



Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- Calculul notei finale
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- Demo Eclipse + Java

Agenda

- **Echipa Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- Calculul notei finale
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- Demo Eclipse + Java

Echipa

Octavian Dospinescu (doctav@uaic.ro)

Catalin Strimbei (linus@uaic.ro)

Sabina Necula (sabina.necula@uaic.ro)

Nicolai Romanov (nicolai.romanov3@gmail.com)

David Burcovschi (officedsoft@gmail.com)

Agenda

- Echipa **Algoritmi si Logica Programarii**
- **Programa analitica**
- Bibliografie recomandata
- Calculul notei finale
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- Demo Eclipse + Java

Programa analitica

- **Cap. 1 – Algoritmi – concepte generale**
- **Cap. 2 – Recursivitate**
- **Cap. 3 – Algoritmi de sortare**
- **Cap. 4 – Algoritmi de căutare**
- **Cap. 5 – Metoda greedy**
- **Cap. 6 – Metoda backtracking**
- **Cap. 7 – Tehnici de programare OO: abstractizare si genericitate**
- **Cap. 8 – Structuri de date bazate pe tablouri, liste înlanțuite și Iteratori**
- **Cap. 9 – Stive, cozi, tabele asociative și de dispersie**
- **Cap. 10 – Arbori**
- **Cap. 11 – Grafuri**
- **Cap. 12 – Șiruri de caractere**

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- **Bibliografie recomandata**
- Calculul notei finale
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- Demo Eclipse + Java

Bibliografie

Bibliografie

- Sedgewick Robert, Wayne Kevin, „Algorithms”, 4th Edition, Princeton University, Addison-Wesley, 2011
- Sedgewick Robert, Wayne Kevin, „Algorithms – Part 2”, 4th Edition, Princeton University, Addison-Wesley, 2014
- Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Java™, Sixth Edition, Wiley, 2014
- Erickson Jeff, „Algorithms”, free E-book, 2015
- Cormen Thomas, Leiserson Charles, Rivest Ronald, Stein Clifford, „Introduction to Algorithms”, 3rd Edition, MIT Press Cambridge, England, 2009
- **Pe langa aceste titluri vor mai fi facute recomandari pe parcursul semestrului, in mod punctual.**

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- **Calculul notei finale**
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- Demo Eclipse + Java

Nota

- Profesorii spera ca studentii sa obtina nota 10! 😊
- Componentele notei finale:
 - 20% un test practic in sapt. nr. 7 (aferent laboratorului);
 - 30% un test in sapt. 7 din capitolele 1-6 (aferent cursului);
 - 20% un test practic in saptamana nr. 14 (aferent laboratorului);
 - 30% un examen in sesiune din capitolele 7-12 (aferent cursului).
- Conditii cumulative de promovare:
 - Media testelor practice (laborator) sa fie minim 5.00;
 - Media testelor scrise (curs) sa fie minim 5.00.
- Testele aferente cursului se pot reface in sesiunea de restante, fiind parte a componentei de examen.
- Testele practice de laborator nu se pot reface in sesiune; ele constituie componenta de activitate pe parcurs.
- Nu se fac rotunjiri la notele individuale obtinute la teste

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- Calculul notei finale
- **Algoritmi – notiuni generale**
- Complexitatea algoritmilor
- Demo Eclipse + Java

Algoritmi – notiuni generale

- Algoritmul este o secventa de pasi logici care rezolva o **clasa de probleme**.
- Exemple:
 - Aflarea valorii maxime/minime dintr-un sir;
 - Ordonarea valorilor dintr-o multime;
 - Cautarea unei anumite valori intr-o multime;
 - Analiza unei secvente de text;
 - Gasirea drumului optim intr-o retea de transport;
 - Rezolvarea ecuatiei de grad 1 & 2

Algoritmi – reprezentare/implementare

- Reprezentarea formală a algoritmilor poate fi realizată prin:
 - Pseudocod;
 - Scheme logice.
- Implementarea se realizează în diverse limbaje de programare:
 - C#;
 - Java;
 - Python;
 - R;
 - etc.

!!! etc NU este limbaj de programare!!! 😊

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- Calculul notei finale
- Algoritmi – notiuni generale
- **Complexitatea algoritmilor**
- Demo Eclipse + Java

Complexitatea algoritmilor

Complexitatea unui algoritm se refera la resursele utilizate in timpul executiei:

- Spatiu de memorie
- Timp de executie (rulare)

Eficiența algoritmilor

Se evaluează din punct de vedere al timpului de execuție și al spațiului de memorie necesar.

Trebuie analizate:

- Complexitatea spațiu;
- Complexitatea timp.

Complexitatea **spatiu**

- Se refera la cantitatea de memorie necesara pentru rulara algoritmului.
- Depinde in cea mai mare masura de **tipurile** si **structurile de date** utilizate de catre algoritm.
- Discutie: tipuri primitive (int, char), tipuri struct, tipul obiect.

Complexitatea timp

Se refera la timpul necesar pentru rulara algoritmului.

Depinde de **numarul de operatii elementare** realizate de algoritm.

In general, un algoritm efectueaza 3 tipuri de operatii elementare:

- atribuire;
- decizie;
- intrare/iesire.

Putem considera ca fiecare operatie elementara are nevoie de acelasi interval de timp pentru executie, desi in realitate apar unele diferente.

Complexitatea timp

Complexitatea poate fi exprimata/masurata in 2 variante:

- **Cantitativ** (complexitatea exacta);
- **Calitativ** (complexitatea aproximativa).

Complexitatea **calitativa** ne ofera informatii despre **clasa de complexitate** a algoritmului.

Complexitate – masurare cantitativa

Vom calcula într-o funcție $F(n)$ numărul de operații elementare executate.

Exemplu: ordonarea unei multimi de n elemente:

```
pentru  $i \leftarrow 0, n-1$  execută  
    pentru  $j \leftarrow i+1, n$  execută  
        dacă  $(v[i] > v[j])$  atunci  
            temp  $\leftarrow v[i]$   
             $v[i] \leftarrow v[j]$   
             $v[j] \leftarrow temp$ 
```

Avem 3 cazuri posibile:

- cazul cel mai favorabil
- cazul cel mai nefavorabil
- cazul mediu

Complexitatea pe cazuri

Cazul cel mai favorabil (multimea este deja ordonata), deci se realizeaza doar operatia de comparatie.

$$F(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1 = \mathbf{n*(n-1)/2}$$

Cazul cel mai nefavorabil (multimea este ordonata invers), deci se realizeaza operatia de comparatie + 3 operatii de atribuire.

$$F(n) = 4 * ((n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1) = 4*n*(n-1)/2 = \mathbf{2*n*(n-1)}$$

Cazul mediu poate fi considerat a fi media aritmetica a cazurilor de mai sus.

$$F(n) = (n*(n-1)/2 + 2*n*(n-1)) / 2 = \mathbf{5*n*(n-1)/4}$$

Cazurile de mai sus au o complexitate patratica (n^2) si se foloseste notatia **$O(n^2)$** .

Complexitatea patratica garanteaza ca **timpul de executie este cel mult patrativ**.

Complexitatea pe cazuri

In realitate, **in functie de datele de intrare**, un algoritm ruleaza pe mai multe tipuri de cazuri:

- **Cazul cel mai favorabil;**
- **Cazul cel mai putin favorabil;**
- **Cazul mediu;**
- **Cazuri situate intre cazul cel mai favorabil si cazul cel mai nefavorabil.**

Este dificil de calculat complexitatea statistica a **tuturor** cazurilor.

In analiza algoritmilor se considera **relevant** calculul pentru **cazul cel mai nefavorabil (O)**.

Tipuri de complexitati

Cele mai intalnite cazuri pentru complexitatea algoritmilor sunt date de urmatoarele clase de complexitate:

- **n**: complexitatea liniara $O(n)$;
- **Log(n)**: complexitate logaritmica $O(\log(n))$
- **n^a**: complexitate polinomiala ($a > 1$). Exemplu: $O(n^2)$.
- **aⁿ**: complexitate exponentiala
- **n!**: complexitate factoriala.

In mod normal, daca gasim mai multi algoritmi cu complexitati diferite pentru rezolvarea unei clase de probleme, **il vom alege pe cel care are ordinul de complexitate cel mai mic.**

Tipuri de complexitati

Cele mai intalnite cazuri pentru complexitatea algoritmilor sunt date de urmatoarele clase de complexitate:

- **n**: complexitatea liniara $O(n)$;
- **Log(n)**: complexitate logaritmica $O(\log(n))$
- **n^a**: complexitate polinomiala ($a > 1$). Exemplu: $O(n^2)$.
- **aⁿ**: complexitate exponentiala
- **n!**: complexitate factoriala.

Pentru **n** cu valori suficient de mari, au loc relatiile:

$$\log(n) < n < n \cdot \log(n) < n^2 < n^3 < 2^n < 3^n$$

Deci:

$$O(\log(n)) < O(n) < O(n \cdot \log(n)) < O(n^2) < O(n^3) < O(2^n) < O(3^n)$$

Complexitate – ordine de marime

Evolutia **timpului de executie** in functie de **n** si de **ordinul de complexitate** .

- **n** : complexitatea liniara $O(n)$;
- **Log(n)** : complexitate logaritmica $O(\log(n))$
- **n^a** : complexitate polinomiala ($a > 1$). Exemplu: $O(n^2)$.
- **aⁿ** : complexitate exponentiala
- **n!** : complexitate factoriala.

$$O(\log(n)) < O(n) < O(n \cdot \log(n)) < O(n^2) < O(n^3) < O(2^n) < O(3^n)$$

n	$\log_2(n)$	$n \cdot \log_2(n)$	n^2	2^n	$n!$
10	3.3	33	100	1024	3628800
100	6.6	664	10000	10^{30}	10^{157}
1000	10	9965	1000000	10^{301}	10^{2567}
10000	13	132877	100000000	10^{3010}	10^{35659}

Complexitate – moduri de estimare

Avem cateva reguli generale privind estimarea complexitatii, in functie de tipul de operatiune.

Structuri secventiale

Timpii de executie individuali se aduna, rezultand o complexitate $O(n)$.

Cicluri repetitive (for/loop/do/while)

Un ciclu repetitiv se estimeaza ca avand ordinul $O(n)$ deoarece timpul de executie este egal cu numarul de iteratii inmultit cu timpul de executie al instructiunilor din interiorul ciclului.

Cicluri repetitive imbricate

Timpul de executie al instructiunilor se inmulteste cu produsul numarului de iteratii ale tuturor ciclurilor. Daca avem 2 cicluri imbricate, putem estima complexitatea la $O(n^2)$.

Structuri decizionale (if/switch)

Timpul de executie = timpul de executie al testului **if** + maximul dintre timpii de rulare pe ramura **then** si ramura **else**.

Complexitate – exemple

- Parcurgerea diagonalei unei matrici patratice pentru a calcula suma elementelor de pe diagonala.

```
for(int i=0; i<n; i++)  
{  
    suma = suma + matrice[i,i];  
}
```

```
for(int i=0; i<n; i++)  
{  
    for(int j=0; j<n; j++)  
    {  
        if(i==j)  
            suma = suma + matrice[i,j];  
    }  
}
```

Complexitate – moduri de estimare

Determinarea empirica a complexitatii unui algoritm se poate face la momentul rularii sale, prin introducerea unor secvente/prelucrari de monitorizare.

Variabile contor

Sunt folosite atunci cand complexitatea se calculeaza/estimeaza in functie de numarul de executii ale unor operatii.

Utilizarea unor functii care returneaza timpul curent

Se apeleaza la aceasta metoda atunci cand masura eficientei este determinata de timpul fizic de executie. Eventual se pot folosi mai multe esantioane reprezentative si se pot calcula medii statistice.

Bibliografie recomandata

- Antti Laaksonen, “Competitive Programmer’s Handbook”, 2018, pp. 17-24

Recomandare video

<https://www.youtube.com/watch?v=IT6tYkLGgBQ>

Agenda

- Echipa **Algoritmi si Logica Programarii**
- Programa analitica
- Bibliografie recomandata
- Calculul notei finale
- Algoritmi – notiuni generale
- Complexitatea algoritmilor
- **Demo Eclipse + Java**

Demo Eclipse + cod accesare fisiere

- Instalare;
- Rulare;
- Afisare la consola;
- Debug;
- Accesare fisiere.

Accesare fisiere

- Generare numere aleatoare;
- Scriere in fisier a unei secvente de numere aleatoare;
- Citirea din fisier a listei de numere aleatoare

Import bibliotece

```
package prjFisiere;  
import java.io.BufferedWriter;  
import java.io.File;  
import java.io.FileNotFoundException;  
import java.io.FileWriter;  
import java.util.LinkedList;  
import java.util.List;  
import java.util.Scanner;
```

Functie pentru generare de numere aleatoare

```
public static int genereazaAleator(int maxim, int minim)
{
    return (int) (minim + Math.random()*(maxim-minim));
}
```

Math.random() genereaza un numar aleator cuprins intre 0 si 1.

Functie pentru scriere in fisier

```
public static void scrieInFisier(String caleFisier)
{
    try
    {
        int n = genereazaAleator(100, 0);

        int numarCurent;
        BufferedWriter scriitor = new BufferedWriter(new
FileWriter(caleFisier));

        while(n>0)
        {
            numarCurent=genereazaAleator(1000, 0);
            scriitor.write(Integer.toString(numarCurent));
            scriitor.newLine();
            n--;
        }

        scriitor.flush();
        scriitor.close();
    }
    catch(Exception eroare)
    {
        System.out.println("Eroare fisier: " + eroare.getMessage());
    }
}
```

Functie pentru citire din fisier

```
public static int[] citesteDinFisier(String caleFisier) throws FileNotFoundException
{
    int[] listaNumere;

    Scanner cititor;
    cititor=new Scanner(new File(caleFisier));
    List<String> lista = new LinkedList<>();
    while(cititor.hasNext())
    {
        lista.add(cititor.nextLine());
    }

    listaNumere = new int[lista.size()];
    for(int i=0;i<listaNumere.length;i++)
    {
        listaNumere[i] = Integer.parseInt(lista.get(i));
    }

    return listaNumere;
}
```

Functia main

```
public static void main(String[] args) {  
    System.out.println("" + genereazaAleator(1000, 0));  
    scrieInFisier("random.txt");  
    System.out.println("Am scris cu succes!");  
    try  
    {  
        int[] sursa = citesteDinFisier("random.txt");  
        for(int i=0;i<sursa.length;i++)  
            System.out.println("Numarul " + i + " este " + sursa[i]);  
    }  
    catch (Exception eroare) {  
        System.out.println("Eroare la citire din fisier: " + eroare.getMessage());  
    }  
}
```

Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

RECURSIVITATE

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- Recursivitatea directa
- Recursivitatea mutuala (indirecta)
- Analiza executiei apelurilor recursive
- Exemple de functii/probleme recursive
- Implementari Java

Agenda

- **Recursivitatea – concepte generale**
- Tipuri de recursivitate
- Recursivitatea directa
- Recursivitatea mutuala (indirecta)
- Analiza executiei apelurilor recursive
- Exemple de functii/probleme recursive
- Implementari Java

Recursivitatea – concepte generale

- In general, un fenomen este recursiv daca in definitia sa se face referire la el insusi.
- **O functie este recursiva** daca in definitia sa se intalneste **cel putin un apel catre ea insasi**.

Recursivitatea – concepte generale

- In programarea calculatoarelor, **recursivitatea** construiește o clasă de metode prin definirea câtorva **cazuri de baza** (adeseori doar unul singur) și prin descrierea regulilor ce **descompun cazurile complexe în unele mai simple**, până când se ajunge la caz(urile) de baza.

Recursivitatea – concepte generale

- **Recursivitatea** este o tehnica prin care se rezolva o problema cu ajutorul rezultatului obtinut prin rezolvarea unei aceleiasi probleme dar de dimensiune mai redusa.
- O functie recursiva se apeleaza pe ea insasi, dar cu un argument de un ordin mai mic.
- Trebuie acordata atentie sporita evolutiei argumentului pentru a nu se ajunge la cazuri invalide.

Recursivitatea – gluma recursiva? 😊

Daca inca nu ati inteles **recursivitatea**, studiat
recursivitatea. 😊

Agenda

- Recursivitatea – concepte generale
- **Tipuri de recursivitate**
- Recursivitatea directa
- Recursivitatea mutuala (indirecta)
- Analiza executiei apelurilor recursive
- Exemple de functii/probleme recursive
- Implementari Java

Tipuri de recursivitate

- **Recursivitatea directa:** apelul recursiv se realizeaza din functia invocata.
- **Recursivitatea mutuala (indirecta):** apelul recursiv se efectueaza prin intermediul mai multor functii care se apeleaza in mod circular.

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- **Recursivitatea directa**
- Recursivitatea mutuala (indirecta)
- Analiza executiei apelurilor recursive
- Exemple de functii/probleme recursive
- Implementari Java

Recurсивitatea directa

- **Recurсивitatea directa:** apelul recursiv se realizeaza **direct din functia invocata**.
- Exemplu: Functia **f()** apeleaza in mod **direct** functia **f()**.

Recursivitatea directa

- Pentru ca executia sa se deruleze corect, apelurile recursive trebuie sa contina **obligatoriu o conditie de oprire** care sa determine la un moment dat oprirea apelurilor recursive si revenirea din sirul de apeluri.
- De regula, conditia de oprire face referire la caz(urile) de baza.

Recursivitatea directa

- Apelurile recursive sunt memorate in stiva de date impreuna cu parametrii lor de la momentul apelului.
- Fiecare apel recursiv suplimentar determina incarcarea **stivei din memorie**.
- Revenirea din apelul recursiv determina eliberarea stivei cu datele aferente acelui apel.

