Recapitulare notiuni C/C++. Complexitatea algoritmilor. Recursivitate.

1. **Recapitularea unor notiuni C. Limbajul C++.**
2. **Complexitatea algoritmilor – algoritmi de sortare.**
3. **Recursivitate**

1. Recapitularea unor noțiuni C. Limbajul C++.

Expresii si valori stânga

In C ne putem referi la un obiect folosind o expresie "valoare-stânga". O expresie "valoare-stânga" este o expresie care poate sa apară in STÂNGA unei atribuiri. De exemplu, numele unei variabile este o expresie valoare-stânga in timp ce rezultatul unei operații aritmetice nu este valoare-stânga.

```
int i;
int v[10];
```

Valori stinga: `i  v[i]  v[i+1]`
Nu sunt valori stinga: `i+1  2*v[i]`

In C, prin combinarea numelor de variabile cu anumiți operatori se obțin valori-stânga.

Operatorul *

Se aplica numai pointerilor, rezultatul fiind o valoare-stânga care se refera la obiectul a cărei adresă este conținută de către pointer:

```
int* p;          // valori stinga:
p = &i;          //  p      nume_variabila
*p=5;            // *p      * pointer
```

Observați ca expresia *p (rezultată din combinația numelui lui p cu operatorul *) apare in stânga atribuirii.

Operatorul []

Se aplica numelor de tablouri si pointerilor, rezultatul fiind o valoare-stânga care se referă la obiectul al *n*-lea din tablou:

```
int tab[10];
int* p = &tab[0];          // valori stinga:
tab[2] = 3;                // nume_tablou [ index ]
p[2] = 4;                  // pointer [ index ]
```

Mai sus, pointerul `p` este inițializat cu adresa primului element din vectorul `tab`. Expresia `p[2]` va referi al doilea element din vectorul a cărui adresa este memorată in pointerul `p`, deci `tab[2]`.

Operatorii . si ->

Apar in legătura cu structurile si vor fi tratați puțin mai târziu.

Structuri

1. Definire

O structură este un tip de date nou. Atunci când definim o structură, trebuie să specificăm numele structurii şi câmpurile ei:

```
struct student {
   char* nume;
   int   nota;
   };
```

Am introdus tipul `struct student`. Pentru a evita repetarea lui `struct` putem să introducem un pseudonim pentru tipul `struct student` si anume `Student` astfel:

```
typedef struct student  Student;
```

Cele două declarații de mai sus pot fi comprimate in una singură:

```
typedef struct {
   char* nume;
   int   nota;
   } Student ;
```

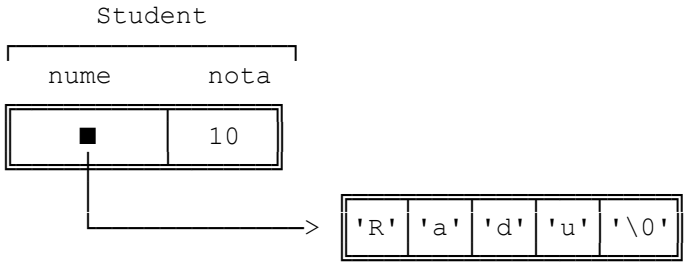
In C++, tipul definit de declarația:

```
struct Student {
   char* nume;
   int   nota;
   };
```

poate fi denumit `struct Student` sau, doar simplu, `Student`. In continuare ne vom folosi de această facilitate care măreşte lizibilitatea programelor. Repetăm: **programele vor avea extensia .CPP** .

2. Obiecte de tip structură

Am definit structura `Student` având CÂMPURILE "*nume*" (adresa unui sir de caractere) si "*nota*" de tip int.



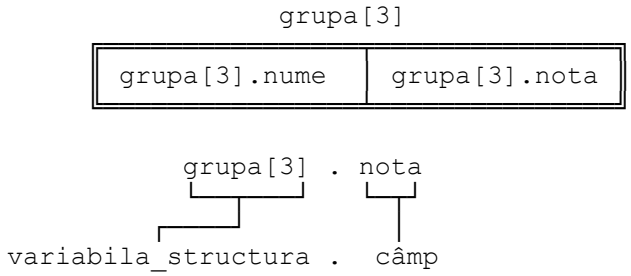
Odata definit tipul, acesta poate fi folosit pentru a declara variabile:

```
Student s1, s2;          // s1 si s2 sint doua variabile de tip Student
Student grupa[5];        // un tablou de variable Student
Student* ps;             // o variabila pointer la Student
Student* idx[5];         // o variabila tablou de pointeri la Student
```

