija %c\n", a, c);  
    else {  
        hanoi(n - 1, a, c, b);  
        printf("Mut discul de pe tija %c pe tija %c\n]", a, c);  
        hanoi(n - 1, b, a, c);  
    }  
}
```

Sortarea prin interclasare

- Sortarea prin interclasare **MergeSort** este un algoritm destul de simplu de implementat
- Din punct de vedere conceptual, este mai simplu decât algoritmul de sortare rapidă **Quicksort**
- Dezavantajul constă în faptul că are nevoie de un vector suplimentar în memorie, de dimensiune egală cu vectorul sortat

Interclasarea a doi vectori

- Ideea algoritmului este de a interclasa doi vectori deja sortați
- Interclasând doi vectori deja sortați **A** și **B**, rezultă un al treilea vector **C**, care conține toate elementele vectorilor **A** și **B**, așezate de asemenea în ordine crescătoare
- Examinăm mai întâi procesul de interclasare și vedem apoi cum poate fi utilizat pentru sortare

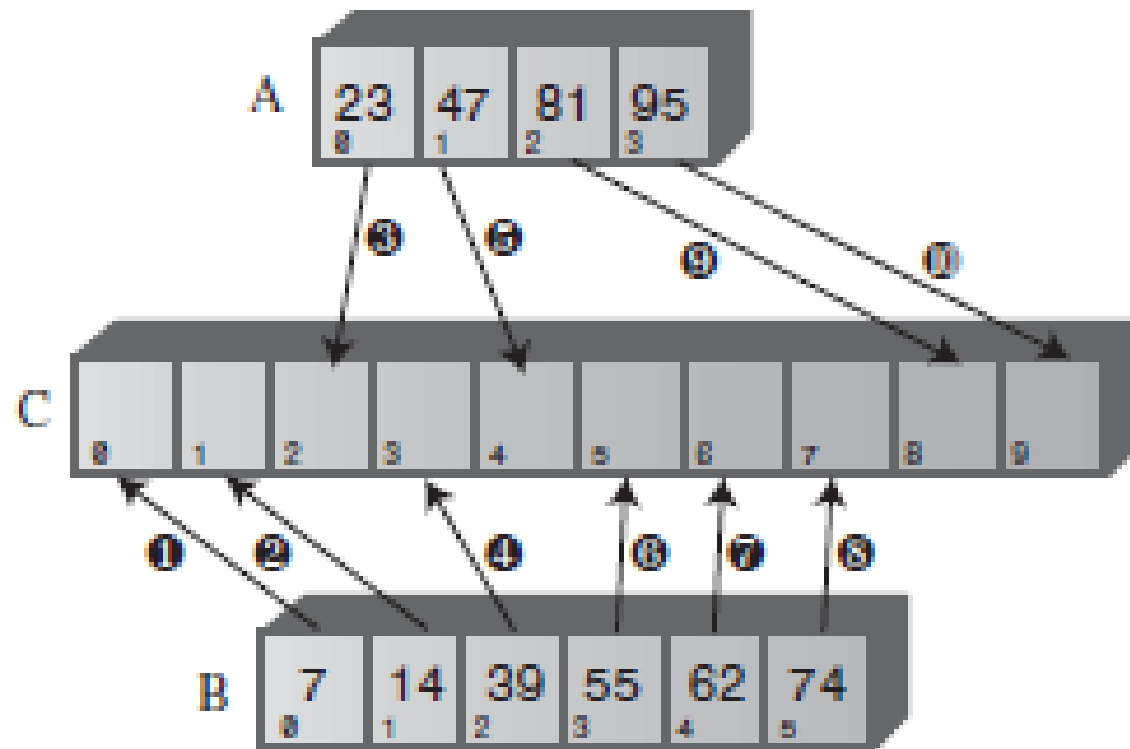


Presupunem că avem doi vectori sortați

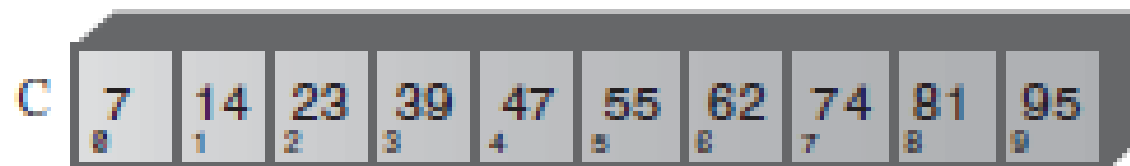
Vectorii pot avea dimensiuni diferite

Presupunem că vectorul **A** are **4** elemente,
iar **B** are **6** elemente

Vectorii vor fi interclasați în vectorul **C**, care
inițial conține **10** celule neocupate




a) Before Merge




b) After Merge

Pasul	Comparația (dacă este cazul)	Elementul copiat
1	Compară 23 și 7	Copiază 7 din B în C
2	Compară 23 și 14	Copiază 14 din B în C
3	Compară 23 și 39	Copiază 23 din A în C
4	Compară 39 și 47	Copiază 39 din B în C
5	Compară 55 și 47	Copiază 47 din A în C
6	Compară 55 și 81	Copiază 55 din B în C
7	Compară 62 și 81	Copiază 62 din B în C
8	Compară 74 și 81	Copiază 74 din B în C
9		Copiază 81 din A în C
10		Copiază 95 din A în C