Recurсивitatea directa

Exemplu: scrierea pe verticala a cifrelor unui numar, folosind o functie recursiva.

```
public class Apeluri {  
    public static void main(String[] args) {  
        scrieVertical(1234);  
    }  
  
    public static void scrieVertical (int numar)  
    {  
        if( numar < 10 )  
            System.out.println (numar);  
        else  
        {  
            scrieVertical (numar / 10);  
            System.out.println (numar % 10);  
        }  
    }  
}
```

Conditia de oprire

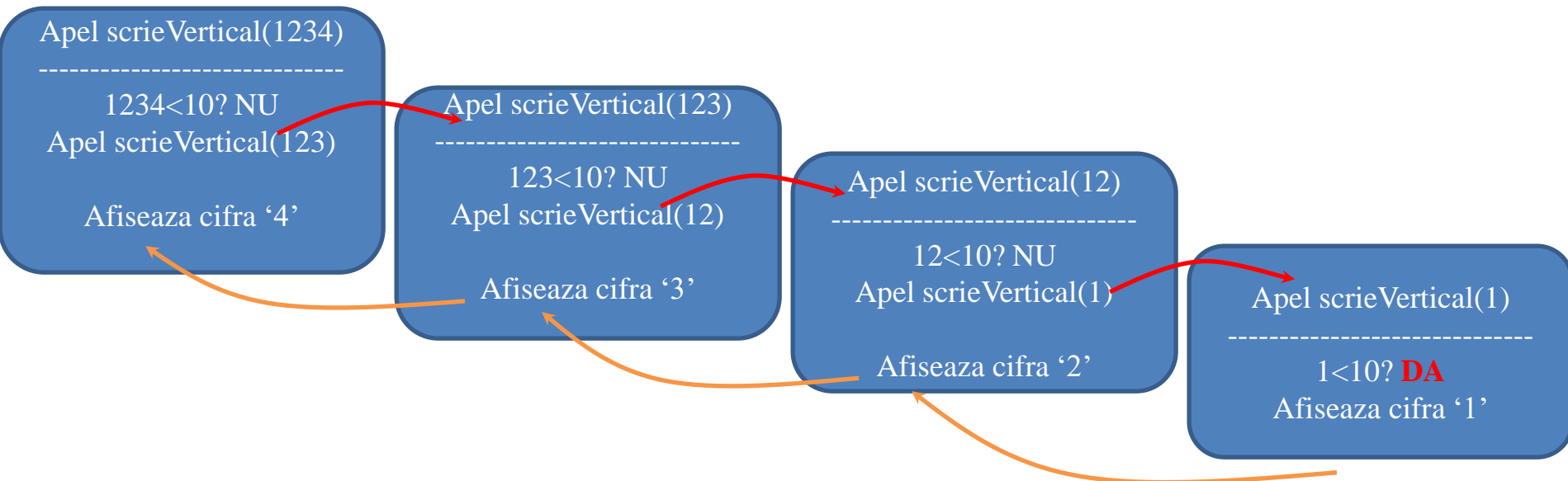
Apelul recursiv

Recursivitatea directa – ordine apeluri

```
public class Apeluri {  
    public static void main(String[] args) {  
        scrieVertical(1234);  
    }  
  
    public static void scrieVertical (int numar)  
    {  
        if( numar < 10 )  
            System.out.println (numar);  
        else  
        {  
            scrieVertical (numar / 10);  
            System.out.println (numar % 10);  
        }  
    }  
}
```

Conditia de oprire

Apelul recursiv



Recursivitatea directa – implementare JavaEclipse/IntelliJdea cu Debug pentru vizualizare stiva

```
public class Apeluri {  
    public static void main(String[] args) {  
        scrieVertical(1234);  
    }  
  
    public static void scrieVertical (int numar)  
    {  
        if( numar < 10 )  
            System.out.println (numar);  
        else  
        {  
            scrieVertical (numar / 10);  
            System.out.println (numar % 10);  
        }  
    }  
}
```

Conditia de oprire

Apelul recursiv

Recursivitatea directa

Observatii importante:

- Parametrii functiei curente precum si variabilele sale locale sunt salvate in stiva **inainte** de inceperea apelului recursiv.
- La intoarcerea din apel, se reface contextul apelului precedent (parametri + variabile) folosind stiva.
- Instructiunea de dupa apelul recursiv se executa **dupa** ce s-a revenit din apelul recursiv.

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- Recursivitatea directă
- **Recursivitatea mutuală (indirectă)**
- Analiza execuției apelurilor recursive
- Exemple de funcții/probleme recursive
- Implementări Java

Recurсивitate mutuala (indirecta)

- O functie **A** este indirect recursiva daca ea contine un apel la o functie **B** care la randul sau contine un apel la functia **A**.
- In acest caz, functiile A si B se numesc **mutual recursive**.

Recursivitate mutuala – model

Funcția A

```
{  
    ...  
    B(); //apel la funcția B  
    ...  
}
```

Funcția B

```
{  
    ...  
    A(); //apel la funcția A  
    ...  
}
```

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- Recursivitatea directa
- Recursivitatea mutuala (indirecta)
- **Analiza executiei apelurilor recursive**
- Exemple de functii/probleme recursive
- Implementari Java

Analiza apelurilor recursive

- Ce se intampla la executia functiei urmatoare?

```
int functiaMea()  
{  
    return functiaMea();  
}
```

Raspuns:

- executie teoretic infinita.
- practic, se va depasi stiva de date.

Analiza apelurilor recursive

- Ce se afiseaza pentru apelul **f(5)**?

```
public static void f(int n) {  
    System.out.println (n);  
    if (n > 1)  
        f(n-1);  
}
```

Analiza apelurilor recursive

- Ce se afiseaza pentru apelul **f(5)**?

```
public static void f(int n) {  
    if (n > 1)  
        f(n-1);  
    System.out.println (n);  
}
```


Analiza apelurilor recursive

- Ce se afiseaza pentru apelul **f(5)**?

```
public static void f(int n) {  
    System.out.println (n);  
    if (n > 1)  
        f(n-1);  
    System.out.println (n);  
}
```

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- Recursivitatea directă
- Recursivitatea mutuală (indirectă)
- Analiza execuției apelurilor recursive
- **Exemple de funcții/probleme recursive**
- Implementări Java

Exemple de functii/probleme recursive

- Suma primelor n numere naturale
- Factorial(n)
- Sirul lui Fibonacci
- Turnurile din Hanoi
- Sortari recursive
- Cautari recursive

Exemple de functii/probleme recursive

- Suma primelor **n** numere naturale are urmatoarea idee:
 - Daca $n==0$, atunci $\text{suma}=0$; (caz de baza)
 - Daca $n==1$, atunci $\text{suma}=1$; (caz de baza)
 - Daca $n>1$, atunci suma este egala cu **n plus suma precedentelor n-1 numere**, adica **$n+\text{suma}(n-1)$**

```
public static int sumaN(int n)
{
    if (n==0)
        return 0;
    if (n==1)
        return 1;
    return n+sumaN(n-1);
}
```

Exemple de functii/probleme recursive

- **Factorial(n)** are urmatoarea idee:
 - Daca $n==0$, atunci $\text{factorial}=1$; (caz de baza)
 - Daca $n==1$, atunci $\text{factorial}=1$; (caz de baza)
 - Daca $n>1$, atunci factorial este egal cu **n inmultit cu produsul precedentelor $n-1$ numere**, adica **$n * \text{factorial}(n-1)$**

```
public static int factorialN(int n)
{
    if (n==0)
        return 1;
    if (n==1)
        return 1;
    return n*factorialN(n-1);
}
```

Exemple de functii/probleme recursive

- **Sirul lui Fibonacci** se defineste prin urmatoarea relatie de recurenta:
 - $F(0) = 1$ (caz de baza);
 - $F(1) = 1$ (caz de baza);
 - $F(n) = F(n-1) + F(n-2)$, pentru $n \geq 2$

```
public static int Fibo(int n)
{
    if (n==0)
        return 1;
    if (n==1)
        return 1;
    return Fibo(n-1) + Fibo(n-2);
}
```

Exemple de functii/probleme recursive

- **Intrebare: ce se intampla la executia urmatoarei variante de cod? 😊**

```
public static int Fibo(int n)
{
    return Fibo(n-1) + Fibo(n-2);
    if (n==0)
        return 1;
    if (n==1)
        return 1;
}
```

Exemple de functii/probleme recursive

- Aflarea MAX dintr-un sir in varianta recursiva:

Solutia este data de relatia de recurenta:

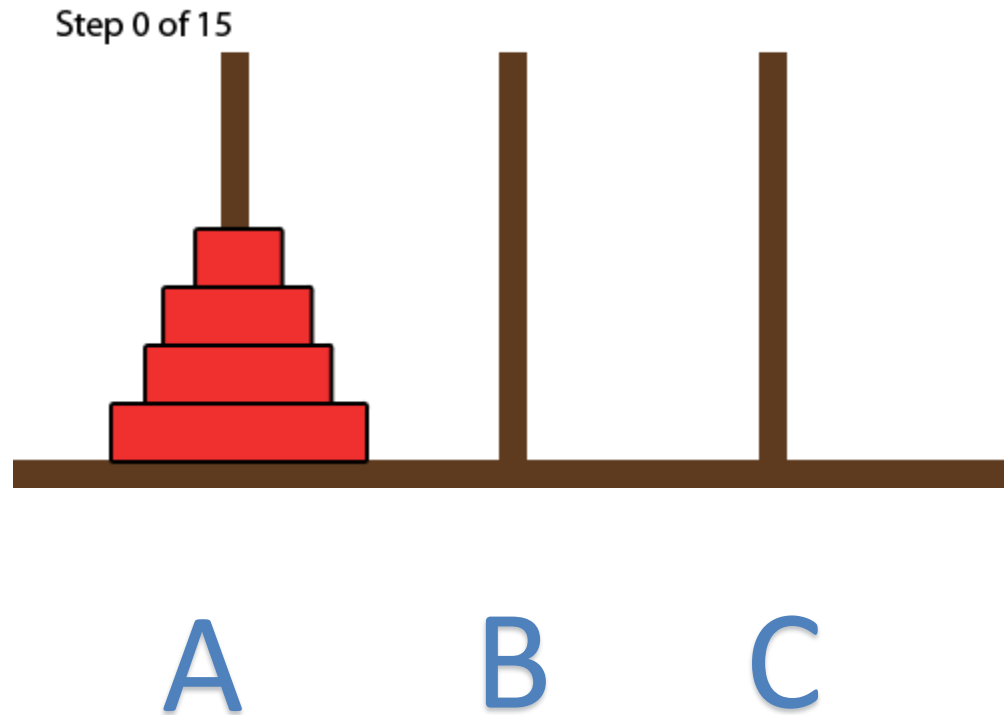
$$\max(a_1, a_2, \dots, a_n) = \max(a_n, \max(a_1, a_2, \dots, a_{n-1}))$$

Exemple de functii/probleme recursive

Turnurile din Hanoi

- Fie trei tije verticale notate A,B,C.
- Pe tija A se gasesc asezate n discuri de diametre diferite, in ordinea crescatoare a diametrelor, privind de sus in jos.
- Initial, tijele B si C sunt goale.
- Sa se afiseze toate mutarile prin care discurile de pe tija A se muta pe tija C, in aceeaasi ordine, folosind ca tija de manevra B si respectand urmatoarele reguli:
 - La fiecare pas se muta un singur disc;
 - Un disc se poate aseza numai peste un disc cu diametrul mai mare.

Turnurile din Hanoi



Exemple de functii/probleme recursive

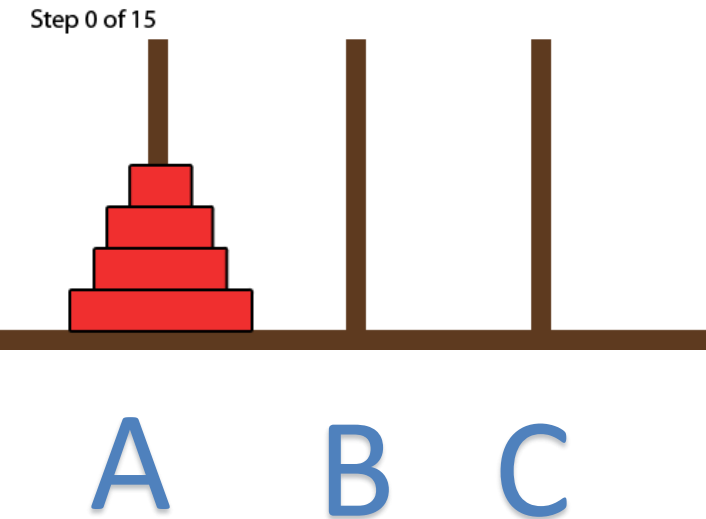
Turnurile din Hanoi – model de abordare

- Daca **$n == 1$** , atunci se muta discul de la 'A' la 'C' (caz de baza).
- Daca **$n > 1$** , atunci:
 - Se muta $n-1$ discuri de la 'A' la 'B';
 - Se muta discul **n** de la 'A' la 'C';
 - Se muta $n-1$ discuri de la 'B' la 'C'.

Turnurile din Hanoi – implementare Java

```
public class TurnuriHanoi {  
  
    public static void main(String[] args) {  
        System.out.println("Turnurile din Hanoi:");  
        Hanoi(4, 'A', 'C', 'B');  
    }  
  
    public static void Hanoi(int n, char from, char to, char auxiliar)  
    {  
        if (n==1)  
        {  
            System.out.println("Muta discul " + " 1 de la " + from + " la " + to);  
        }  
        else  
        {  
            Hanoi(n-1, from, auxiliar, to);  
            System.out.println("Muta discul " + n + " de la " + from + " la " + to);  
            Hanoi(n-1, auxiliar, to, from);  
        }  
    }  
}
```

Turnurile din Hanoi – rezultat pentru **n=4**



Turnurile din Hanoi:

```
Muta discul 1 de la A la B
Muta discul 2 de la A la C
Muta discul 1 de la B la C
Muta discul 3 de la A la B
Muta discul 1 de la C la A
Muta discul 2 de la C la B
Muta discul 1 de la A la B
Muta discul 4 de la A la C
Muta discul 1 de la B la C
Muta discul 2 de la B la A
Muta discul 1 de la C la A
Muta discul 3 de la B la C
Muta discul 1 de la A la B
Muta discul 2 de la A la C
Muta discul 1 de la B la C
```

Turnurile din Hanoi - complexitate

Complexitate:

- Pentru **n** discuri, sunt necesare **$2^n - 1$** mutari.
- Daca **n=10**, atunci numarul de mutari = $1024 - 1 = 1023$ mutari.
- Daca **n=64**, si pentru fiecare mutare ar fi necesara 1 secunda, atunci jocul se finalizeaza in 584.942.417.355 de **ani!!!**

Agenda

- Recursivitatea – concepte generale
- Tipuri de recursivitate
- Recursivitatea directa
- Recursivitatea mutuala (indirecta)
- Analiza executiei apelurilor recursive
- Exemple de functii/probleme recursive
- **Implementari Java**

Implementari Java

- **Implementati la curs si la laborator functiile recursive descrise anterior.**
- **Urmariti in Eclipse/IntelliJ stiva de executie si de apeluri recursive.**
- **Incercati sa estimati apelurile si valorile aferente INAINTE de a lansa in executie, apoi faceti comparatia cu situatia reala. Explicati-va eventualele diferente de perceptie.**

Meditatie

- **Esti singur(a) intr-o camera.**
- **Pe peretele din fata ta este o oglinda mare.**
- **Pe peretele din spatele tau este o oglinda mare.**
- **Te asezi cu fata la peretele din fata ta.**
- **Cate imagini de-ale tale se vad in oglinda din fata?**
- **Dar in cea din spate?**

Bibliografie recomandata

- **Jeff Erickson, Algorithms, 2015, pp. 1-4, Recursion Chapter**

Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

Sortarea

Sortarea se refera la procedeul rearanjarii elementelor dintr-o structura de date in ordine (de regula crescatoare).

Sortarea se poate aplica pentru valori numerice, siruri de caractere sau alte tipuri de valori care suporta un criteriu de comparatie.

Sortarea – exemplu

Structura initiala (nesortata)

10,20,5,50,4,6,3

Structura finala (**sortata**)

3,4,5,6,10,20,50

Algoritmi de sortare

- Bubble Sort (sortarea prin metoda bulelor);
- Insert Sort (sortarea prin insertie);
- Select Sort (sortarea prin selectie);
- Merge Sort (sortarea prin interclasare);
- QuickSort (sortarea rapida).

Algoritmi de sortare

- **Bubble Sort (sortarea prin metoda bulelor);**
- Insert Sort (sortarea prin insertie);
- Select Sort (sortarea prin selectie);
- Merge Sort (sortarea prin interclasare);
- QuickSort (sortarea rapida).

Bubble Sort

Bubble Sort (sortarea prin metoda bulelor) foloseste urmatoarea metodologie:

- Se parcurge vectorul si se compara fiecare element cu succesorul sau;
- Daca nu sunt in ordine (adica este o **inversiune**), cele 2 elemente se interschimba intre ele;
- Vectorul se parcurge de mai multe ori **pana cand se reuseste o parcurgere integrala fara nicio interschimbare**. Acest lucru inseamna ca vectorul este complet sortat.