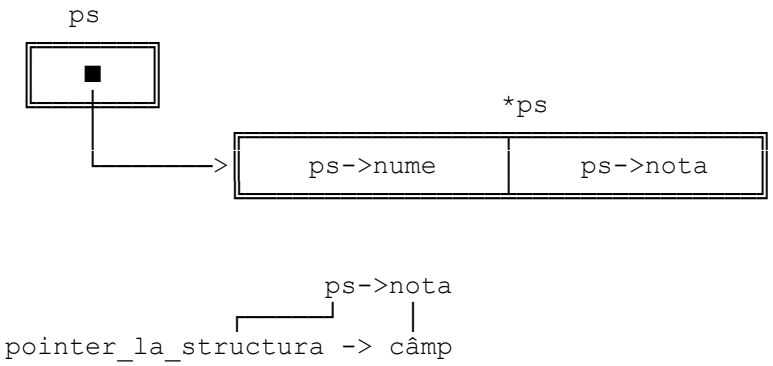
3. Operatorii . si ->

Folosirea câmpurilor unei structuri se face NUMAI CU REFERIRE LA UN OBIECT de tipul respectiv. Obiectul este referit printr-o valoare stânga care semnifică obiectul structură sau adresa obiectului structură.

Operatorul **.** cere in stânga sa o valoare stânga de tip structura iar in dreapta, numele câmpului selectat, rezultatul fiind o valoare-stânga care se refera la câmpul selectat. De exemplu, din declarațiile de mai sus, câmpurile variabilei `grupa[3]` vor fi denumite: `grupa[3].nume` si `grupa[3].nota`



Operatorul **->** cere in stânga sa o expresie de tip "pointer la structura" iar in dreapta numele câmpului selectat, rezultatul fiind o valoare-stânga care se refera la câmpul selectat:



Exerciții

Având următoarele declarații:

```
int i, *pi;
Student s;
Student* ps;
Student ts[5];
Student* tps[5];
```

numiți tipurile următoarelor expresii. Decideți daca sunt valori stânga sau nu:

i+2	pi	i	p+3
ts[2]	ts	ps	ps[2]
ps->nume	*pi	*(pi+2)	*pi+2
(ps+3)->nume	ps->nume[2]	tps	tps[2]
tps[2]->nume	tps[2]->nume[2]		

Pointeri

1. Definire

Pentru un tip de date T o variabila "*pointer la T*" se defineşte astfel:

```
T* ptrT;        // ptrT este un pointer la T
```

O variabila pointer la T poate retine adresa unui obiect de tip T.

Exemple:

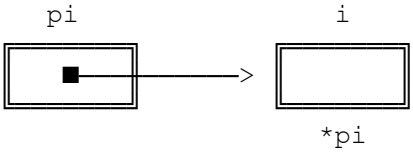
```
int* pi;          // pi este un pointer la int
char* tab;        // tab este un pointer la char
Nod *nou, *cap;   // nou si cap sunt pointeri la tipul Nod
```

2. Inițializare

Prima operație care se face cu un pointer este inițializarea. Un "*pointer la T*" poate fi inițializat cu:

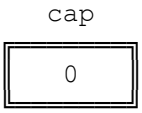
a) adresa unui T (care exista)

```
pi = &i;
```

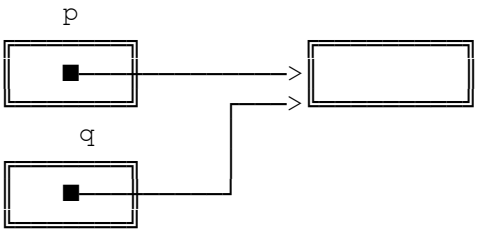


b) valoarea 0 (sau NULL) care semnifica adresa invalida

```
cap = 0;
```

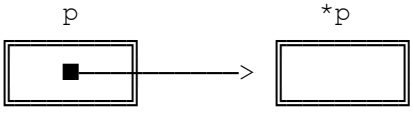


c) valoarea altui "*pointer la T*". De exemplu: daca *p* si *q* sunt de tip *T** si *p* conține adresa unei variabile de tip *T* (a fost inițializat in prealabil), atribuirea *q = p* va face ca ambii pointeri sa indice aceeaşi variabilă.