```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void) {
    int a[] = {23, 47, 81, 95};
    int b[] = {7, 14, 39, 55, 62, 74};
    int c[10];
    int i = 0, j = 0, k = 0;
    int s = 4, t = 6;
    while (i < s && j < t)
        if (a[i] < b[j]) c[k++] = a[i++];
        else c[k++] = b[j++];
```

```
while (i < s)
    c[k++] = a[i++];
while (j < t)
    c[k++] = b[j++];
printf("Vectorul obtinut prin interclasare este:\n");
for (i = 0; i < s + t; i++)
    printf("C[%d] = %d\n", i, c[i]);
getch();
}
```


Sortarea prin interclasare

- Ideea sortării prin interclasare este de a împărți vectorul în două părți, de a sorta fiecare jumătate și apoi de a interclasa jumătățile sortate într-un singur vector
- Se împarte fiecare jumătate în două sferturi, se sortează aceste sferturi, după care se interclasează pentru a obține o jumătate sortată




Sortarea prin interclasare

- Similar, se interclasează perechi de optimi de vector pentru a obține sferturile sortate ș.a.m.d.
- Vectorul se împarte până când se obțin subvectori cu un singur element
- Acesta este punctul fix



Rezultatul unei funcții recursive se reduce, ca dimensiune, la fiecare apel recursiv, pentru a fi apoi reconstruit pe lanțul de revenire

Domeniul se divide în jumătăți la fiecare apel recursiv, iar la orice revenire, se realizează interclasarea a două domenii mai mici într-unul singur, mai mare

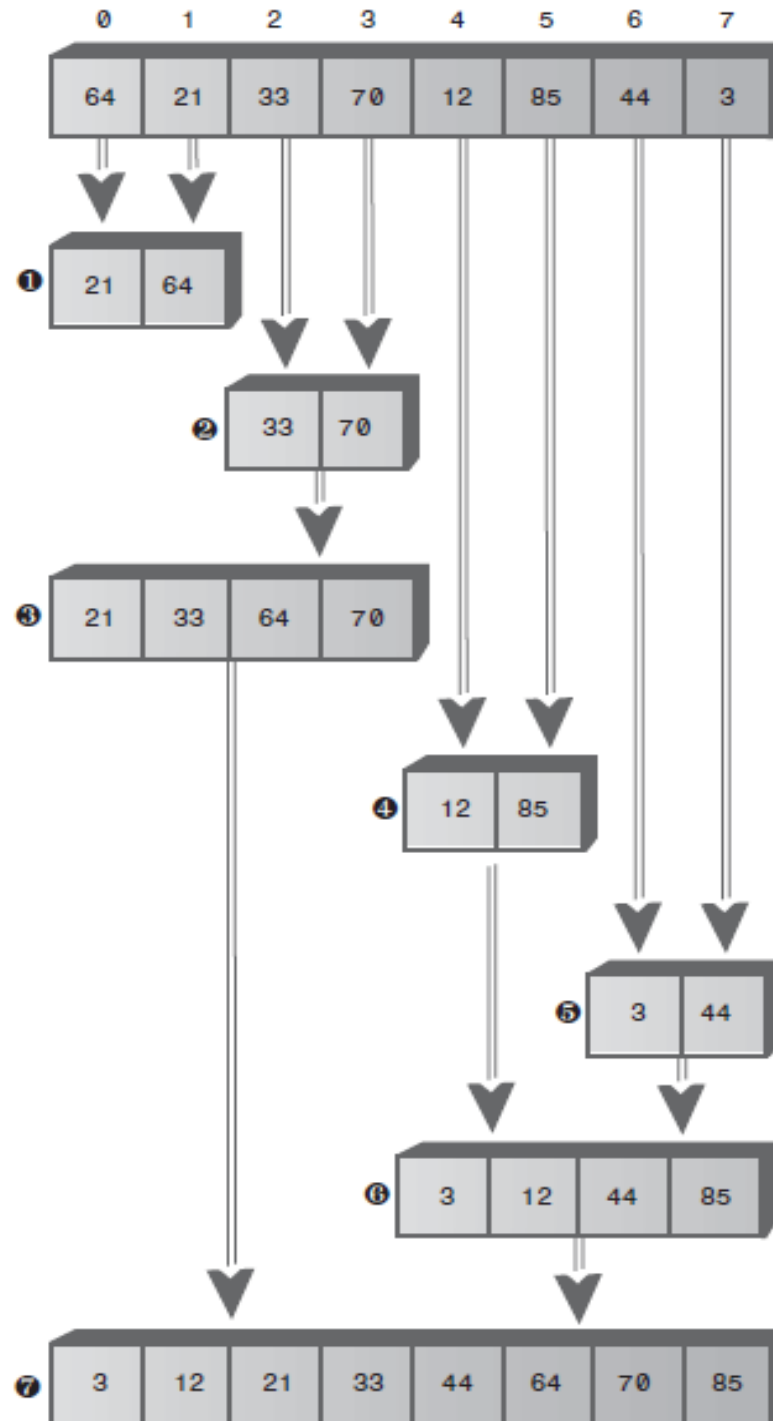



Când se revine, după ce s-au găsit doi vectori de câte un element fiecare, se interclasează într-un vector sortat, de două elemente

Fiecare pereche de vectori de 2 elemente astfel rezultate este interclasată apoi într-un vector de 4 elemente

Procesul continuă cu vectori din ce în ce mai mari, până la sortarea întregului vector

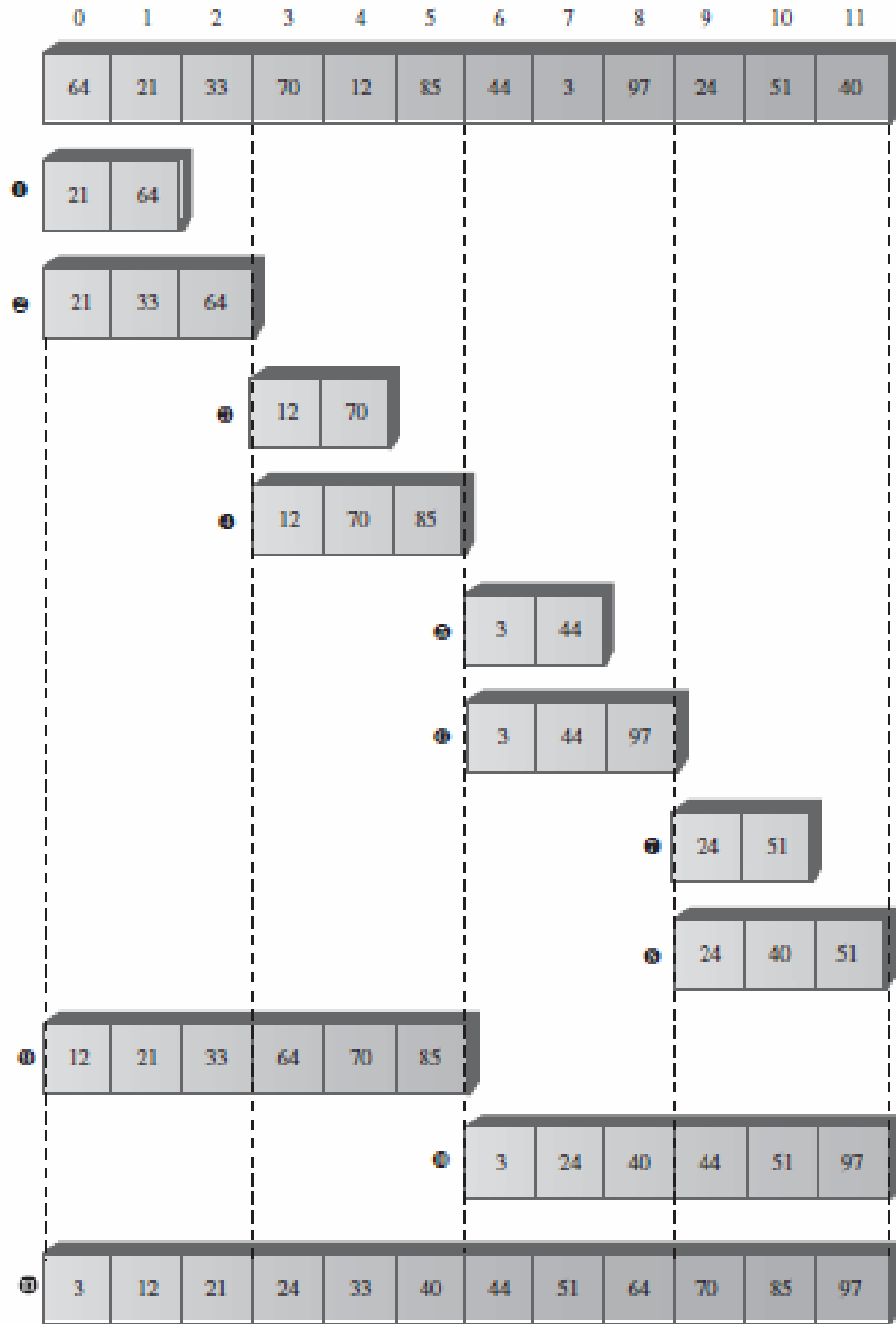
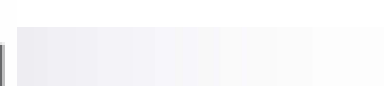
Cel mai simplu caz de ilustrat este acela în care dimensiunea vectorului inițial este o putere a lui 2






Când dimensiunea vectorului nu este o putere a lui 2, apar situații în care se interclasează vectori de dimensiuni diferite. Dacă vectorul inițial are dimensiunea 12, un vector de dimensiune 2 va fi interclasat cu unul de dimensiune 1, formând un vector de dimensiune 3.

Se interclasează părți componente ale aceluiași vector, spre deosebire de programul precedent, unde se interclasau doi vectori distincti în al treilea vector.






Toți sub-vectorii intermediari se memorează într-un vector de lucru, având aceeași dimensiune cu cel original

Sub-vectorii sunt memorați în secțiuni ale vectorului de lucru


Adică sub-vectorii din vectorul original sunt copiați în pozițiile corespunzătoare din vectorul de lucru

După fiecare operație de interclasare, se copiază vectorul de lucru înapoi în vectorul original


Vezi demonstrația MergeSort