Implementare Java BubbleSort (variante 1)

```
void sortareBubble_1(int [] v)
{
    bool ordonat=false;
    while(!ordonat)
    {
        ordonat=true; //presupunem ca sirul este ordonat
        for(int i=1;i<v.length;i++)
        {
            if(v[i]<v[i-1])
            {
                //avem inversiune
                //si interschimbam
                int aux;
                aux = v[i-1];
                v[i-1]=v[i];
                v[i]=aux;
                ordonat=false;
            }
        }
    }
}
```

Observatii BubbleSort varianta 1

- Fiecare iteratie parcurge intregul vector de valori.
- Acest lucru poate fi optimizat daca tinem cont de urmatorul detaliu:
 - La parcurgerea unei iteratii, dupa pozitia **ultimei interschimbari**, in mod **sigur** nu vom mai avea nicio inversiune la urmatoarea iteratie.
 - Ca urmare, poate fi mai avantajos ca iteratia curenta sa mearga nu pana la finalul vectorului, ci doar pana la **precedenta** ultima interschimbare.

Implementare Java BubbleSort (varianta 2)

```
//functia pentru sortarea de tip BUBBLE_2
//parcurgem vectorul si verificam eventualele inversiuni existente
//de asemenea, tinem cont ca dupa fiecare parcurgere, elementele de DUPA ultima inversiune sunt deja pe pozitiile lor finale
void sortareBubble_2(int[] v)
{
    int ultim=v.length;
    int ultimaInterschimbare=0;
    bool ordonat = false;
    while(!ordonat)
    {
        ordonat=true; //presupunem ca vectorul este ordonat inainte de iteratia curenta
        for(int i=1;i<ultim;i++)
        {
            if(v[i]<v[i-1])
            {
                //avem inversiune si inter schimbam
                int aux;
                aux=v[i-1];
                v[i-1]=v[i];
                v[i]=aux;
                ordonat=false;
                //memoram pozitia interschimbării
                ultimaInterschimbare=i;
            }
        }
        ultim=ultimaInterschimbare;
    }
}
```

Cod complet BubbleSort varianta 1 (pentru laborator)

```
#include <iostream>
using namespace std;
//declara functii
void afisareVector(int v[], int marimeVector);
void sortareBubble_1(int v[], int lungime);
//definire functii
//functia main
int main()
{
    //declar un vector initial cu valori neordonate
    int valori[7]={10,20,5,50,4,6,3};
    int marime = sizeof(valori) / sizeof(valori[0]);
    cout<<"Vectorul initial\n";
    afisareVector(valori, marime);
    cout<<"Sortez vectorul\n";
    sortareBubble_1(valori, marime);
    cout<<"vectorul sortat\n";
    afisareVector(valori, marime);
    return 0;
}
//functia pentru afisarea valorilor dintr-un vector
void afisareVector(int v[], int marimeVector)
{
    //functie pentru afisarea valorilor dintr-un vector
    string afisaj= "Vector=";
    for(int i=0;i<marimeVector;i++)
    {
        afisaj = afisaj + std::to_string(v[i]) + " ";
    }
    cout<<afisaj;
}
//functia pentru sortarea de tip BUBBLE_1
//parcurem vectorul si verificam eventualele inversiuni existente
void sortareBubble_1(int v[], int lungime)
{
    bool ordonat=false;
    while(!ordonat)
    {
        ordonat=true; //presupunem ca sirul este ordonat
        for(int i=1;i<lungime;i++)
        {
            if(v[i]<v[i-1])
            {
                //avem inversiune
                //si interschimbam
                int aux;
                aux = v[i-1];
                v[i-1]=v[i];
                v[i]=aux;
                ordonat=false;
            }
        }
    }
}
```

Cod complet BubbleSort varianta 2 (pentru laborator)

```
#include <iostream>
using namespace std;
//declarare functii
void afisareVector(int v[], int marimeVector);
void sortareBubble_2(int v[], int lungime);
//definire functii
//functia main
int main()
{
    //declar un vector initial cu valori neordonate
    int valori[7]={10,20,5,50,4,6,3};
    int marime = sizeof(valori) / sizeof(valori[0]);
    cout<<"Vectorul initial\n";
    afisareVector(valori, marime);
    cout<<"Sortez vectorul\n";
    sortareBubble_2(valori, marime);
    cout<<"Vectorul sortat\n";
    afisareVector(valori, marime);
    return 0;
}

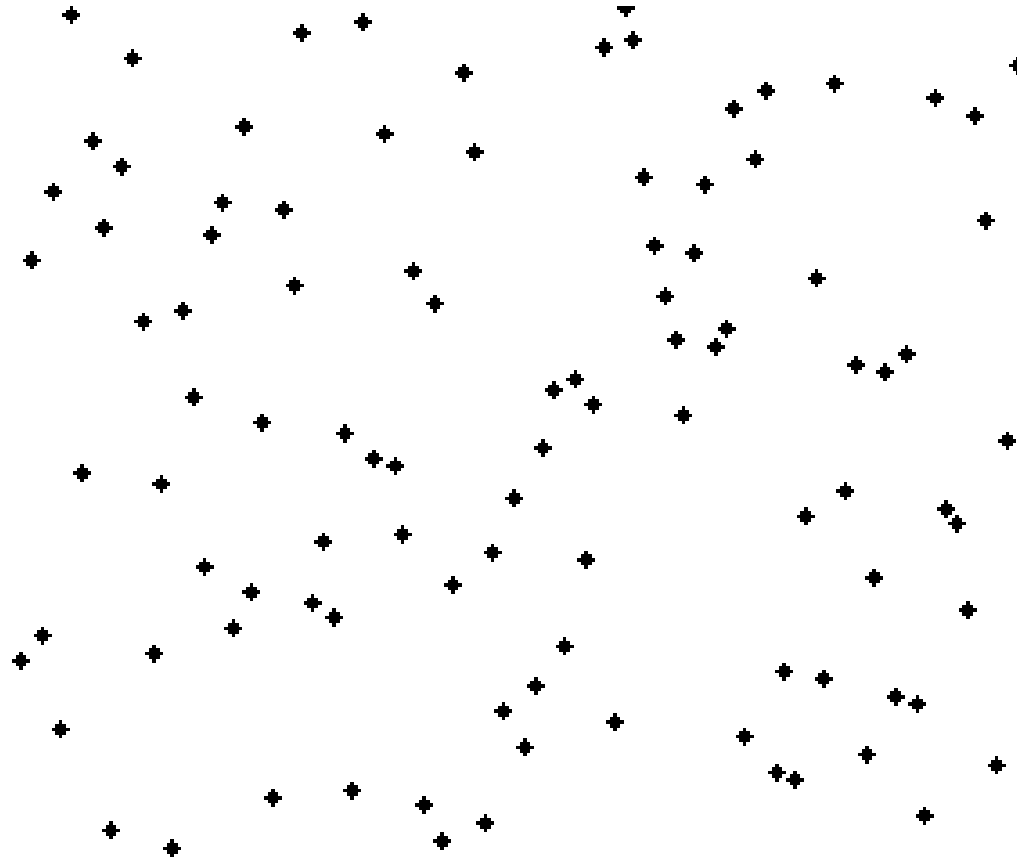
//functia pentru afisarea valorilor dintr-un vector
void afisareVector(int v[], int marimeVector)
{
    //functie pentru afisarea valorilor dintr-un vector
    string afisaj= "Vector=";
    for(int i=0;i<marimeVector;i++)
    {
        afisaj = afisaj + std::to_string(v[i]) + " ";
    }
    cout<<afisaj;
}

//functia pentru sortarea de tip BUBBLE_2
//parcurem vectorul si verificam eventualele inversiuni existente
//de asemenea, tinem cont ca dupa fiecare parcurgere, elementele de DUPA ultima inversiune sunt deja pe pozitiile lor finale
void sortareBubble_2(int v[], int lungime)
{
    int ultim=lungime;
    int ultimaInterschimbare=0;
    bool ordonat = false;
    while(!ordonat)
    {
        ordonat=true; //presupunem ca vectorul este ordonat inainte de iteratia curenta
        for(int i=1;i<ultim;i++)
        {
            if(v[i]<v[i-1])
            {
                //avem inversiune si interschimbam
                int aux;
                aux=v[i-1];
                v[i-1]=v[i];
                v[i]=aux;
                ordonat=false;
                //memoram pozitia interschimbarii
                ultimaInterschimbare=i;
            }
        }
        ultim=ultimaInterschimbare;
    }
}
```

Model rulare **BubbleSort** varianta 2

6 5 3 1 8 7 2 4

Model rulare **BubbleSort** varianta 2 –
partea “finala” a vectorului e cea care
se sorteaza mai intai



Bibliografie **BubbleSort**

- GeeksForGeeks, disponibile a:

<https://www.geeksforgeeks.org/bubble-sort/>

Algoritmi de sortare

- Bubble Sort (sortarea prin metoda bulelor);
- **Insert Sort (sortarea prin insertie);**
- Select Sort (sortarea prin selectie);
- Merge Sort (sortarea prin interclasare);
- QuickSort (sortarea rapida).

Insert Sort

Insert Sort (sortarea prin insertie) foloseste urmatoarea metodologie:

- Traversam toate elementele unul cate unul.
- Elementul curent este inserat pe **pozitia corecta** in **subsirul deja ordonat**.
- In acest fel, toate elementele deja procesate sunt in mod sigur in ordinea corecta.
- Dupa traversarea intregului sir, toate elementele sunt sortate.

Implementare Java Insert Sort

```
//functia pentru sortarea de tip INSERT SORT
void sortareInsertie(int v[])
{
    for(int i=1;i<v.length;i++)
    {
        int elementCurent=v[i];
        int pozitieCurenta=i-1;
        while(pozitieCurenta>=0 && elementCurent<v[pozitieCurenta])
        {
            v[pozitieCurenta+1]=v[pozitieCurenta];
            pozitieCurenta--;
        }
        v[pozitieCurenta+1]=elementCurent;
    }
}
```

Explicatii Insert Sort

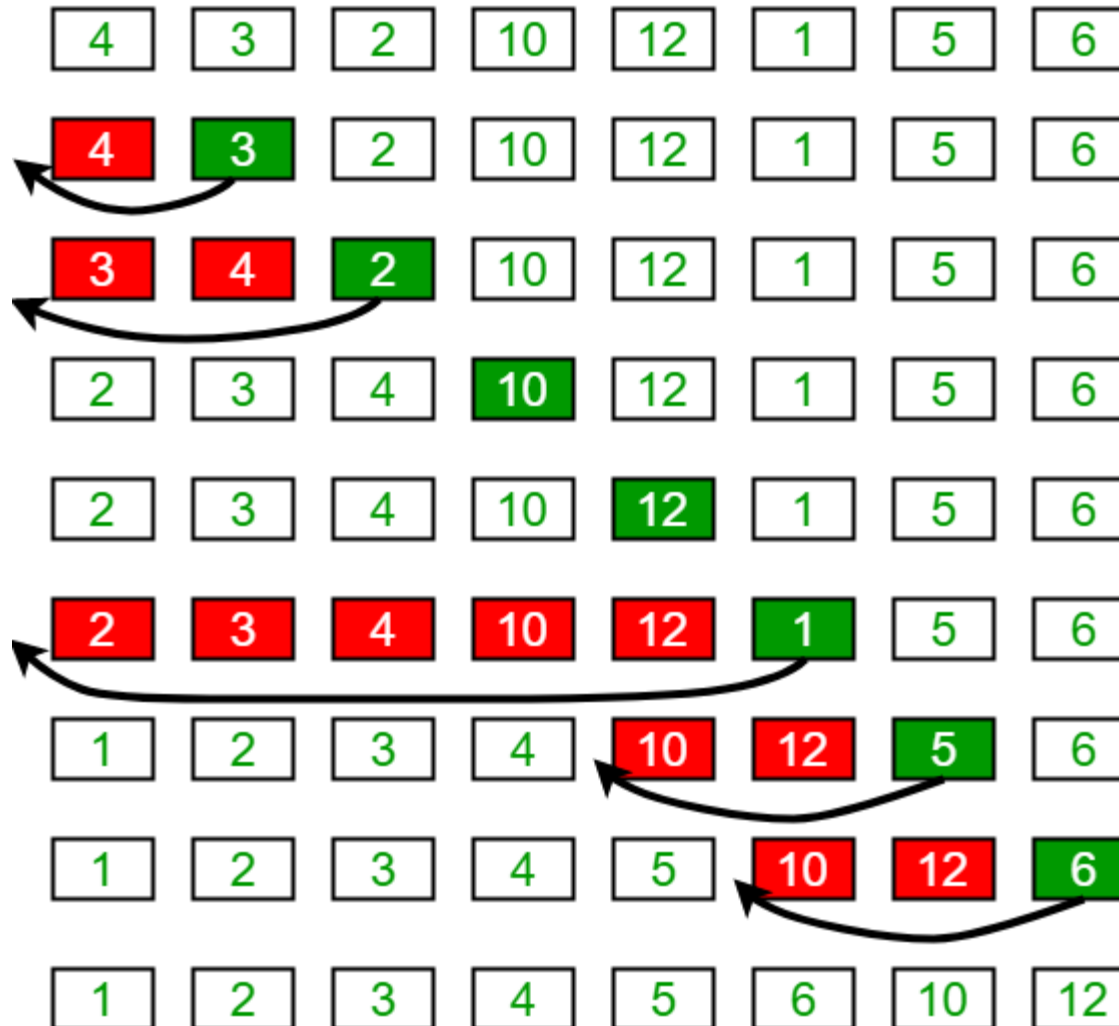
```
//functia pentru sortarea de tip INSERT SORT
void sortareInsertie(int v[])
{
    for(int i=1;i<v.length++)
    {
        int elementCurent=v[i];
        int pozitieCurenta=i-1;
        while(pozitieCurenta>=0 && elementCurent<v[pozitieCurenta])
        {
            v[pozitieCurenta+1]=v[pozitieCurenta];
            pozitieCurenta--;
        }
        v[pozitieCurenta+1]=elementCurent;
    }
}
```

- **elementCurent** este iteratorul cu care tot avansam spre “dreapta” sirului;
- se presupune ca intervalul cuprins intre *zero* si *elementCurent* este deja sortat;
- ca urmare, trebuie sa gasim in acest interval pozitia corecta pentru *elementCurent*;
- **pozitieCurenta** se determina prin “deplasarea spre stanga” sirului in functie de cat de mult trebuie deplasat **elementCurent** in subsirul deja sortat din stanga;
- pentru fiecare “deplasare spre stanga” a **pozitieiCurente**, valorile din subsirul din stanga se deplaseaza cu cate o pozitie spre dreapta pentru “a face loc” elementului curent.

Model ruleare InsertSort

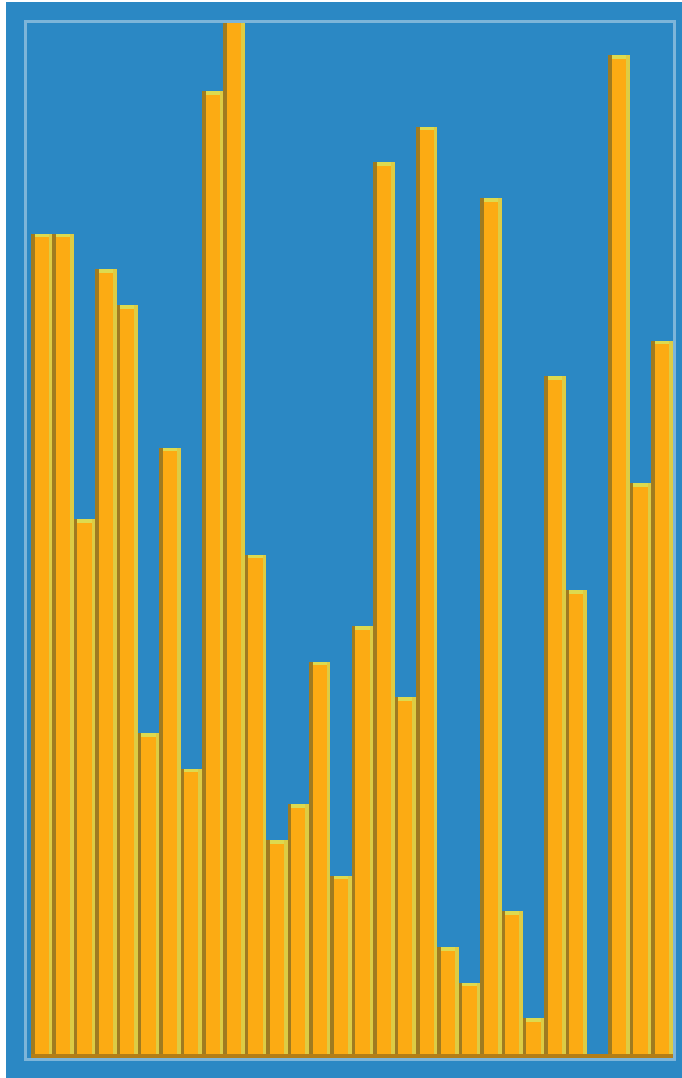
Sursa: <https://cdncontribute.geeksforgeeks.org/wp-content/uploads/insertionsort.png>

Insertion Sort Execution Example



Model rulare InsertSort

https://en.wikipedia.org/wiki/Insertion_sort#/media/File:Insertion_sort.gif



Model rulare InsertSort

https://en.wikipedia.org/wiki/Insertion_sort#/media/File:Insertion-sort-example-300px.gif

6 5 3 1 8 7 2 4

Bibliografie **Insert Sort**

- George T. Heineman, Gary Pollice & Stanley Selkow, “Algorithms in a Nutshell”, 2nd Edition, O’Reilly, 2016, pp. 57-60

Algoritmi de sortare

- Bubble Sort (sortarea prin metoda bulelor);
- Insert Sort (sortarea prin insertie);
- **Select Sort (sortarea prin selectie);**
- Merge Sort (sortarea prin interclasare)
- QuickSort (sortarea rapida).