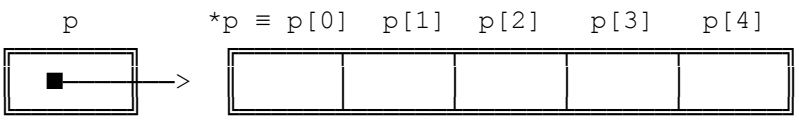
d) adresa unui spațiu de memorie alocat in zona de alocare dinamica. Spațiul alocat poate sa conțină un singur obiect de tip T, acesta se exprima:

in C: `p = (T*) malloc( sizeof(T) );`
in C++: `p = new T;`



sau poate să conțină mai multe (n) obiecte de tip T:

in C: `p = (T*) malloc( n*sizeof(T) );`
in C++: `p = new T[n];`



Exprimările din C++ sunt in mod evident mult mai simple si le vom folosi pe acestea in continuare.

Iată un exemplu:

```
typedef Student* PStudent;
PStudent* ptps;                    // pointer la tablou de pointeri la studenți
ptps = new PStudent[nr];
```

3. Dereferențierea

Este operația prin care având un "*pointer la T*" (care conține adresa unui T) obținem o valoare stânga care se refera la obiectul pointat (vezi operatorul *).

Pentru a obține obiectul pointat folosim operatorul * astfel:

```
*pi = 5;          // obiectul pointat ia valoarea 5
```

!!! Atentie !!!
Aceasta operație poate fi aplicata numai pointerilor care conțin intr-adevăr adresa unui obiect. De exemplu, nu puteți face dereferentierea unui pointer nul (cu valoare 0) sau a unui pointer neinițializat. Este valabil si pentru operatorul -> care conține si el o dereferentiere care se observa in scrierea echivalenta: ps->nota este echivalent cu (*ps).nota

4. Pointeri si tablouri

Numele unui "*tablou de T*" este convertit automat la tipul "*pointer la T*", deci poate fi folosit pentru a inițializa un "*pointer la T*". Valoarea acestui pointer este adresa primului element al tabloului:

```
T tab[10];
T* ptrT = tab;    // ptrT conține adresa primului element
```

Un "*pointer la T*" este deseori folosit pentru a se referi pe rând la elementele unui tablou.

Următoarele operații semnifica:

```
ptrT++      // pointează la următorul element din tablou
            // creează o valoare stânga!
 ptrT--     // pointează la elementul precedent din tablou
            // creează o valoare stânga!
```

5. Eliberarea spațiului alocat dinamic

Daca un pointer a fost inițializat cu adresa unui spațiu din zona de alocare dinamică, atunci când nu mai avem nevoie de spațiul respectiv (adică nu mai avem nevoie de obiectul din spațiul respectiv) vom elibera spațiul. El va putea fi astfel utilizat pentru alocări ulterioare.

Daca *p* este un pointer care a fost inițializat printr-o alocare de memorie, eliberarea memoriei alocate se exprimă:

in C: `free(p);`
in C++: `delete p;`

!!!Atentie!!!
Nici `free()` nici `delete` nu modifică valoarea pointerului *p,* dar obiectul a cărui adresa este conținută de *p* nu trebuie sa fie referit după eliberare.

6. Observație

In laboratoarele următoare, vor exista cazuri in care anumiți pointeri nu sunt variabile simple, ci vor fi componente ale unor structuri de date complexe. Toate regulile de mai sus se păstrează.

Referințe

Când vrem ca o funcție, atunci când este apelata, sa modifice valoarea unei variabile din funcția apelanta, trebuie sa trimitem ca argument un pointer la acea variabilă. De exemplu, o funcție care interschimbă valoarea a doi "*pointeri la student*" va trebui să primească ca parametri doi "*pointeri la pointeri la student*":

```
void Schimba(Student** unu, Student** doi)
{
     Student* trei;
     trei = *unu;
     *unu = *doi;
     *doi = trei;
}
```

Daca argumentele au tipuri mai complicate, atunci sintaxa din interiorul funcției devine greoaie. Pentru a rezolva aceasta problema putem trimite ca argumente REFERINTE. Un argument referința trebuie interpretat ca fiind un PSEUDONIM pentru argumentul pasat. Orice modificare a referinței se va face asupra argumentului pasat:

```
void Schimba(Student*& unu, Student*& doi)
{
     Student* trei;
     trei = unu;
     unu = doi;
     doi = trei;
}
```

Observați ca sintaxa din interiorul funcției a devenit mai simplă.

Scriere / citire cu ajutorul cin şi cout

În C++ s-a elaborat o modalitate mai simplă de scriere/citire la consolă comparativ cu funcțiile scanf/printf din C. La începutul execuției fiecărui program sunt instanțiate automat 2 variabile globale speciale - cin şi cout. Ele sunt folosite pentru citire, respectiv scriere la consolă.