```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int nElems; //numarul de elemente din vector
int *theArray; //vectorul care trebuie sortat crescator
void mergeSort();
void recMergeSort(int *workSpace, int lowerBound, int
                    upperBound);
void merge(int *workSpace, int lowPtr, int highPtr, int
            upperBound);
```



```
int main(void)
{
    printf("Introduceti dimensiunea vectorului
nElems = ");
    scanf("%d", &nElems);
    theArray = (int*) malloc(nElems * sizeof(int));
    // create the array
    for (int i = 0; i < nElems; i++) {
        printf("Introduceti elementul a[%d] = ", i);
        scanf("%d", &theArray[i]);
    }
}
```




```
printf("Vectorul inainte de sortare este\n");
    for (int i = 0; i < nElems; i++)
        printf("a[%d] = %d\n", i, theArray[i]);
    mergeSort();                // merge sort the array
    printf("Vectorul sortat este\n");
    for (int i = 0; i < nElems; i++)
        printf("a[%d] = %d\n", i, theArray[i]);
    getch();
} // end main()
```



```
void mergeSort()          // called by main()
{                          // provides workspace
    int *workSpace = (int*) malloc(nElems *
sizeof(int));
    recMergeSort(workSpace, 0, nElems - 1);
}
```

```
void recMergeSort(int *workSpace, int lowerBound,
int upperBound) {
    if(lowerBound == upperBound)    // if range is 1,
        return;                    // no use sorting
    else {                          // find midpoint
        int mid = (lowerBound + upperBound) / 2;
                                   // sort low half
        recMergeSort(workSpace, lowerBound, mid);
                                   // sort high half
        recMergeSort(workSpace, mid + 1,
upperBound);                      // merge them
        merge(workSpace, lowerBound, mid + 1,
upperBound);
    } // end else    } // end recMergeSort()
```



```
void merge(int *workSpace, int lowPtr, int highPtr, int  
upperBound)
```

```
//lowPtr - punctul de inceput al jumatatii inferioare
```

```
//highPtr - punctul de inceput al jumatatii
```

```
superioare
```

```
//upperBound - marginea de sus a jumatatii
```

```
superioare
```

```
//functia calculeaza dimensiunile sub-vectorilor,  
pornind de la aceste informatii
```

```
{
```


```
int j = 0; // workspace index
```

```
int lowerBound = lowPtr;
```

```
int mid = highPtr - 1;
```

```
int n = upperBound - lowerBound + 1;
```

```
// number of items
```



```
while(lowPtr <= mid && highPtr <= upperBound)
    if( theArray[lowPtr] < theArray[highPtr] )
        workSpace[j++] = theArray[lowPtr++];
    else
        workSpace[j++] = theArray[highPtr++];
while(lowPtr <= mid)
    workSpace[j++] = theArray[lowPtr++];
while(highPtr <= upperBound)
    workSpace[j++] = theArray[highPtr++];
for(j = 0; j < n; j++)
    theArray[lowerBound+j] = workSpace[j];
} // end merge()
```


Concluzii

- O funcție recursivă se **autoapelează repetat**, cu **valori diferite ale parametrilor**, pentru fiecare apel
- Unele valori ale parametrilor conduc la revenirea funcției, fără ca aceasta să se mai autoapeleze
- Astfel de situații se numesc **cazuri de bază** sau **puncte fixe**



Concluzii

- Atunci când apelul recursiv cel mai de jos își termină execuția, procesul se “desfășoară”, conducând la terminarea instanțelor în așteptare ale funcției, mergând înapoi de la apelurile mai recente până la apelul inițial al funcției

Concluzii

- Căutarea binară se poate efectua recursiv, verificând în care jumătate a domeniului sortat se află elementul căutat și apoi continuând algoritmul asupra acelei jumătăți

Concluzii

- Problema turnurilor din Hanoi se poate rezolva recursiv, deplasând toate discurile de pe tija inițială, mai puțin cel de la bază, pe o tijă intermediară, deplasând apoi discul de la bază pe tija destinație și, în fine, deplasând discurile de pe tija intermediară pe tija destinație

Concluzii

- Interclasarea a doi vectori sortați crescător constă în crearea unui al treilea vector, care conține toate elementele ambilor vectori, în ordine crescătoare



Concluzii

- În sortarea prin interclasare, sub-vectorii cu un singur element dintr-un vector mai mare sunt interclasate, rezultând sub-vectori cu 2 elemente

Concluzii

- Aceștia sunt interclasați în vectori cu 4 elemente ș.a.m.d., până când tot vectorul este sortat
- Sortarea prin interclasare necesită un vector de lucru de dimensiune egală cu vectorul inițial

Concluzii

- Pentru căutarea binară, funcția recursivă conține un singur apel către ea însăși
- Deși programul de căutare binară conține două astfel de apeluri, numai unul se utilizează la orice trecere prin codul funcției



Concluzii

- Pentru problema turnurilor din Hanoi și pentru sortarea prin interclasare, funcția recursivă se autoapelează de două ori