Select Sort

Select Sort (sortarea prin selectie) foloseste urmatoarea metodologie:

- Se determina pozitia elementului cu **valoarea minima** din sir;
- Acel element **se va interschimba cu primul element**;
- Procedetul **se repeta pentru subsirul ramas**, pana cand mai ramane doar elementul maxim.

Implementare Java Select Sort

```
void sortareSelectie(int v[])
{
    for(int i=0;i<v.length-1;i++)
    {
        //selectam minimul din restul sirului
        //adica din sirul "din dreapta"
        int minPoz=i;
        for(int j=i+1;j<v.length;j++)
        {
            if(v[j]<v[minPoz])
                minPoz=j;
        }
        //mutam minimul pe pozitia curenta
        if(minPoz != i)
        {
            int aux=v[minPoz];
            v[minPoz]=v[i];
            v[i]=aux;
        }
    }
}
```

Explicatii Select Sort

```
void sortareSelectie(int v[])
{
    for(int i=0;i<v.length-1;i++)
    {
        //selectam minimul din restul sirului
        //adica din sirul "din dreapta"
        int minPoz=i;
        for(int j=i+1;j<lv.length;j++)
        {
            if(v[j]<v[minPoz])
                minPoz=j;
        }
        //mutam minimul pe pozitia curenta
        if(minPoz != i)
        {
            int aux=v[minPoz];
            v[minPoz]=v[i];
            v[i]=aux;
        }
    }
}
```

- algoritmul **Select Sort** este un algoritm lent;
- acest algoritm nu “invata” nimic de la o iteratie la alta;
- selectarea celui mai mic element (min) necesita **n-1** comparatii, iar selectarea celui de-al doilea cel mai mic element necesita **n-2** comparatii, adica un progres nu prea notabil.
- algoritmul **Select Sort** poate fi implementat in maniera similara abordand varianta **max** in loc de **min**.

Model rulare **Select Sort**

Sursa: https://en.wikipedia.org/wiki/Selection_sort#/media/File:Selection-Sort-Animation.gif

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Bibliografie **Select Sort**

- George T. Heineman, Gary Pollice & Stanley Selkow, “Algorithms in a Nutshell”, 2nd Edition, O’Reilly, 2016, p. 61

Algoritmi de sortare

- Bubble Sort (sortarea prin metoda bulelor);
- Insert Sort (sortarea prin insertie);
- Select Sort (sortarea prin selectie);
- **Merge Sort (sortarea prin interclasare);**
- QuickSort (sortarea rapida).

Sortarea **Merge Sort**

Merge Sort foloseste notiunea de **interclasare**.

Interclasarea presupune “combinarea” a 2 subsiruri sortate pentru a obtine un singur sir sortat complet.

Interclasarea

- Avand la dispozitie 2 vectori (A,B), cu elemente **ordonate** trebuie sa construim un al 3-lea vector (C) care sa contina elementele primelor 2, respectand acelasi criteriu de ordonare.
- **La fiecare pas:**
 - comparam elementul de indice "i" din A cu elementul de indice "j" din B si il alegem pe cel mai mic pentru a fi adaugat la vectorul C, pe pozitia "k"
 - dupa fiecare adaugare, "k" creste cu 1 si tot cu 1 creste si indicele corespunzator vectorului din care am ales elementul (daca am adaugat la C elementul din A, atunci creste "i", iar daca am adaugat elementul din B, creste "j")
 - acesti pasi ii repetam atata timp cat mai avem elemente neadaugate in ambii vectori ($i \leq n$ si $j \leq m$), deci cat timp se justifica compararea valorilor.

Interclasarea

- Ne oprim cu aceste operatii in momentul in care am adaugat deja toate elementele unui vector (de exemplu A), insa interclasarea nu se termina aici, pentru ca au ramas elemente neadaugate din celalalt vector (de exemplu B).
- Profitand de faptul ca vectorul este ordonat crescator **nu trebuie decat sa copiem elementele ramase in vectorul C.**

Interclasarea - algoritmul

```
k=-1; i=0; j=0;
while(i<n && j<m)
    if(a[i]<b[j])
    {
        k++;
        c[k]=a[i];
        i++;
    }
    else
    {
        k++;
        c[k]=b[j];
        j++;
    }
if(i<n)
    for(l=i; l<n; l++)
    {
        k++;
        c[k]=a[l];
    }
else
    for(l=j; l<m; l++)
    {
        k++;
        c[k]=b[l];
    }
```

Interclasarea – implementare Java

```
public static int[] interclasare(int[] A, int[] B)
{
    int[] C;
    int n,m;
    n=A.length;
    m=B.length;
    C = new int[n+m];

    int i,j,K;
    i=0;j=0;K=-1;

    while(i<n && j<m)
    {
        //interclasez A cu B
        if(A[i]<B[j])
        {
            K++;
            C[K]=A[i];
            i++;
        }
        else
        {
            K++;
            C[K]=B[j];
            j++;
        }
    }

    //adaug in C restul elementelor ramase in A sau in B
    if(i<n)
    {
        //adaug in C restul elementelor din A
        for(int l=i;l<n;l++)
        {
            K++;
            C[K]=A[l];
        }
    }
    else
    {
        //adaug in C restul elementelor din B
        for(int l=j;l<m;l++)
        {
            K++;
            C[K]=B[l];
        }
    }

    //la final returnez vectorul C care contine interclasarea dintre A si B
    return C;
}
```

Interclasarea – o noua abordare

- O alta abordare a interclasarii porneste de la premisa ca 2 “jumatați” ale **aceluiasi vector** sunt sortate.
- Scopul este sa obtinem vectorul sortat complet.

Interclasarea – o noua abordare

- Exemplu: pentru vectorul $A[]$, consideram ca fiind importante următoarele repere:
 - **s**: indexul valorii aflate în partea stângă a vectorului;
 - **d**: indexul valorii aflate în partea dreaptă a vectorului;
 - **m**: indexul valorii aflate la mijlocul vectorului;
 - porțiunea **$A[s..m]$** este sortată;
 - porțiunea **$A[m+1..d]$** este sortată.

Interclasarea – implementare Java

```
public void merge(int[] a, int s, int m, int d)
{ // Merge a[s..m] cu a[m+1..d].
    int i,j,k;
    int n1=m-s+1; //dimensiune vector "stanga"
    int n2=d-m; //dimensiune vector "dreapta"
    //creare vectori temporari
    int[] S= new int[n1];
    int[] D= new int[n2];
    //copiem datele in vectorii partiali
    for(i=0;i<n1;i++)
        S[i]=a[s+i];

    for(j=0;j<n2;j++)
        D[j]=a[m+1+j];

    //interclasez
    i=0;j=0;k=s-1;
    while(i<n1 && j<n2)
    {
        if(S[i]<D[j])
        {
            k++;
            a[k]=S[i];
            i++;
        }
        else
        {
            k++;
            a[k]=D[j];
            j++;
        }
    }
    while(i<n1)
    {
        k++;
        a[k]=S[i];
        i++;
    }

    while(j<n2)
    {
        k++;
        a[k]=D[j];
        j++;
    }
}
```

Sortarea Merge Sort

MergeSort foloseste strategia divide-et-impera.

1.MergeSort divide vectorul initial in 2 vectori.

2.MergeSort se apeleaza pe ea insasi pentru fiecare dintre cei 2 vectori.

3.Apoi interclaseaza (merge) cei 2 vectori sortati.

Pentru interclasare se foloseste functia **merge** (vezi slide precedent).

Sortarea Merge Sort

MergeSort(a[], s, d)

if $d > s$

1. Gaseste punctul de mijloc pentru a diviza vectorul in 2 portiuni:

$$m = (s+d)/2.$$

2. Apeleaza mergeSort pentru prima jumatate:

mergeSort(a, s, m)

3. Apeleaza mergeSort pentru a doua jumatate:

mergeSort(a, m+1, d)

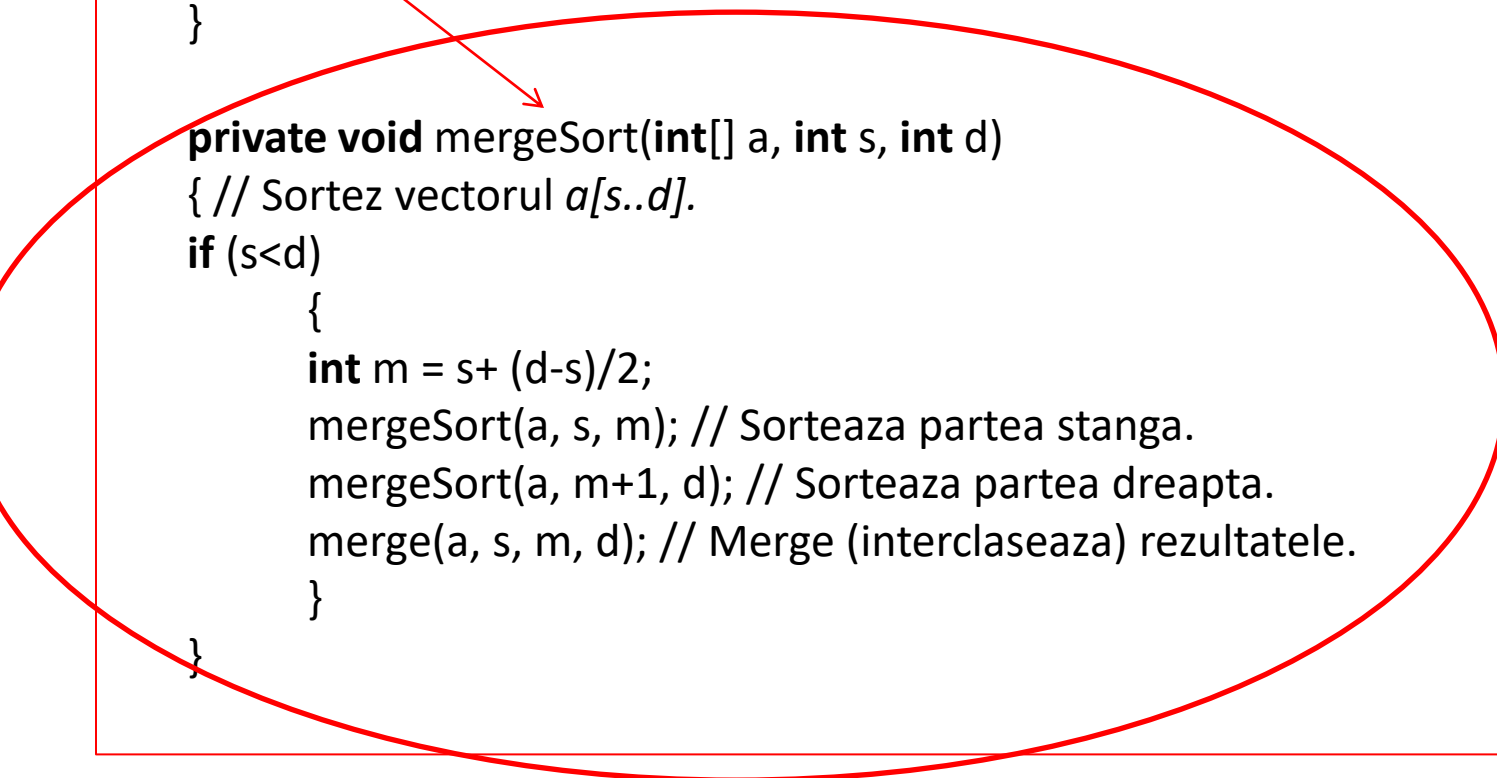
4. Interclaseaza (merge) cele 2 jumatati sortate in pasii 2 si 3:

merge(a, s, m, d)

MergeSort – implementare Java

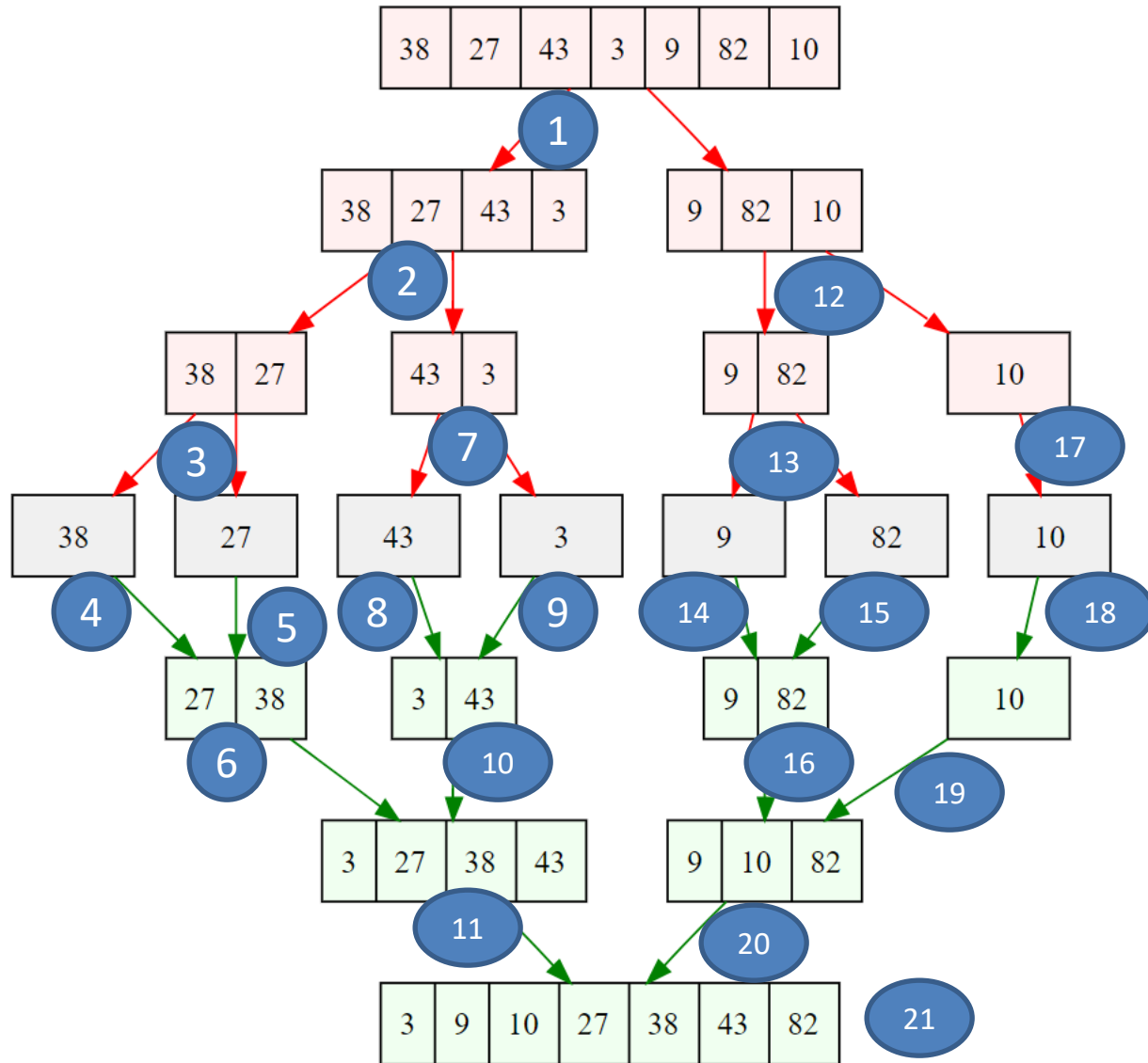
```
public void sort(int[] a)
{
    mergeSort(a, 0, a.length-1);
}

private void mergeSort(int[] a, int s, int d)
{ // Sortez vectorul a[s..d].
  if (s<d)
  {
    int m = s+ (d-s)/2;
    mergeSort(a, s, m); // Sorteaza partea stanga.
    mergeSort(a, m+1, d); // Sorteaza partea dreapta.
    merge(a, s, m, d); // Merge (interclaseaza) rezultatele.
  }
}
```



MergeSort – model de rulare

Sursa: https://upload.wikimedia.org/wikipedia/commons/e/e6/Merge_sort_algorithm_diagram.svg



MergeSort – model de rulare

https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge-sort-example-300px.gif

6 5 3 1 8 7 2 4

Bibliografie **MergeSort**

- Robert Sedgewick, Kevin Wayne, “Algorithms”, 4nd Edition, Addison-Wesley, 2011, pp. 270-282

Algoritmi de sortare

- Bubble Sort (sortarea prin metoda bulelor);
- Insert Sort (sortarea prin insertie);
- Select Sort (sortarea prin selectie);
- Merge Sort (sortarea prin interclasare);
- **QuickSort (sortarea rapida).**

Sortarea QuickSort

QuickSort foloseste urmatoarea metodologie:

- alege o valoare **pivot** din vector;
- Pe baza valorii pivot, realizeaza 2 partitii ale vectorului initial, astfel incat:
 - in partitia din **stanga** sa fie toate **valorile mai mici** decat pivotul;
 - in partitia din **dreapta** sa fie toate **valorile mai mari** decat pivotul.