Pentru a citi o variabilă de la consolă, se foloseşte următoarea sintaxă:

```
int a;
cin >> a;
```

Operatorul >> are un rol special pentru variabila cin. Expresia

```
cin >> a;
```

semnifică faptul că de la consolă este citită o valoare şi depozitată în variabila a. Tipul variabilei din dreapta poate fi de orice tip simplu – int, char, float, double sau şir de caractere – char*. Pentru fiecare tip citirea se va face în mod corect.

Pentru a scrie o variabilă la consolă, folosim sintaxa:

```
char str[] = "abc";
cout << str;
```

În mod similar, operatorul << are o semnificație specială pentru variabila cout. Expresia:

```
cout << str;
```

semnifică faptul că variabila `str` este scrisă la consola. Variabilele scrise pot fi de aceleaşi tipuri ca şi cele citite cu cin.

Observați că în exemplul de mai sus a fost scrisă la consolă o variabilă de tip char[], tip care nu a fost menționat în lista de tipuri suportate pentru operandul dreapta. Totuşi, utilizarea lui a fost posibilă într-o expresie cu cout. De ce?

Variabilele cin şi cout sunt definite în header-ul <iostream>. Pentru a le putea folosi, trebuie să includem la începutul programului următoarele linii:

```
#include <iostream>
using namespace std;
```

Aceste variabile speciale fac parte din categoria "obiecte". Obiectele vor fi studiate în detaliu la Programare orintată obiect (POO).

Iată un exemplu complet folosind noile facilități de scriere/citire:

```
// exemplu: cin si cout
#include<conio.h>
#include <iostream>
using namespace std;

int main()
{
  int iVal;
  char sVal[30];

  cout << "Introduceti un numar: ";
  cin >> iVal;
  cout << "Si un sir de caractere: ";
  cin >> sVal;
  cout << "Numarul este: " << iVal << "\n"
       << "Sirul este: " << sVal << endl;
  _getch();
  return 0;
}
```

```
Introduceti un numar: 12
Si un sir de caractere: abc
Numarul este: 12
Sirul este: abc
```

Un aspect nou este cuvântul `endl`. Acesta este o variabilă globală tip şir de caractere cu valoarea "\n" (sfârşit de linie).

Sintaxa:

```
cout << endl;
```

este echivalentă cu:

```
cout << "\n";
```

dar are avantajul că este mai lizibilă.

Atât expresiile cu cin cât şi cele cu cout pot fi înlănțuite. Expresia:

```
cout << a << " " << b;
```

este echivalentă cu:

```
cout << a;
cout << " ";
cout << b;
```

Comparativ cu funcțiile printf / scanf din C, expresiile cu cin şi cout sunt mai simple şi mai uşor de înțeles. Nu mai este nevoie de specificatori de format. Dezavantajul este că nu putem face afişări formatate pe un anumit număr de caractere. O afişare de genul:

```
printf("%7.2f", f);
```

nu are echivalent folosind cout.

De aceea, folosind cout nu vom putea afişa o matrice sau un vector de structuri sub formă de tabel, aşa cum am făcut în C. Pentru astfel de afişări vom folosi printf.

Operatorii new şi delete

În C++, lucrul cu memoria dinamică este facilitat de doi operatori speciali - new şi delete.

Alocarea dinamică de memorie se face cu operatorul new. El returnează un pointer către începutul blocului de memorie proaspăt alocat. Sintaxa operatorului este următoarea:

```
TIP *p, *pvector;
p = new TIP;
pvector = new TIP[nr_elemente];
```

De exemplu, pentru pointeri la variabile simple:

```
int *pi;
pi = new int;
char *pch;
pch = new char;
```

Şi pentru pointeri la vectori:

```
int *vi;
vi = new int[10];
char *psir;
psir = new char[80];
```

Atunci când nu mai avem nevoie de o variabilă alocată dinamic, aceasta trebuie dealocată. Memoria dealocată devine disponibilă pentru noi cereri de alocare. Pentru dealocari se folosesc operatorii delete – pentru variabile simple, şi delete[] – pentru vectori. Exemplu:

```
delete vi;
delete[] psir;
```

2. Complexitatea algoritmilor

2.1. Consideratii teoretice

La evaluarea (estimarea) algoritmilor secvențiali se pune in evidenta necesarul de timp si de spatiu de memorare al lor.

Studierea complexității presupune analiza completa in cadrul algoritmului a următoarelor 3 apecte:

- configurația de date cea mai defavorabilă (cazurile degenerate);
- configurația de date cea mai favorabilă;
- comportarea medie.

Comportarea medie presupune probabilitatea de apariție a diferitelor configurații de date la intrare. Cazul cel mai defavorabil este cel mai studiat si este folosit, de obicei, pentru compararea algoritmilor.

Complexitatea unui algoritm secvențial se exprima de regulă in limbajul ordinului O.

Definitie
Fie două funcții f : N->N şi g : N->N.
Spunem ca f aparține O(g) (are ordinul de complexitate O(g)) şi se notează f = O(g) dacă şi numai dacă există o constantă reala c si un număr natural n_0 astfel încât pentru n> n_0 => f(n) < c * g(n)

Observatie:
f : N->N este o funcție f(n) cu n dimensiunea datelor de intrare.
f(n) reprezintă timpul de lucru al algoritmului exprimat in "pasi".

Laborator de Structuri de Date – Lucrarea nr. 1

Lema 1
Daca f este o funcție polinomiala de grad k atunci $f = O(n^k)$.

Concluzie: $f = O(n^k)$, si ordinul O exprima viteza de variație a funcției, funcție de argument.

Proprietăți:

1) Fie f, g : N->N.
Daca $f = O(g)$ $\quad k * f = O(g)$
$f = O(k * g)$, $k \in R$ constant.

2) Fie f, g, h : N->N.
si: $f = O(g)$
$g = O(h)$ atunci $f = O(h)$

3) Fie f1, f2, g1, g2 : N->N.
si: $f1 = O(g1) \Rightarrow f1 + f2 = O(g1 + g2)$
$f2 = O(g2) \Rightarrow f1 * f2 = O(g1 * g2)$

Aceasta proprietate permite ca, atunci când avem două bucle imbricate (de complexități diferite), complexitatea totala sa se obțină înmulțindu-se cele doua complexități. Cele doua complexități se adună, daca buclele sunt succesive.

Intre clasa funcțiilor logaritmice, si cea a funcțiilor polinomiale exista relația:
$O(n^c)$ inclusa in $O(a^n)$.

Au loc următoarele incluziuni:
O(1) in ***O(log n)*** in ***O(n)*** in ***O(nlog n)*** in ***$O(n^2)$*** in ... in ***$O(n^k \log n)$*** in ***$O(n^{k+1})$*** in ***$O(2^n)$***

Pentru calculul complexității se va încerca încadrarea in clasa cea mai mica de complexitate din acest sir:

O(1)	clasa algoritmilor constanti;
O(log n)	clasa algoritmilor logaritmici;
O(n)	clasa algoritmilor liniari;
O(nlog n)	clasa algoritmilor polilogaritmici;
$O(n^2)$	clasa algoritmilor patratici;
$O(n^k \log n)$	clasa algoritmilor polilogaritmici;
$O(n^{k+1})$	clasa algoritmilor polinomiali;
$O(2^n)$	clasa algoritmilor exponențiali.

Exemple privind analiza algoritmilor

1. O buclă for

```
for (i=0;i<n;i++)
{
   S;
}    //secvență de ordin O(1)
```

$\Rightarrow$ n*O(1) = O(n)

2. Două bucle for

```
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
   {
      S;
   }    //secvență de ordin O(1)
```

$\Rightarrow$ n*n*O(1) = O(n^2)

3. Două bucle for

```
for (j=0;j<n;j++)
   for (i=0;i<j;i++)
   {
      S;
   }    //secvență de ordin O(1)
```

$\sum i = \frac{n(n+1)}{2} \Rightarrow$ O(n^2)

4. Buclă while

```
h=1;
while (h<=n)
{S;       //secvență de ordin O(1)
h=2*h;
}
```

pas	1	2	3	4.....	*k*
h	2^1	2^2	2^3	2^4.....	$2^k \le n$

$2^k \le n \Rightarrow k \le \log_2 n \Rightarrow O(\log n)$

5. Buclă while ce conține buclă for

```
h=n;
while (h>10^-6)
{
   for (i=0;i<n;i++)
      S;     //secvență de ordin O(1)
   h=h/2;
}
```

$$\left.\begin{array}{l} f_1 = O(g_1(n)) \\ f_2 = O(g_2(n)) \end{array}\right\} \Rightarrow f_1 \cdot f_2 = O(g_1(n) \cdot g_2(n))$$

while – O(log n); for – O(n) $\Rightarrow$ O(n·log n)

3.2. Algoritmi de sortare

Un vector ordonat reduce timpul anumitor operații ca de pildă căutarea unei valori date, verificarea unicității elementelor, găsirea perechii celei mai apropiate, calculul frecvenței de apariție a fiecărei valori distincte etc. Ordonarea vectorilor se face atunci când este necesar, de exemplu pentru afișarea elementelor sortate după o anumită cheie.