QuickSort - partitionarea

```
int partitie(int a[], int s, int d)
{
    //aceasta functie considera pivotul ca fiind ultimul element din lista
    //plaseaza pivotul in pozitia corecta in vectorul sortat
    //si plaseaza in stanga toate valorile mai mici decat pivotul
    //si in dreapta pivotului toate valorile mai mari decat pivotul
    int pivot = a[d]; //ultimul element este pivotul
    int i = s-1; //indexul celui mai din stanga element
    for(int j=s;j<d;j++)
    {
        //daca elementul curent este mai mic sau egal cu pivotul
        //il mutam in partea stanga a partitiei
        if(a[j] <= pivot)
        {
            i++;
            //mut in stanga (interschimb)
            int temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }
    //duc pivotul in pozitia finala din partitie
    int temp=a[i+1];
    a[i+1]=a[d];
    a[d]=temp;
    //returnez indexul partitiei
    return i+1;
}
```


QuickSort – apelul recursiv

```
void sort(int[] a, int s, int d)
{
    if(s<d)
    {
        //indexPartitie este obtinut astfel incat a[indexPartitie] este in pozitia corecta
        int indexPartitie = partitie(a, s, d);
        sort(a, s, indexPartitie-1);
        sort(a, indexPartitie+1, d);
    }
}
```

QuickSort – invocare algoritm

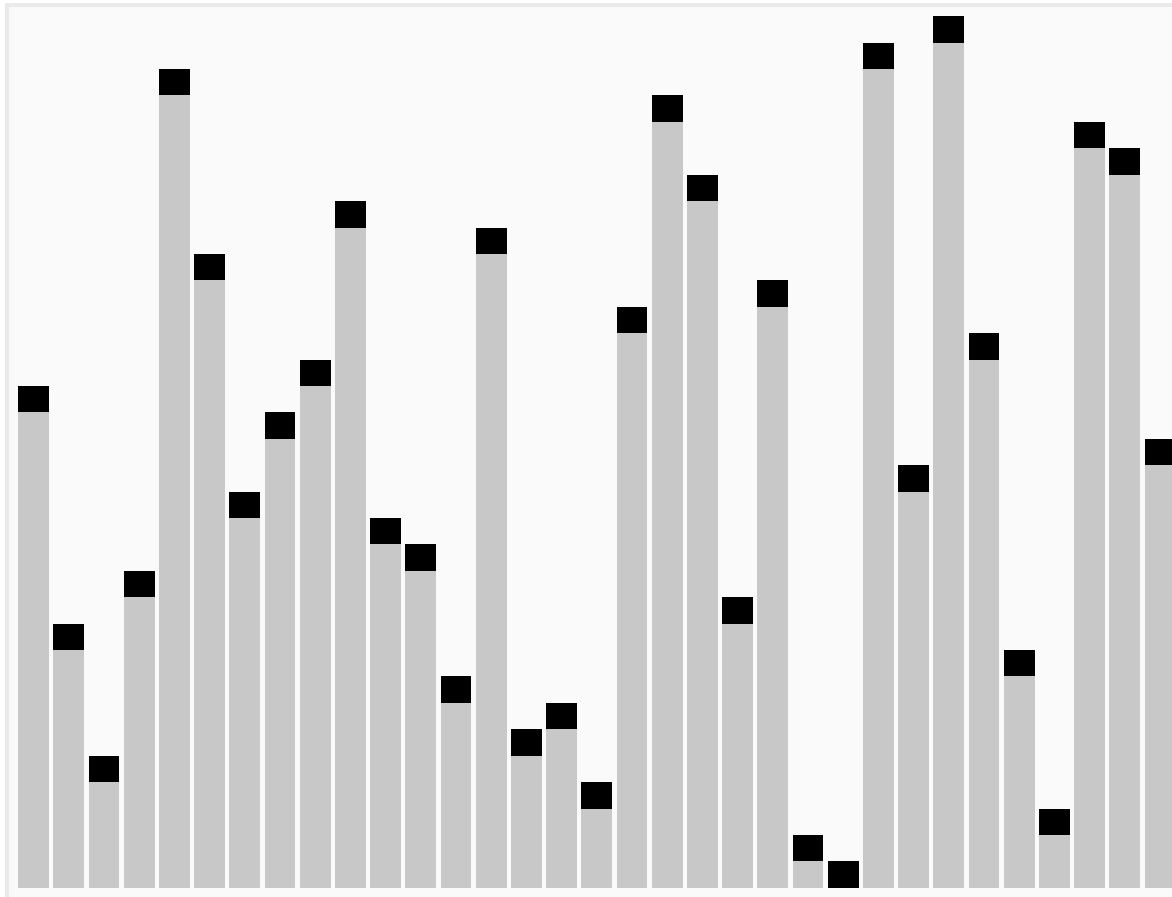
```
public class QuickSort {
```

```
    public static void main(String[] args) {  
        int[] a = new int[] {1,50,10,40,5,100,80,60};  
        int n= a.length-1;  
        QuickSort app = new QuickSort();  
        app.sort(a, 0, n);  
        System.out.println("Valorile sortate:");  
        afiseazaVector(a);  
    }
```

```
static void afiseazaVector(int[] a)  
{  
    for(int i=0;i<a.length;i++)  
        System.out.print("'" + a[i] + " ");  
    System.out.println();  
}
```

QuickSort – model rulare

Sursa: https://ro.wikipedia.org/wiki/Quicksort#/media/File:Sorting_quicksort_anim.gif



QuickSort – model rulare pentru prima partitionare

<https://www.cmprogrammers.com/post.php?id=53>



Bibliografie **QuickSort**

Thomas Cormen et. al., “Introduction to Algorithms”, 3rd Edition, MIT Press, 2009, pp. 170-180

Implementari laborator

- Cititi dintr-un fisier o lista de valori numerice;
- Implementati in Java algoritmi de sortare InsertSort, BubbleSort, MergeSort, QuickSort, SelectSort pentru a ordona valorile initiale;
- Masurati timpul necesar rularii fiecarui algoritm. Sugestie:

```
long start = System.currentTimeMillis();  
//apelare functie de sortare specifica;  
long finish = System.currentTimeMillis();  
long durata = finish - start;  
System.out.println("Durata = " + durata);
```

Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

Cautare / Search

Cautarea se refera la tehnica prin care o valoare este gasita intr-o colectie (multime) de valori.

Tipuri de cautari

- Cautarea secventiala
- Cautarea in multimi ordonate
- Cautarea divide-et-impera (maniera secventiala);
- Cautarea divide-et-impera (maniera recursiva);
- Cautarea in arbori binari.

Tipuri de cautari

- **Cautarea secventiala**
- Cautarea in multimi ordonate
- Cautarea divide-et-impera (maniera secventiala);
- Cautarea divide-et-impera (maniera recursiva);
- Cautarea in arbori binari.

Cautarea secventiala

- Cautarea secventiala (liniara) este o metoda de a gasi o anumita valoare intr-o lista.
- Metoda presupune verificarea fiecărei valori pas cu pas, pana cand este gasita valoarea dorita.
- Cautarea secventiala este cel mai simplu algoritm de cautare.

Cautarea secventiala

- Pentru o colectie cu “ n ” elemente, cazul cel mai favorabil este atunci cand valoarea cautata este egala cu primul element din colectie.
- Cazul cel mai nefavorabil este cand valoarea cautata nu se afla in colectie (sau se afla pe ultima pozitie in colectie), ceea ce inseamna ca sunt necesare “ n ” operatiuni de comparare.
- Cautarea secventiala nu necesita sortarea prealabila a colectiei.

Cautarea secventiala – implementare Java

```
public static int linearSearch(int[] v, int cheie){  
    //cautarea secventiala (liniara)  
    int size = v.length;  
    for(int i=0;i<size;i++){  
        if(v[i] == cheie){  
            //daca s-a gasit valoarea cautata  
            //se returneaza pozitia curenta  
            return i;  
        }  
    }  
    //daca nu s-a gasit valoarea cautata  
    //se returneaza indexul -1  
    return -1;  
}
```

Cautarea secventiala – apelare Java

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    System.out.println("Cautare secventiala");  
    int[] valori= {10,4,70,20,35,9,20,14,55};  
    int valCautata = 35;  
    System.out.println("Valoarea " + valCautata + " se gaseste la  
    pozitia " + linearSearch(valori, valCautata));  
}
```

Cautare secventiala

Valoarea 35 se gaseste la pozitia 4

Tipuri de cautari

- Cautarea secventiala
- **Cautarea in multimi ordonate**
- Cautarea divide-et-impera (maniera secventiala);
- Cautarea divide-et-impera (maniera recursiva);
- Cautarea in arbori binari.

Cautarea in multimi ordonate

- **Daca** stim ca multimea este **ordonata**, cautarea secventiala poate fi optimizata.
- Optimizarea are in vedere faptul ca in cazul in care valoarea cautata ajunge sa fie mai mica decat valoarea curenta din colectie, atunci cautarea **poate fi oprita** deoarece nu mai sunt sanse de a gasi valoarea cautata.

Cautarea in multimi ordonate – implementare Java

```
public static int linearSearchInSorted(int[] v, int cheie)
{
    int size = v.length;
    int i=0;
    //mergem "in dreapta" pana cand "cheie" este
    //mai mica decat valoarea curenta din colectie
    while (i<size && v[i]<=cheie)
    {
        i++;
    }
    //verificam daca s-a gasit cheia de cautare in colectie
    if(i==0 || v[i-1]!=cheie)
        return -1;
    else
        return i-1;
}
```

Cautarea in multimi ordonate – discutii

```
public static void main(String[] args) {  
    System.out.println("Cautare secventiala in multime ordonata");  
    int[] valori= {10,20,30,40,50,60,70,80,90};  
    int valCautata = 35;  
    System.out.println("Valoarea " + valCautata + " se gaseste la pozitia "  
+ linearSearchInSorted(valori, valCautata));  
}
```

- in cazul **valCautata=35**, procesul de cautare merge pana la **i=3** deoarece dupa acest index, valoarea 35 **sigur nu mai poate fi gasita**.

- Nu uitati!!! Acest algoritm functioneaza DOAR DACA stim sigur ca multimea initiala este sortata.

Tipuri de cautari

- Cautarea secventiala
- Cautarea in multimi ordonate
- **Cautarea divide-et-impera (maniera secventiala);**
- Cautarea divide-et-impera (maniera recursiva);
- Cautarea in arbori binari.

Cautarea Divide-et-Impera (secventiala)

- Cautarea divide-et-impair (binara) este un algoritm de cautare pe jumatati de intervale in cadrul unei colectii sortate.

Cautarea binara - metodologie

- La fiecare etapa, algoritmul compara **cheia de cautare** cu **valoarea din mijlocul colectiei**.
- Daca valorile sunt egale, inseamna ca s-a gasit elementul cautat si se returneaza indexul respectiv.
- In caz contrar, daca cheia de cautare este mai mica decat valoarea de la mijlocul colectiei, algoritmul se repeta in **sub-colectia din stanga mijlocului colectiei**.
- Daca cheia de cautare este mai mare, algoritmul se repeta in **sub-colectia din dreapta mijlocului**.
- Daca sub-colectia de cautare se reduce la zero, inseamna ca cheia de cautare nu a fost gasita.

Cautarea binara – implementare Java

```
public static int binarySearch(int[] a, int cheie) {  
  
    int start = 0;  
    int end = a.length - 1;  
    while (start <= end) {  
        int mijloc = (start + end) / 2;  
        if (cheie == a[mijloc]) {  
            return mijloc;  
        }  
        if (cheie < a[mijloc]) {  
            end = mijloc - 1;  
        } else {  
            start = mijloc + 1;  
        }  
    }  
    return -1;  
}
```

Cautarea binara – apelare Java

```
public static void main(String[] args) {  
    System.out.println("Cautare binara");  
    int[] valori= {10,20,30,40,50,60,70,80,90};  
    int valCautata = 80;  
    System.out.println("Valoarea " + valCautata + " se gaseste la  
pozitia " + binarySearch(valori, valCautata));  
}
```

Cautare binara

Valoarea 80 se gaseste la pozitia 7

Cautarea binara - remarci

- La fiecare etapa (iteratie), algoritmul elimina jumatate dintre valorile colectiei.
- Acest lucru face ca algoritmul de cautare binara sa fie foarte eficient, chiar si pentru colectii de dimensiuni foarte mari.
- Cautarea binara functioneaza **doar** pe colectii sortate.
- Complexitatea? 😊

Tipuri de cautari

- Cautarea secventiala
- Cautarea in multimi ordonate
- Cautarea divide-et-impera (maniera secventiala);
- **Cautarea divide-et-impera (maniera recursiva);**
- Cautarea in arbori binari.

Cautarea binara recursiva - metodologie

- La fiecare etapa, algoritmul compara cheia de cautare cu valoarea din mijlocul colectiei.
- Daca valorile sunt egale, inseamna ca s-a gasit elementul cautat si se returneaza indexul respectiv.
- In caz contrar, daca cheia de cautare este mai mica decat valoarea de la mijlocul colectiei, algoritmul se **re-apeleaza recursiv** avand ca date de intrare **sub-colectia din stanga mijlocului colectiei**.
- Daca cheia de cautare este mai mare, algoritmul se re-apeleaza recursiv cu **sub-colectia din dreapta mijlocului**.
- Marimea colectiei este ajustata prin manipularea indecsilor de inceput si de sfarsit ai colectiei.
- Daca sub-colectia de cautare se reduce la zero, inseamna ca cheia de cautare nu a fost gasita.

Cautarea binara recursiva – implementare Java

```
public static int recursiveBinarySearch(int[] a, int start, int end, int cheie) {  
  