În general, operația de sortare este eficientă dacă se aplică asupra vectorilor şi de obicei nu se aplică altor structuri de date (liste, arbori neordonați sau tabele de dispersie).

Există mai mulți algoritmi de sortare cu performanțe diferite. Cei mai ineficienți algoritmi de sortare au o complexitate de ordinul $O(n^2)$, iar cei mai buni necesită pentru cazul mediu un timp de ordinul O(n*log(n)).

În anumite aplicații, ne interesează un algoritm de sortare "stabilă", care păstrează ordinea inițială a valorilor egale din vectorul sortat. Există algoritmi care nu sunt stabili.

De multe ori prezintă interes algoritmii de sortare „pe loc", care nu necesită memorie suplimentară. Algoritmii de sortare "pe loc" a unui vector se bazează pe compararea de elemente din vector, urmată eventual de schimbarea între ele a elementelor comparate pentru a respecta condiția de inegalitate valorică față de cele precedente si cele care-i urmează.

În algoritmii ce urmează, se consideră sortarea crescătoare a elementelor unui vector A.

Algoritmul Bubble Sort

Sortarea prin metoda bulelor (Bubble Sort) compară mereu elemente vecine. În prima parcurgere, ce se compară toate perechile vecine (de la prima către ultima) şi dacă este cazul se schimbă elementele între ele pentru a asigura ordinea dorită. După prima parcurgere, valoarea maximă se va afla la sfârşitul vectorului. La următoarele parcurgeri se reduce treptat dimensiunea vectorului, prin eliminarea valorilor finale (deja sortate). Dacă se compară perechile de elemente vecine de la ultima către prima, atunci se aduce în prima poziție valoarea minimă şi apoi se modifică indexul de început.

PseudoCod:

```
bubble_sort(A,n)
{     swapped=1;
      j=0;
      while(swapped)
      {
            swapped=0;
            j = j+1
            for i = 1 to n-j do
                  if (A[i]>A[i+1]
                  {
                        tmp = A[i];
                        A[i] = A[i + 1];
                        A[i + 1] = tmp;
                        swapped = 1;
                  }
      }
}
```

Timpul de sortare prin metoda bulelor este proporțional cu pătratul dimensiunii vectorului (complexitatea algoritmului este de ordinul n^2).

Algoritmul SelectionSort

Sortarea prin selecție determină în mod repetat elementul minim dintre toate care urmează unui element A[i] şi îl aduce în poziția i, după care creşte indexul i.

```
void selSort( T a[ ], int n)
{ // sortare prin selectie
      int i, j, m; // m = indice element minim dintre i,i+1,..n
      for (i = 0; i < n-1; i++)
       {    // in poz. i se aduce min (a[i+1],..[a[n])
            m = i; // considera ca minim este a[i]
            for (j = i+1; j < n; j++) // compara minim partial cu a[j]
                                      //(j > i)
                  if ( a[j] < a[m] ) // a[m] este elementul minim
                        m = j;
            swap(a,i,m); // se aduce minim din poz. m in pozitia i
      }
}
```

Sortarea prin selectie are si ea complexitatea $O(n^2)$, dar în medie este mai rapidă decât sortarea prin metoda bulelor (constanta care înmulteste pe n^2 este mai micã).

Algoritmul Insertion Sort

Algoritmul INSERTION_SORT consideră că in pasul ***k***, elementele ***A[1÷k-1]*** sunt sortate, iar elementul ***k*** va fi inserat, astfel încât, după aceasta inserare, primele elemente ***A[1÷k]*** sa fie sortate.

Pentru a realiza inserarea elementului ***k*** in secvența ***A[1÷k-1]***, aceasta presupune:

- memorarea elementului intr-o variabilă temporară;
- deplasarea tuturor elementelor din vectorul ***A[1÷k-1]*** care sunt mai mari decât ***A[k]***, cu o poziție la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);
- plasarea lui ***A[k]*** in locul ultimului element deplasat.

Complexitate: ***O(n)***

PseudoCod:

```
insertion_sort(A,n)
{
      for k = 2 to n do
              temp = A[k]
              i=k-1
              while (i >= 1) and (A[i] > temp) do
                        A[i+1] = A[i]
                        i = i-1
              A[i+1] = temp
}
```

Cazul cel mai defavorabil: situația in care deplasarea (la dreapta cu o poziție in vederea inserării) se face pana la începutul vectorului, adică şirul este ordonat descrescător.

Exprimarea timpului de lucru:
T(n) = 3 (n - 1) + 3 (1 + 2 + 3+ ... + n - 1) = 3(n-1) + 3n * (n - 1)/2
Rezulta complexitatea: ***T(n) = O(n^2)*** funcție polinomială de gradul II.

Observatie: Când avem mai multe bucle imbricate, termenii buclei celei mai interioare dau gradul polinomului egal cu gradul algoritmului.

Bucla cea mai interioara ne dă complexitatea algoritmului.

3.3. Cautarea binara (Binary Search)

Fie ***A***, de ordin ***n***, un vector ordonat crescător. Se cere sa se determine daca o valoare ***b*** se afla printre elementele vectorului. Limita inferioara se numeşte ***low***, limita superioara se numeşte ***high***, iar mijlocul virtual al vectorului, ***mid*** (de la middle).

PseudoCod:

```
Binary_search(A,n,b)
{
      low = 1
      high = n
      while low <= high do
            mid = [(low + high)/2]    // partea intreaga
            if A[mid] = b then
                  return mid
            else
                  if A[mid] > b then
                        high = mid-1  // restrang cautarea la stanga
                  else
                        low = mid+1   // restrang cautarea la dreapta
      return(0)
}
```

Calculul complexității algoritmului consta in determinarea numărului de ori pentru care se executa bucla *while*. Se observa că, la fiecare trecere, dimensiunea zonei căutate se înjumătățeşte.

Cazul cel mai defavorabil este ca elementul căutat să nu se găsească in vector.

Pentru simplitate, se considera $n = 2^k$, unde k este numărul de înjumătățiri.

Rezulta $k = \log_2 n$ si făcând o majorare,
$T(n) <= \log_2 n + 1$, a.î.
$2 * 2^k <= n < 2^{k+1}$

Rezultă complexitatea acestui algoritm: este ***O(log₂ n).*** Dar, baza logaritmului se poate ignora, deoarece: $\log_a x = \log_a b * \log_b x$ si $\log_a b$ este o constantă, deci rămâne ***O(log n)***, adică o funcție logaritmică.

Căutare binară (Binary Search)
Se caută valoare b în vectorul A[n]
st – limita din stânga
dr - limita din dreapta
m – mijlocul (st+dr)/2

varianta iterativă

```
BINARY_SEARCH(A,n,b)
st←1;
dr←n;
while (st<=dr) do
        m←(st+dr)/2;
        if a[m]=b  then
          return m;
        else
          if a[m]>b then
              dr←m-1;
          else
              st←m+1;
          end_if
        end_if
end_while
end
```

Se consideră $n=2^k$, k - numărul de înjumătățiri

$k = \log_2 n \Rightarrow$ timpul de execuție

$T(n) \le \log_2 n + 1 \Rightarrow O(\log n)$

In cele ce urmează va fi prezentată şi o variantă recursivă a algoritmului de căutare binară.

3. Recursivitate

Recursivitatea este una dintre noțiunile fundamentale ale informaticii şi constă în posibilitatea unui subprogram de a se autoapela o dată sau de mai multe ori.

Recursivitatea a apărut din necesitatea de a transcrie direct formule matematice recursive. În timp acest mecanism a fost extins şi pentru alți algoritmi.

În cazul autoapelării unui algoritm (unei funcții), în ceea ce privește transmiterea parametrilor se procedează ca la orice apel de funcție (subprogram). Pentru memorarea parametrilor se folosește o zonă de memorie numită stivă iar memorarea parametrilor se realizează în ordinea in care aceştia apar în antet. Parametri pot fi transmişi prin valoare sau prin referință (caz în care de fapt se transmite o adresă). În cadrul subprogramului, parametrii transmişi şi memorați în stivă sunt variabile.

Funcțiile recursive tipice corespund unei relații de recurență de tipul f(n)=rec(f(n-1)), n fiind parametrul după care se face recursivitatea)funcția poate avea mai mulți

parametri). La fiecare nou apel valoarea parametrului n se decrementează, până când n ajunge 1 sau 0, iar valoarea f(1) sau f(0) se poate calcula direct.

Orice subprogram recursiv poate fi rescris si nerecursiv, iterativ, prin repetarea explicitã a operatiilor executate la fiecare apel recursiv. O funcție recursivã realizeazã repetarea unor operatii fãrã a folosi instructiuni de ciclare.