    if (start < end) {  
        int mijloc = start + (end - start) / 2;  
        if (cheie < a[mijloc]) {  
            return recursiveBinarySearch(a, start, mijloc, cheie);  
        } else if (cheie > a[mijloc]) {  
            return recursiveBinarySearch(a, mijloc+1, end, cheie);  
        } else {  
            return mijloc;  
        }  
    }  
    //daca nu s-a gasit cheia de cautare, returnam index negativ  
    return -1;  
}
```

Cautarea binara recursiva – apelare Java

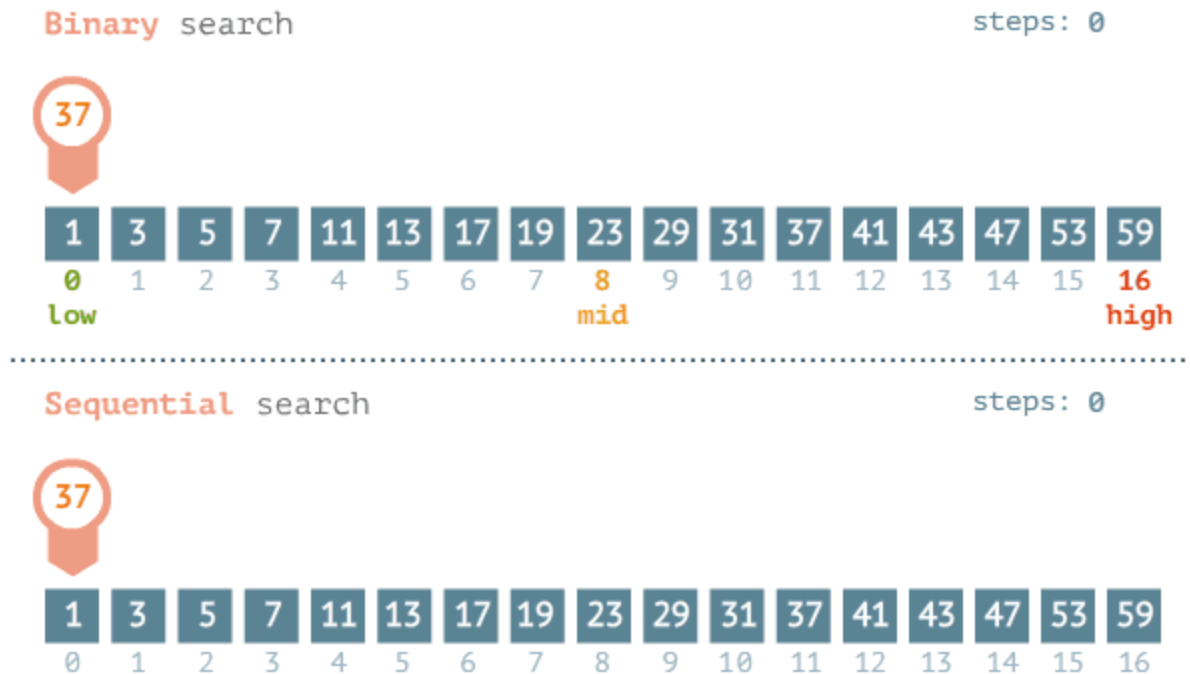
```
public static void main(String[] args) {  
    System.out.println("Cautare binara recursiva");  
    int[] valori= {10,20,30,40,50,60,70,80,90};  
    int valCautata = 30;  
    System.out.println("Valoarea " + valCautata + " se gaseste la  
pozitia " + recursiveBinarySearch(valori,0,valori.length,valCautata));  
}
```

Cautare binara recursiva

Valoarea 30 se gaseste la pozitia 2

Cautarea secventiala versus cautare binara – comparatie grafica

Sursa: <https://www.mathwarehouse.com/programming/gifs/binary-vs-linear-search.php>



Cautare secventiala si cautare binara - **bibliografie**

- George T. Heineman, Gary Pollice & Stanley Selkow, “Algorithms in a Nutshell”, 2nd Edition, O’Reilly, 2016, pp. 91-99

Tipuri de cautari

- Cautarea secventiala
- Cautarea in multimi ordonate
- Cautarea divide-et-impera (maniera secventiala);
- Cautarea divide-et-impera (maniera recursiva);
- **Cautarea in arbori binari.**

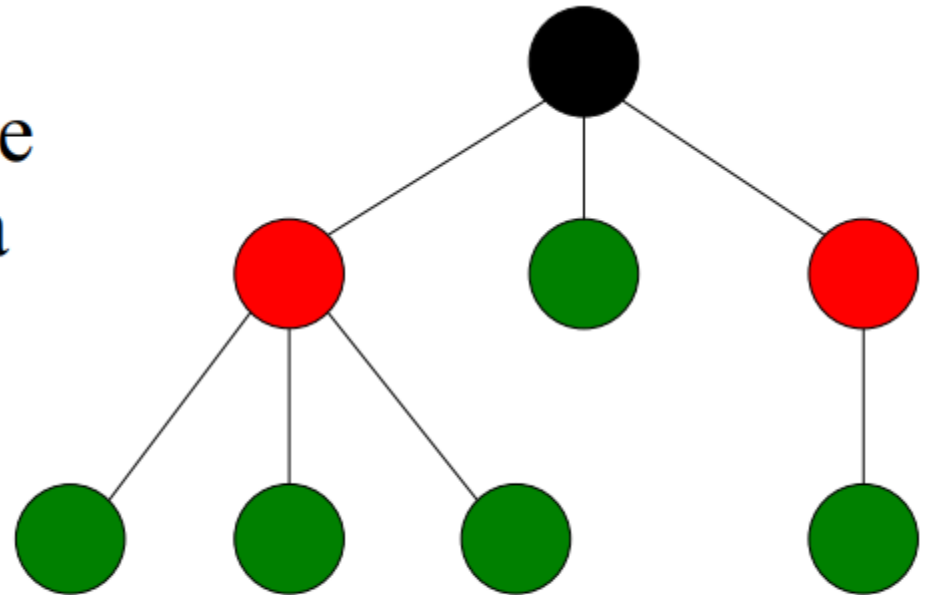
Arbori

Un **arbore** este un set finit de noduri pentru care:

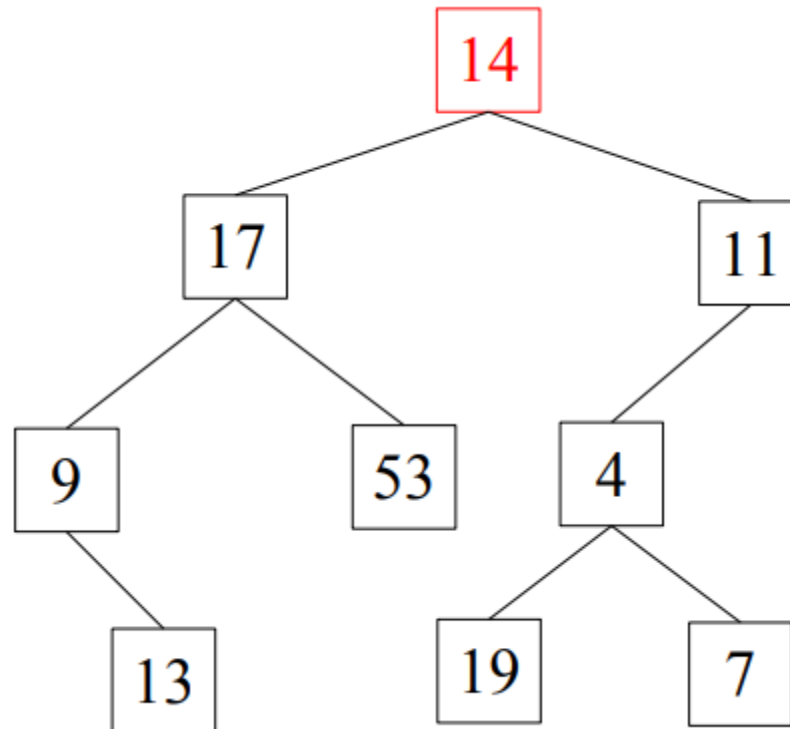
- Exista un singur nod “initial” numit radacina (root);
- Fiecare nod poate fi asociat mai jos in ierarhie cu unul sau mai multe noduri “copil”;
- Fiecare nod are exact un parinte, cu exceptia radacinii, care nu are parinte.

Arbori

- Accesul arborelui se face pornind de la **rădăcina** sa
- Nodurile pot fi:
 - **rădăcină**
 - **frunze**
 - **noduri interne**

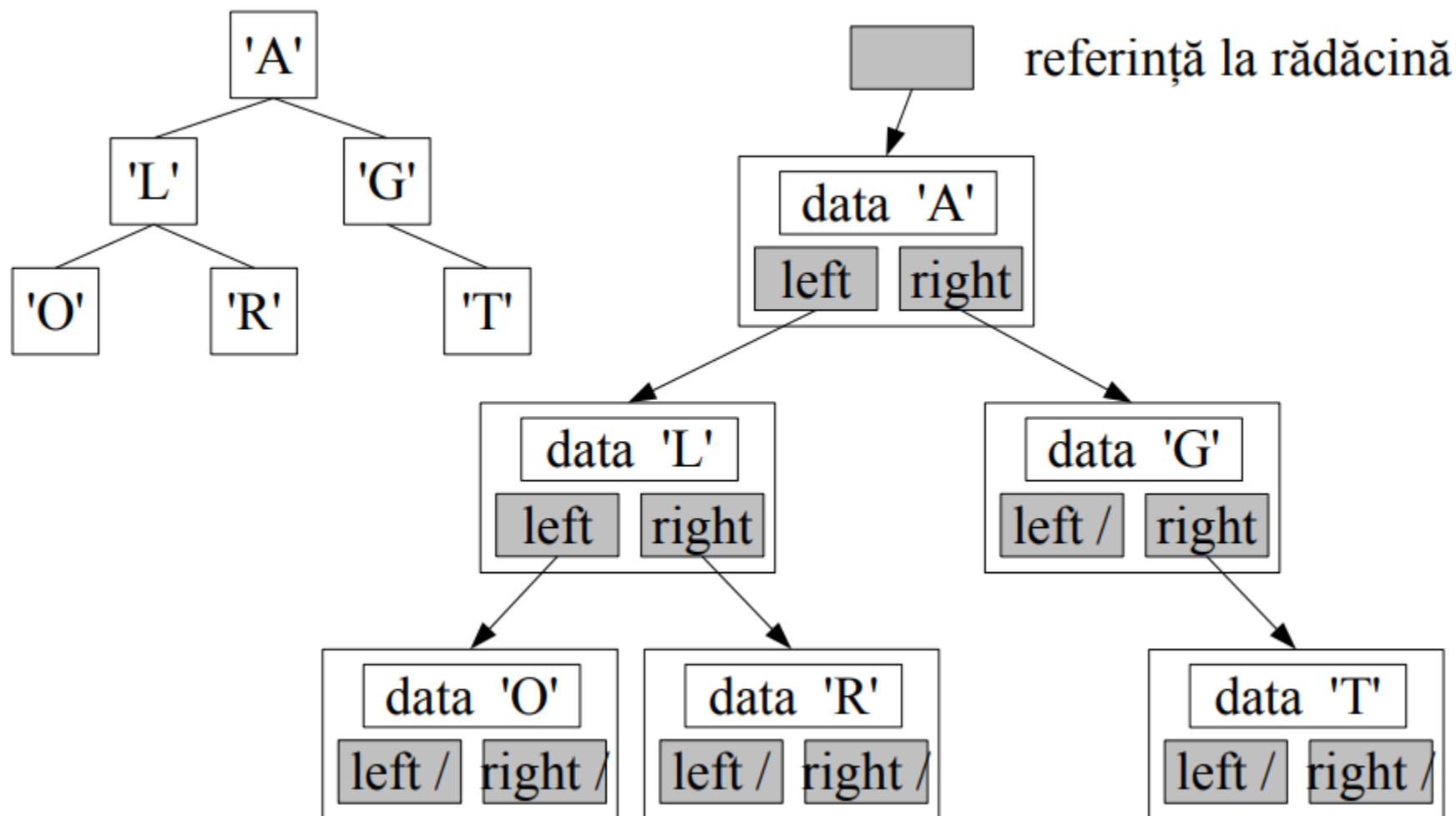


Arbori **binari** (Binary Tree)



- Orice nod are **maxim 2 copii**, cu exceptia frunzelor.
- Din orice nod, deplasarea catre parinte si apoi catre parintele parintelui ne duce in final la radacina (root).
- Singurul nod fara parinte este radacina.

Structura de date a unui nod



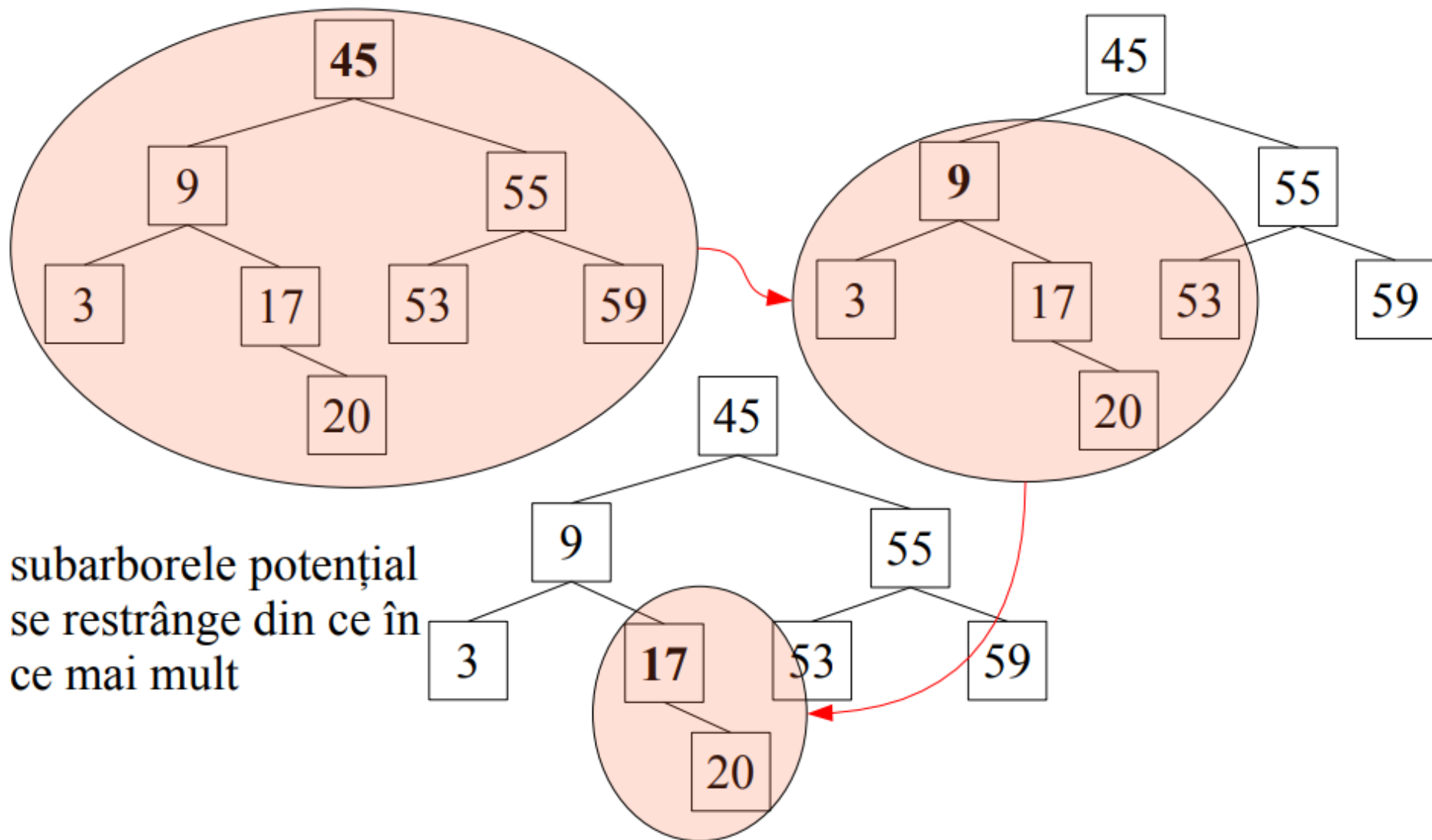
Clasa BTNode

```
public class BTNode {  
    public Object data;  
    private BTNode left, right;  
  
    public BTNode(Object data, BTNode left, BTNode right) {  
        this.data = data;  
        this.left = left;  
        this.right = right;  
    }  
  
    public Object getData() {  
        return data;  
    }  
  
    public BTNode getLeft() {  
        return left;  
    }  
  
    public BTNode getRight() {  
        return right;  
    }  
  
    public void setData(Object data) {  
        this.data = data;  
    }  
  
    public void setLeft(BTNode left) {  
        this.left = left;  
    }  
  
    public void setRight(BTNode right) {  
        this.right = right;  
    }  
}
```

Arbori binari de cautare

- Un nod **x** al unui arbore binar de cautare respecta urmatoarele reguli:
 - orice nod din subarborele stang este mai mic decat nodul **x**
 - orice nod din subarborele drept este mai mare decat nodul **x**
- **Remarci:**
 - Cautarea intr-un arbore binar de cautare se aseamana cu cautarea binara
 - Cu cat arborele este mai mic de inaltime, cu atat cautarea dureaza mai putin

Căutarea elementului 17



Functia de cautare in arbore

```
public static BTreeNode search(BTreeNode node, int k)
{
    if(null==node || (int)node.data==k)
        return node;

    if(k<(int)node.data)
        return search(node.getLeft(), k);
    else
        return search(node.getRight(), k);
}
```


Cod complet – de implementat la laborator

```
public class CautariBinare {  
  
    public static void main(String[] args) {  
        BTreeNode nod1, nod2, nod3, nod4, nod5, nod6, nod7, nod8;  
        nod8 = new BTreeNode(20, null, null);  
        nod7 = new BTreeNode(17, null, nod8);  
        nod6 = new BTreeNode(3, null, null);  
        nod5 = new BTreeNode(9, nod6, nod7);  
        nod4 = new BTreeNode(53, null, null);  
        nod3 = new BTreeNode(59, null, null);  
        nod2 = new BTreeNode(55, nod4, nod3);  
        nod1 = new BTreeNode(45, nod5, nod2);  
  
        BTreeNode nodCautat = search(nod1, 9);  
        if(nodCautat==null)  
            System.out.println("Valoarea cautata nu este in arborele binar de cautare");  
        else  
            System.out.println("S-a gasit valoarea cautata: " + nodCautat.getData());  
    }  
  
    public static BTreeNode search(BTreeNode node, int k)  
    {  
        if(null==node || (int)node.data==k)  
            return node;  
  