In unele cazuri, ca de exemplu operațiile cu arborilor binari, utilizarea funcțiilor recursive este mai naturală şi oferăun cod sursă mai simplu şi mai elegant. In schimb, pentru implementarea operațiilor specifice listelor, pentru anumite calcule sau pentru operații de căutare este mai simplu şi mai eficient să se utilizeze variante iterative.

Observații

- Recursivitatea nu este de neînlocuit! Orice funcție recursivă se poate transforma într-o structură ciclică.
- În cazul unui număr foarte mare de autoapelări, există posibilitatea ca stiva implicită (folosită de compilator) să se ocupe total, caz în care programul se va termina cu eroare.
- Recursivitatea presupune mai multă memorie.
- Un algoritm recursiv poate fi privit ca fiind ierarhizat pe niveluri (ce corespund nivelurilor din stivă), astfel încât:
  - Ce se execută pe un nivel se execută pe orice nivel.
  - Subprogramul care se autoapelează trebuie să conțină instrucțiunile corespunzătoare unui nivel.
- Funcțiile recursive au cel puțin un argument, a cărui valoare se modifică de la un apel la altul şi care este verificat pentru oprirea procesului recursiv.
- Orice subprogram recursiv trebuie să conțină o instrucțiune "if" (de obicei la început), care să verifice condiția de oprire a procesului recursiv. In caz contrar, se ajunge la un proces recursiv ce tinde la infinit şi se opreşte numai prin umplerea stivei.

R1. Algoritmul recursiv pentru calculul factorialului (n!)

0!=1
n!= 1*2*3*...*n, n>0

```
// n! fara recursivitate
unsigned long fact(int n)
{
	int i;
	unsigned long p=1;
	if(n==0) return 1;
	else
	{
	for(i=1;i<=n;i++)
	p*=i;
	return p;
	}
}
```

0!=1
n!= n*(n-1)!, n>0

```
// calzulez n! prin recursivitate
int factorial(int n)
{
	if(n == 0) return 1;
	else
	return n*factorial(n-1);
}
```

R2. Folosind o funcție recursivă, să se determine cel mai mare divizor comun a două numere introduse de la tastatură.

```
int cmmdc(int m, int n)
{
	if(!n) return m;
	return cmmdc(n,m%n);
}
```

R3. Să se calculeze suma $S_n = \sum_{i=0}^{n} \frac{x^i}{i!}$

Se utilizează trei funcții: factorial, putere şi S.

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

float putere(float x, int n)
{
	if (n==0)
		return 1;
	return x*putere(x,n-1);
}

float S(float x, int n)
{
	if (n==0)
		return 1;
	return S(x,n-1)+putere(x,n)/fact(n);
}
```

Varianta 1

```
S=1;
for(i=1;i<=n;i++)
	s+=putere(x,i)/fact(i);
```

Varianta 2. Relația de recurență a termenilor se poate scrie $T_i = \frac{x}{i} T_{i-1}$ iar varianta recursivă va fi

```
float S(float x, int n)
{
      static float T=1;
      if (n==0)
      {
            T=1;
            return 1;
      }
      return S(x,n-1)+(T*=x/n);
}
```

Obs.: variabilele statice îşi păstrează valoarea la ieşirea din funcție, valoare ce se regăseşte la următorul apel. Variabila statică T păstrează rezultatul (calculele efectuate) anterior.

R4. Căutarea binară (varianta recursivă)

```
Int binSearch(int a[],int b,int
st,int dr)
{
  int m;
  assert(st<=dr);
  if(st==dr)
     return (b==a[dr]) ? dr: -1 //*1
  else {                        //*2
      m=(st+dr)/2;
      if(b<=a[m])
            return
            binSearch(a,b,st,m);
      else
            return
            binSearch(a,b,m+1,dr);
      }
 }
```

a- timpul pentru secvența *1
b- timpul pentru secvența *2

T(n) satisface relația de recurență

$$T(n) = \begin{cases} T(n/2) + a & \text{daca n} > 1 \\ b & \text{daca n} = 1 \end{cases}$$

se demonstrează prin inducție că dacă T(n) satisface relația de recurență de mai sus atunci:

$$T(n) \le a * \log(n) + b \Rightarrow \text{O(log n)}$$

TEMA

Să se implementeze în Microsoft Visual Studio 2008, programe C++ pentru fiecare algoritm descris mai sus. Să se calculeze numărul de operații executate în cadrul fiecărui algoritm de sortare. Să se calculeze numărul de autoapeluri în cazul soluțiilor recursive.