        if(k<(int)node.data)  
            return search(node.getLeft(), k);  
        else  
            return search(node.getRight(), k);  
    }  
}
```

Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

Backtracking

Petrecere 😊

Imaginați-vă că astăzi este ziua voastră și aveți invitați. Aranjați o masă frumoasă, apoi vă gândiți cum să vă așezați invitații la masă.

Ați vrea să știți toate posibilitățile de așezare a invitaților la masă, dar realizați în același timp că trebuie să țineți seama și de preferințele lor. Printre invitați există anumite simpatii dar și unele antipatii, de care doriți neapărat să țineți seama, pentru ca petrecerea să fie o bucurie pentru toți.

Să ne gândim cum procedați pentru a identifica toate posibilitățile de a plasa invitații la masă. Începeți prin a scrie niște cartonașe cu numele invitaților.

Petrecere 😊

Alegeți un invitat.

Pentru a-l alege pe al doilea, selectați din mulțimea cartonașelor rămase un alt invitat. Dacă știți că cele două persoane nu se agreează, renunțați la cartonașul lui și alegeți altul și așa mai departe.

Se poate întâmpla ca la un moment dat, când vreți să alegeți cartonașul unui invitat să constatați că nici unul dintre invitații rămași nu se potrivește cu ultima persoană selectată până acum. Cum procedați?

Schimbați ultimul invitat plasat cu un invitat dintre cei rămași și încercați din nou, dacă nici așa nu reușiți, schimbați penultimul invitat cu altcineva și încercați din nou și așa mai departe până când reușiți să plasați toți invitații. Înseamnă că ați obținut o soluție.

Petrecere 😊

Pentru a obține toate celelalte soluții posibile, nu vă rămâne decât să o luați de la început. Aveți cam mult de muncit, iar dacă numărul invitaților este mare...operațiunea devine foarte anevoioasă.

Iată de ce aveți nevoie de un calculator și cunoștințe de algoritmi si logica programarii. 😊

Petrecere 😊

- Relatii:
 - ❤️ desemneaza o relatie de simpatie.
 - O celula fara ❤️ desemneaza o relatie de antipatie.

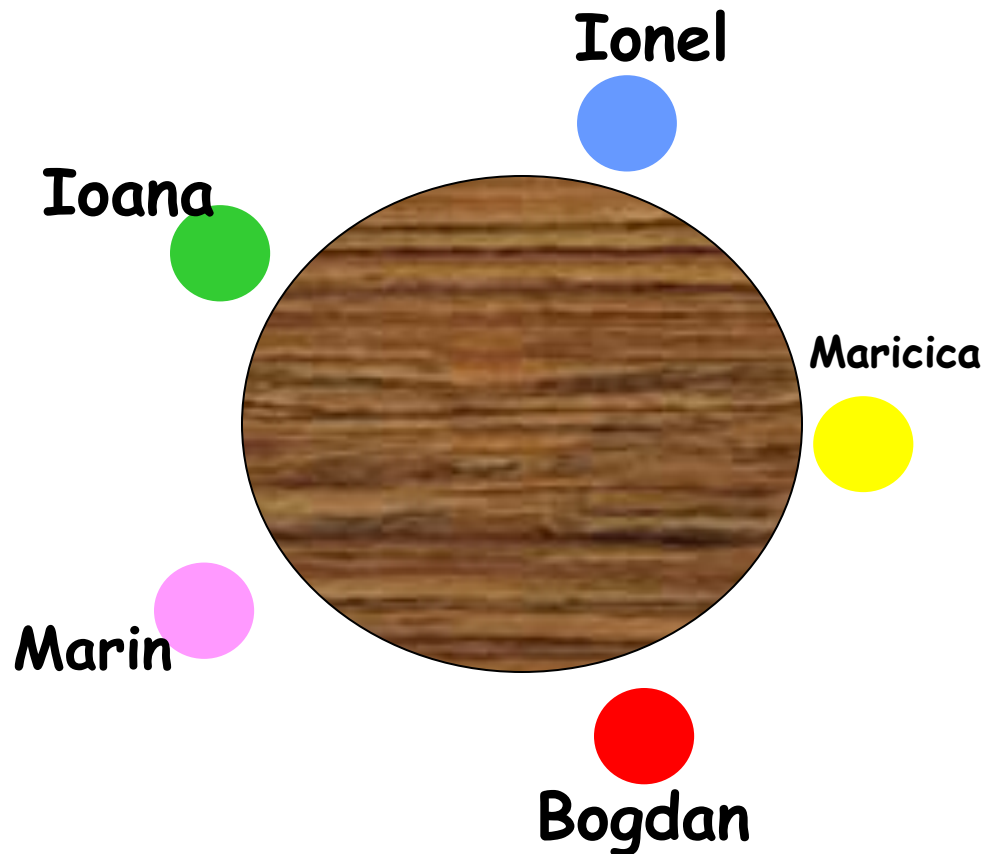
	Ionel	Maricica	Bogdan	Ioana	Marin
Ionel		❤️		❤️	
Maricica	❤️		❤️		❤️
Bogdan		❤️			❤️
Ioana	❤️	❤️	❤️		❤️
Marin			❤️	❤️	

Petrecere 😊

- Conditii de indeplinit:
 - Toti invitatii trebuie asezati in jurul mesei.
 - Invitatii care stau pe locuri adiacente (unul langa altul) trebuie sa se simpatizeze reciproc (♥).

	Ionel	Maricica	Bogdan	Ioana	Marin
Ionel		♥		♥	
Maricica	♥		♥		♥
Bogdan		♥			♥
Ioana	♥	♥	♥		♥
Marin			♥	♥	

Solutie pentru exemplul curent



	Ionel	Maricica	Bogdan	Ioana	Marin
Ionel		♥		♥	
Maricica	♥		♥		♥
Bogdan		♥			♥
Ioana	♥	♥	♥		♥
Marin			♥	♥	

Abordarea “naiva”

Pentru fiecare ordonare/aranjare a invitatilor in jurul mesei, se verifica daca fiecare invitat il simpatizeaza pe urmatorul care se aseaza langa el.



Cat timp este necesar pentru verificarea tuturor posibilitatilor? (cazul cel mai nefavorabil)

Invitati	Nr. de permutari
n	$(n-1)!$
5	24
15	87178291200
100	$\approx 9 \cdot 10^{155}$



Daca avem un computer capabil de 10^{10} instructiuni pe secunda, va fi nevoie de $\approx 3 \cdot 10^{138}$ ani!

Tehnica backtracking (cautare cu revenire)

Ideea de bază:

- Soluțiile sunt construite în manieră **incrementală** prin găsirea succesivă a valorilor potrivite pentru componente (la fiecare etapă se completează o componentă; o soluție în care doar o parte dintre componente sunt completate este denumită **soluție parțială**)
- Fiecare soluție parțială este evaluată cu scopul de a stabili dacă este **validă (promițătoare, viabilă)**. O soluție parțială validă poate conduce la o soluție finală pe când una invalidă încalcă restricțiile parțiale, însemnând că nu va conduce **niciodată** la o soluție care să satisfacă toate restricțiile problemei
- Dacă nici una dintre valorile corespunzătoare unei componente nu conduce la o soluție parțială validă atunci **se revine la componenta anterioară** și se încearcă altă valoare pentru aceasta.

Tehnica backtracking (cautare cu revenire)

Aceasta tehnica se foloseste in rezolvarea problemelor care indeplinesc simultan urmatoarele conditii:

- solutia lor poate fi pusa sub forma unui vector $S=(x_1, x_2, \dots, x_n)$;
- fiecare element al solutiei x_i poate sa ia valori intr-o multime A_i ;
- multimele $A_1, A_2 \dots A_n$ sunt multimi finite, iar elementele lor se considera ca se afla intr-o relatie de ordine bine stabilita;
- A_1, A_2, \dots, A_n pot coincide;
- nu se dispune de o alta metoda de rezolvare, mai rapida.

Tehnica backtracking (pas cu pas)

Se alege primul element x , ce aparține lui A .

Presupunând generate elementele $x_1, x_2 \dots, x_k$, aparținând mulțimilor $A_1, A_2 \dots, A_k$, se alege (dacă există) x_{k+1} , primul element disponibil din mulțimea A_{k+1} . Apar două posibilități:

1. Nu s-a găsit un astfel de element, caz în care se reia căutarea considerând generate complet elementele $x_1, x_2 \dots, x_{k-1}$, iar aceasta se reia de la următorul element al mulțimii A_k rămas netestat;

2. A fost găsit, caz în care se testează dacă acesta îndeplinește anumite condiții de continuare apărând astfel două posibilități:

a) îndeplinește, caz în care se testează dacă s-a ajuns la soluție și apar din nou două posibilități

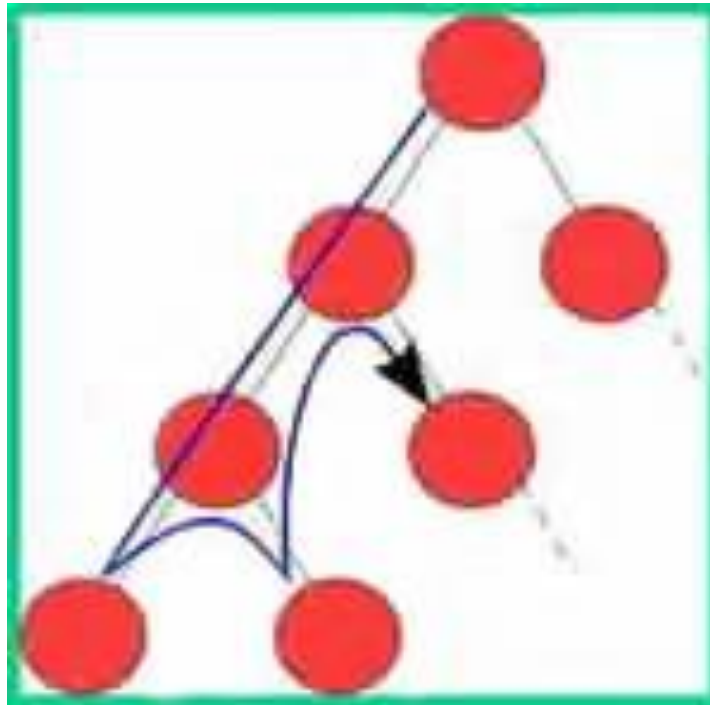
a1) s-a ajuns la soluție, se tipărește soluția și se reia algoritmul considerând generate elementele $x_1, x_2 \dots, x_k$. (se caută în continuare, un alt element al mulțimii A_{k+1} , rămas netestat);

a2) nu s-a ajuns la soluție, caz în care se reia algoritmul considerând generate elementele $x_1, x_2 \dots, x_{k+1}$, și se caută un prim element $x_{k+2} \in A_{k+2}$.

b) nu le îndeplinește caz în care se reia algoritmul considerând generate elementele $x_1, x_2 \dots, x_k$, iar elementul x_{k+1} se caută între elementele mulțimii A_{k+1} , rămase netestate.

Algoritmul se termină atunci când nu există nici un element $x_1 \in A_1$ netestat.

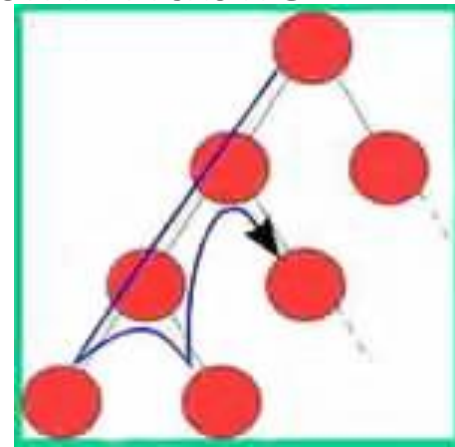
Backtracking



Backtracking

Backtracking = parcurgerea limitată (conform condițiilor de continuare) în adâncime a unui arbore.

- Spațiul soluțiilor este organizat ca un arbore
 - un vârf este viabil dacă sunt șanse să se găsească o soluție explorând subarborele cu rădăcina în acel vârf
 - sunt explorați numai subarborii cu rădăcini viabile



Backtracking iterativ (nerecursiv)

```
k ← 1;
cât timp k > 0 repetă
    dacă ( k = n+1 ) atunci {am găsit o soluție}
        prelucrează(x1, ..., xn); {afișează soluție}
        k ← k-1; {revenire după găsirea soluției}
    altfel
        dacă (∃ v ∈ Ak netestat) atunci
            xk ← v; {atribuie}
            dacă (x1, ..., xk îndeplinește cond. continuare)
                atunci
                    k ← k+1; {avansează}
            sfârșit dacă
        altfel
            k ← k-1; {revenire}
        sfârșit dacă
sfârșit cât timp
```

Backtracking recursiv
pentru cazul particular $A_i = \{1, 2, \dots, n\}$, $i = 1, \dots, n$.
Apelul initial este backrec(1).

```
procedura backrec(k)
  dacă (k=n+1) atunci {am găsit o soluție}
    prelucrează( $x_1, \dots, x_n$ ); {afișează soluție}
  altfel
     $i \leftarrow 1$ ;
    cât timp  $i \leq n$  repetă {toate valorile posibile }
       $x_k \leftarrow i$ ;
      dacă ( $x_1, \dots, x_k$  îndeplinește cond. continuare)
        atunci backrec(k+1);
      sfârșit dacă
         $i \leftarrow i+1$ ;
    sfârșit cât timp
  sfârșit dacă
  sfârșit procedura
```

Backtracking recursiv - simplificat

```
void BK(int k)                                //k-pozitia din vector care se completeaza
{
    int i;
    for(i=1;i<=nr_elemente_Sk;i++)           //parcure elementele multimii Sk
    {
        v[k]=i;                               //selecteaza un element din multime
        if(validare(k)==true)                 //valideaza conditiile de continuare a problemei
        {
            if(solutie(k)==true)              //verifica daca s-a obtinut o solutie finala
                afisare(k);                   //afiseaza solutia
            else
                BK(k+1);                       //reapeleaza functia pentru pozitia k+1
        }
    }
    //daca nu mai exista niciun element neselectat in multimea Sk,
    //se inchide nivelul de stiva si se revine pe pozitia k-1 a vectorului
    //executia functiei se incheie. Dupa ce s-au inchis toate nivelurile stivei,
    // inseamna ca in vectorul v nu mai poate fi selectat niciun element din multimile Sk
}
```

Model implementare Java

```
invitati=new int[5][5];  
invitati[0][0]=0;invitati[0][1]=1;invitati[0][2]=0;invitati[0][3]=1;invitati[0][4]=0;  
invitati[1][0]=1;invitati[1][1]=0;invitati[1][2]=1;invitati[1][3]=0;invitati[1][4]=1;  
invitati[2][0]=0;invitati[2][1]=1;invitati[2][2]=0;invitati[2][3]=0;invitati[2][4]=1;  
invitati[3][0]=1;invitati[3][1]=1;invitati[3][2]=1;invitati[3][3]=0;invitati[3][4]=1;  
invitati[4][0]=0;invitati[4][1]=0;invitati[4][2]=1;invitati[4][3]=1;invitati[4][4]=0;
```

	0	1	2	3	4
	Ionel	Maricica	Bogdan	Ioana	Marin
0	Ionel	1	0	1	0
1	Maricica	0	1	0	1
2	Bogdan	1	0	0	1
3	Ioana	1	1	0	1
4	Marin	0	1	1	0

Implementare functie de afisare a rezultatului obtinut

```
public static int[][] invitati;  
public static int v[];  
  
public static void afisare(int k)  
{  
    //afisez vectorul de rezultate  
    System.out.println("");  
    for(int i=0;i<=k;i++)  
    {  
        System.out.print(" " + v[i] + " ");  
    }  
}
```

Implementare functie de verificare a solutiei

```
public static boolean solutie (int k) //verificam daca am reusit sa gasim o solutie
{
    if(k==invitati[0].length-1) //am obtinut o permutare, adica
        return true;//am reusit sa depunem in vector "n" invitati
    else
        return false;
}
```

Implementare **functie de verificare a validitatii elementului curent**

```
public static boolean valid(int k) //verificam conditiile de continuare
{
    //prima conditie: toti invitatii asezati pana acum la masa sa fie diferiti de cel curent
    for(int i=0;i<k;i++)
    {
        if(v[i]==v[k])
            return false;
    }

    //a doua conditie: invitatul curent sa se simpatizeze cu precedentul
    for(int i=0;i<k-1;i++)
    {
        if(invitati[v[i]][v[i+1]]==0) //daca nu se simpatizeaza, nu e valida varianta de pana acum
            return false;
    }

    //a treia conditie: daca este ultimul asezat la masa, trebuie sa simpatizeze cu primul
    if(k==invitati[0].length-1 && invitati[v[0]][v[k]]==0)
        return false;

    //daca toate conditiile au fost indeplinite, inseamna ca varianta de pana acum este valida
    //si are sanse de continuare
    return true;
}
```

Implementare functie recursiva backtracking

```
static void BK(int k)
{
    int i;
    //i = elementul selectat din multimea Sk trebuie sa fie variabila locala
    //pentru a se memora pe stiva de apeluri
    for(i=0;i<invitati[0].length;i++) //parcurgem elementele multimii Sk
    {
        v[k]=i; //selectam un element din multimea Sk
        if(valid(k))
        {
            if(solutie(k))
                afisare(k);
            else
                BK(k+1);
        }
    }
}
```


Implementare apel main

```
public static void main(String[] args) {  
    invitati=new int[5][5];  
    invitati[0][0]=0;invitati[0][1]=1;invitati[0][2]=0;invitati[0][3]=1;invitati[0][4]=0;  
    invitati[1][0]=1;invitati[1][1]=0;invitati[1][2]=1;invitati[1][3]=0;invitati[1][4]=1;  
    invitati[2][0]=0;invitati[2][1]=1;invitati[2][2]=0;invitati[2][3]=0;invitati[2][4]=1;  
    invitati[3][0]=1;invitati[3][1]=1;invitati[3][2]=1;invitati[3][3]=0;invitati[3][4]=1;  
    invitati[4][0]=0;invitati[4][1]=0;invitati[4][2]=1;invitati[4][3]=1;invitati[4][4]=0;  
  
    //v=vectorul de solutii pentru invitati  
    //lungimea vectorului este egala cu numarul invitatilor  
    v= new int[invitati[0].length];  
  
    //apelez recursiv backtracking  
    BK(0);  
    System.out.println();  
    System.out.println("Am terminat de aranjat invitatii in toate felurile.");  
}
```













Rezultate obtinute

0 1 2 4 3
0 1 4 2 3
0 3 2 4 1
0 3 4 2 1
1 0 3 2 4
1 0 3 4 2
1 2 4 3 0
1 4 3 0 2
1 4 3 2 0
2 1 0 3 4
2 4 3 0 1
3 0 1 2 4
3 0 1 4 2
3 1 2 4 0
3 1 4 2 0
3 2 1 0 4
3 2 1 4 0
3 4 2 1 0
4 2 1 0 3
4 3 0 1 2
4 3 1 0 2

Am terminat de aranjat invitatii in toate felurile.

Conditii indeplinite:

- Toti invitatii sunt diferiti;
- Oricare 2 invitati alaturati se simpatizeaza;
- Ultimul se simpatizeaza cu primul.

	0	1	2	3	4
	Ionel	Maricica	Bogdan	Ioana	Marin
0	Ionel				
1	Maricica				
2	Bogdan				
3	Ioana				
4	Marin				

Problema celor N dame (regine)

- Fiind dată o tablă de șah de dimensiune $N \times N$, se cere să se aranjeze cele n dame în toate modurile posibile pe tabla de șah, astfel încât să nu se afle pe aceeași linie, coloană, sau diagonală (**damele să nu se atace**).
- **Nota:** orice asemanare cu damele din viata reala este ~~voit~~ pur intamplatoare. 😊

Exemplu solutie

- Pentru $N=4$, o solutie este urmatoarea:

	D		
			D
D			
		D	

Observăm că **o damă va fi plasată întotdeauna singură pe o linie**. Acest lucru ne permite să memorăm fiecare soluție într-un vector v , considerând că o căsuță k a vectorului reprezintă linia k iar conținutul ei, adică $v[k]$ va conține numărul coloanei în care vom plasa regina.

Pentru exemplul de mai sus, vectorul v va avea următorul conținut:

1	3	0	2
0	1	2	3

Nr.coloana

Index vector (nr.linie)

Condițiile de continuare ale problemei

- a) Damele să nu fie pe aceeași linie - această condiție este îndeplinită prin modul în care am organizat memorarea datelor și anume într-o căsuță a vectorului putem trece un singur număr de coloană.
- b) Damele să nu fie pe aceeași coloană – adică $v[k] \neq v[i]$, cu $0 \leq i \leq k$.
- c) Damele să nu fie pe aceeași diagonală. Dacă două dame se găsesc pe aceeași diagonală înseamnă că cele două distanțe dintre cele două dame măsurate pe linie respectiv pe coloană, sunt egale. Prin urmare condiția ca damele să nu fie pe aceeași diagonală este: $|v[k] - v[i]| \neq k - i$, cu $0 \leq i \leq k$.

	$v[i]$	$v[k]$		
linia i		D		
linia k				D

Conditia de gasire a unei solutii valide

Obținem o soluție dacă am reușit să plasăm toate cele n dame, adică $n=k$ (daca consideram ca numerotarea liniilor si coloanelor incepe de la 1) sau $n-1=k$ (daca consideram ca numerotarea incepe de la 0).

	D		
			D
D			
		D	

N=4. Pentru usurinta de asociere mentala a liniilor si coloanelor, consideram prima linie ca avand indexul 1. La fel si prima coloana.

Rulare algoritm backtracking pana la gasirea primei solutii.

Rularea continua in mod asemanator pentru toate solutiile.

v k=1

1			
---	--	--	--

v k=2

1	1		
---	---	--	--

v k=2

1	2		
---	---	--	--

k=2

1	3		
---	---	--	--

pe linia 3 nu mai putem plasa nici o damă, selectăm altă valoare pe linia 2

v k=2

1	4		
---	---	--	--

v k=3

1	4	1	
---	---	---	--

v k=3

1	4	2	
---	---	---	--

-pe linia 4 nu putem plasa nici o damă
-pe linia 3 nici o altă coloană nu este corectă
-pe linia 2 nu mai există nici o poziție disponibilă
-se revine la linia 1

v k=1

2			
---	--	--	--

v k=2

2	1		
---	---	--	--

v k=2

2	2		
---	---	--	--

v k=2

2	3		
---	---	--	--

v k=2

2	4		
---	---	--	--

v k=3

2	4	1	
---	---	---	--

v k=4

2	4	1	1
---	---	---	---

v k=4

2	4	1	2
---	---	---	---

v k=4

2	4	1	3
---	---	---	---

Algoritmul continuă pentru a genera toate soluțiile.

D			

D			
		D	

D			
			D

D			
			D
	D		

	D		

	D		
			D

	D		
D			

	D		
			D
D			

Am obținut prima soluție !

```
public class NDame {
```

```
    static int[] v; //v=vectorul de solutii
    static int N; //N=numarul de dame
    public static void main(String[] args) {
        N=4; v=new int[N];
        BK(0);
    }
```

```
    static void BK(int k)
    {
        int i;
        for(i=0;i<N;i++)
        {
            v[k]=i;
            if(valid(k))
            {
                if(solutie(k))
                    afisare();
                else
                    BK(k+1);
            }
        }
    }
```

```
    static boolean solutie(int k) //daca k=N-1, inseamna ca am plasat toate damele
    {
        if(k==N-1)
            return true;
        else
            return false;
    }
```

```
    static boolean valid(int k) //verificam daca solutia partiala curenta este valida
    {
        for(int i=0;i<k;i++)
            if((v[i]==v[k]) || (Math.abs(v[k]-v[i])==k-i))
                return false;
        return true;
    }
```

```
    static void afisare() //afisam solutiile sub forma unei matrice
    {
        System.out.println();
        for(int i=0;i<N;i++)
        {
            for(int j=0;j<N;j++)
                if(v[i]==j)
                    System.out.print("D ");
            else
                System.out.print("- ");
        }
        System.out.println();
    }
```


Rezultate N=4

```

- D - -
- - - D
D - - -
- - D -

- - D -
D - - -
- - - D
- D -

```

```

D - - -
- - D -
- - - D
- D - -
- - D -

```

```

D - - -
- - D -
- D - -
- - - D
- - D -

```

```

- D - -
- - D -
D - - -
- - D -
- - - D

```

```

- D - -
- - - D
- - D -
D - - -
- - D -

```

```

- - D -
D - - -
- - D -
- D - -
- - - D

```

```

- - D -
- - - D
- D - -
- - D -
D - - -

```

```

- - - D
D - - -
- - D -
- - - D
- D - -

```

```

- - - D
- D - -
- - - D
- - D -
D - - -

```

```

- - - D
- D - -
- - D -
D - - -
- - D -

```

```

- - - D
- - D -
D - - -
- - D -
- D - -

```

Rezultate N=5

Plata unei sume cu monede de valori date

- Fiind data o suma **S** si **n** monede de valori date, sa se determine toate posibilitatile de plata a sumei S cu aceste monede.
- Utilitate economica:
 - un automat de schimbat bancnote in monede;
 - un automat de cafea care da rest;
 - un bancomat.

```
public class Monede {  
    static int v[], w[], z[]; //v=vectorul solutie, w=valoarea monedelor, z=nr. maxim de monede de fiecare tip  
    static int suma;
```

```
    public static void main(String[] args) {  
        w=new int[3];w[0]=1;w[1]=2;w[2]=5;  
        z=new int[3];z[0]=5;z[1]=5;z[2]=5;  
        suma=10;  
        v=new int[w.length];  
        BK(0);  
    }
```

```
    public static void BK(int k)  
    {  
        int i;  
        for(i=0;i<=z[k];i++)  
        {  
            v[k]=i;  
            if(valid(k)==true)  
            {  
                if(solutie(k)==true)        afisare(k);  
                else  
                { if (k<w.length-1)  
                    BK(k+1); }  
            }  
        }  
    }
```

```
    public static void afisare(int k) //afisez vectorul de solutii  
    {  
        System.out.println("");  
        for(int i=0;i<=k;i++)  
        { System.out.print(" " + v[i] + " monede cu valoarea de " + w[i] + ", "); }  
        System.out.print(" fac suma de " + suma);  
    }
```

```
    public static boolean valid(int k)  
    {  
        int sumaActuala=0;  
        for(int i=0;i<k;i++) //calculez suma obtinuta pana la acest moment  
            sumaActuala=sumaActuala+v[i]*w[i];  
        if(sumaActuala<=suma && k<w.length) //verifica daca inca mai pot adauga monede la suma actuala  
            return true;  
        else  
            return false;  
    }
```

```
    public static boolean solutie(int k)  
    {  
        int sumaActuala=0;  
        for(int i=0;i<=k;i++) //calculez suma obtinuta pana in acest moment  
            sumaActuala = sumaActuala + v[i]*w[i];  
        if(sumaActuala==suma) //verific daca suma actuala este egala cu suma dorita  
            return true;  
        else  
            return false;  
    }
```

```
}
```

Rezultate obtinute

0 monede cu valoarea de 1, 0 monede cu valoarea de 2, 2 monede cu valoarea de 5, fac suma de 10

0 monede cu valoarea de 1, 5 monede cu valoarea de 2, fac suma de 10

1 monede cu valoarea de 1, 2 monede cu valoarea de 2, 1 monede cu valoarea de 5, fac suma de 10

2 monede cu valoarea de 1, 4 monede cu valoarea de 2, fac suma de 10

3 monede cu valoarea de 1, 1 monede cu valoarea de 2, 1 monede cu valoarea de 5, fac suma de 10

4 monede cu valoarea de 1, 3 monede cu valoarea de 2, fac suma de 10

5 monede cu valoarea de 1, 0 monede cu valoarea de 2, 1 monede cu valoarea de 5, fac suma de 10

Problema permutarilor

Problema generării permutărilor este probabil cea mai reprezentativă pentru metoda backtracking deoarece ea conține toate elementele specifice metodei.

Probleme similare, care solicită determinarea tuturor soluțiilor posibile, necesită doar adaptarea acestui algoritm modificând fie modalitatea de selecție a elementelor din mulțimea S_k , fie condițiile de continuare fie momentul obținerii unei soluții.

```
public class Permutari {
    static int[] v; //vectorul de solutii
    static int n;

    public static void main(String[] args) {
        n=4; v=new int[n];
        BK(0);
    }

    static void BK(int k)
    {
        int i;
        for(i=0;i<n;i++)
        {
            v[k]=i;
            if(valid(k))
            {
                if(solutie(k))
                    afisare();
                else
                    BK(k+1);
            }
        }
    }

    static void afisare()
    {
        System.out.println();
        for(int i=0;i<v.length;i++)
            System.out.print(" " + v[i]);
    }

    static boolean solutie(int k)
    {
        if(k==n-1)
            return true;
        else
            return false;
    }

    static boolean valid(int k)
    {
        for(int i=0;i<k;i++)
            if(v[i]==v[k])
                return false;
        return true;
    }
}
```

Problema comis-voiajorului

- Exista mai multe localitati (ex.: A,B,C,...).
- Intre diverse localitati existe rute (ex: (A-B, A-C, C-D, D-A,...)).
- Un comis-voiajor trebuie sa treaca prin fiecare localitate fix o singura data si ultima localitate sa fie cea din care a pornit.
- Determinati traseele posibile.

Problema cuvintelor din dictionar (joc fazan)

- Aveti mai multe biletele.
- Fiecare biletel contine un cuvant.
- Reguli:
 - Se trage un biletel cu un cuvant;
 - Trebuie trase pe rand toate biletelele;
 - Fiecare nou biletel trebuie sa aiba un cuvant care incepe cu ultima litera a precedentului biletel.

Bibliografie recomandata

- Jeff Erickson, Algorithms, 2015. Lecture 1.3 (Backtracking) – capitolul incepe la pagina 67.

Algoritmi si Logica Programarii

Informatica Economica

Anul 2

FEAA

© Octavian Dospinescu & Catalin Strimbei

Metoda greedy

Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Joc FloodIt (umplere colorata)

- Avem o grila NxN cu celule colorate in mod aleator.
- Celula din coltul stanga sus este celula de start.
- Toate celulele care au aceeaasi culoare cu celula de start si care pot fi accesate pe **orizontala** sau pe **verticala** (nu pe diagonala!) sunt considerate **conectate** cu celula de start.
- Se pune problema schimbarii succesive a culorii celulei de start si a tuturor celor conectate cu ea astfel incat grila sa fie umpluta complet cu aceeaasi culoare **in cat mai putine etape**.

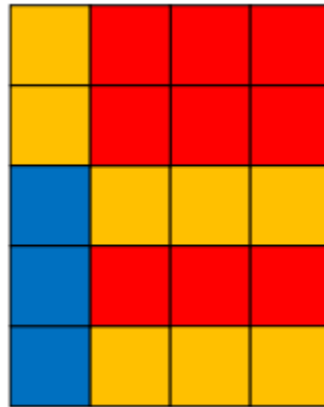
(R.Clifford, The Complexity of Flood Filling Games, 2011)



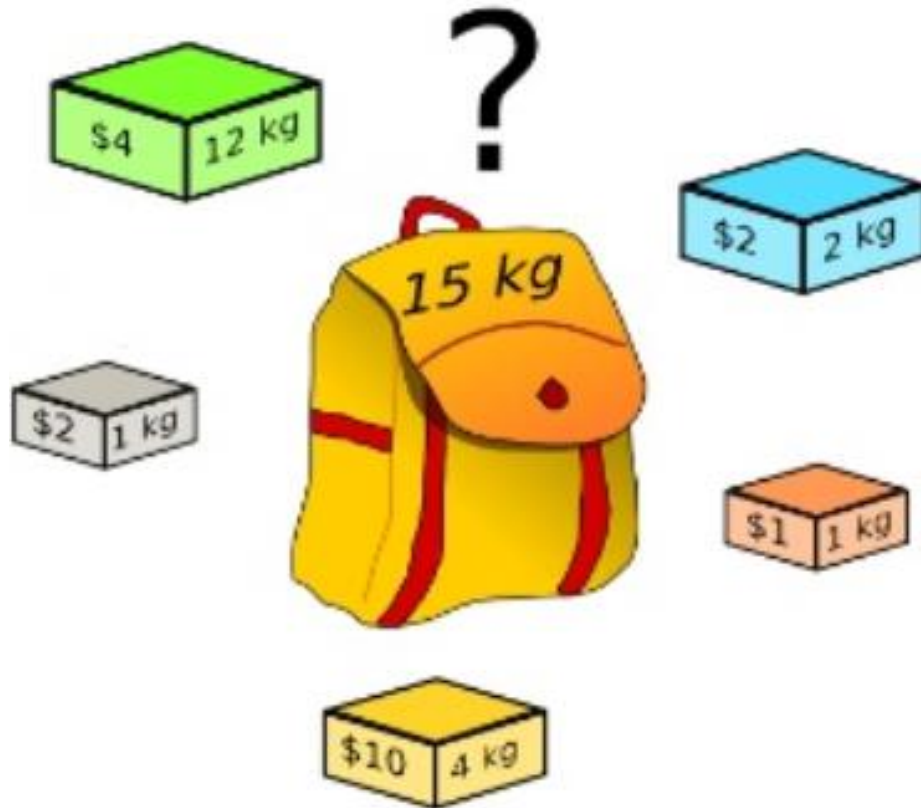
Deci care ar fi o strategie intuitiva de joc? 😊

FloodIT

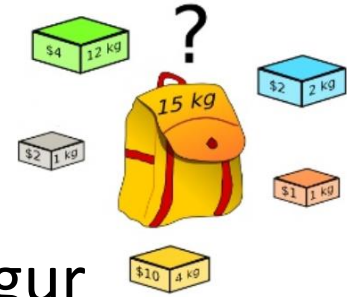
Cu ce culoare ati incepe? 😊



Hotul cu rucsacul



Hotul cu rucsacul



- Un hot nepravazator ☺ are la dispozitie un singur rucsac cu care poate transporta o greutate maxima **G_{max}**.
- Hotul are de ales din $n \leq 50$ obiecte si, evident, intentioneaza sa obtina un **castig maxim** in urma singurului transport pe care il poate face.
- Cunoscand, pentru fiecare obiect i greutatea **G_i** si castigul **C_i** pe care hotul l-ar obtine transportand obiectul respectiv in intregime, scrieti un program care sa determine o incarcare optima a rucsacului, in ipoteza ca hotul poate incarca in rucsac orice parte dintr-un obiect.

Investitorul cel istet

- Un investitor poate cumpara actiuni la diverse companii: A_1, A_2, \dots, A_n .
- Fiecare actiune are un anumit cost: C_1, C_2, \dots, C_n .
- Fiecare actiune aduce un anumit dividend: D_1, D_2, \dots, D_n .
- La fiecare companie, investitorul poate cumpara un anumit numar maxim de actiuni: N_1, N_2, \dots, N_n .
- Investitorul are la dispozitie o suma S de bani pe care o investeste.
- Ajutati investitorul sa-si maximizeze investitia, adica raportul $\text{dividendeObtinue} / \text{costTotalActiuni}$.

Restaurant - meniuri

- La inaugurarea unui restaurant sunt prezente mai multe persoane.
- Clienții își aleg din meniul pus la dispoziție câte o specialitate. Fiecare specialitate necesita cate un anumit timp pentru preparare.
- Dar deocamdată restaurantul a angajat un singur bucătar 😊 care pregătește mâncărurile una după alta, deci clienții nu pot fi serviți decât pe rând.
- Presupunând că bucătarul se apucă de lucru după ce s-au strâns toate comenzile, stabiliți în ce ordine trebuie să pregătească specialitățile, astfel încât **timpul mediu de așteptare** al clienților să fie **minim**.

Restaurant – propunere abordare greedy

- Pentru a minimiza timpul mediu de așteptare ($t_{impmediu}$) va trebui să **minimizăm** timpul total de așteptare (**$t_{imptotal}$**) al persoanelor pentru a fi servite, deoarece **$t_{impmediu} = t_{imptotal}/n$** .
- O persoană va trebui să aștepte prepararea meniului ei și a tuturor persoanelor care vor fi servite în fața lui. Intuitiv, dacă o persoană care dorește un meniu sofisticat este servită înaintea uneia al cărei meniu se prepară mai repede, timpul total de așteptare al celor doi este mai mare decât dacă servirea s-ar fi făcut invers. De exemplu, pentru timpii 30 și 20, în prima situație $t_{imptotal} = 30 + (30 + 20)$, iar în a doua situație $t_{imptotal} = 20 + (20 + 30)$.
- În concluzie, **vom alege persoanele în ordinea crescătoare a timpilor de preparare a meniurilor**, ceea ce necesită ca ele să fie ordonate crescător în funcție de acest criteriu și vom calcula timpul total de așteptare ca fiind suma timpilor de așteptare ai fiecărei persoane.
- În situația în care există mai multe durate egale de preparare, nu contează ordinea alegerii acestora.

Submultimi

- Se considera o multime avand **N** elemente numere intregi.
- Sa se determine o submultime de suma maxima a acesteia.
- Exemplu:
 - Multimea: 1, 7, 9, 0, -5, 2
 - Submultimea rezultat: 1, 7, 9, 2.
- Abordare: vom alege submultimea numerelor strict pozitive.

Spectacole



- Managerul artistic al unui festival trebuie sa selecteze un **numar cat mai mare de spectacole** ce pot fi jucate in **singura sala** pe care o are la dispozitie.
- Stiind ca i s-au propus **n** spectacole si pentru fiecare spectacol i-a fost anuntat intervalul in care se poate desfasura $[S_i, F_i]$ (S_i reprezinta ora si minutul de inceput, iar F_i ora si minutul de final al spectacolului i), scrieti un program care sa permita spectatorilor vizionarea **unui numar cat mai mare de spectacole**.

Spectacole



Exemplu pentru 5 spectacole disponibile. Ora start si ora final.

1. 12:30 16:30
2. 15:00 18:00
3. 10:00 18:30
4. 18:00 20:45
5. 12:15 13:00

Rezultat posibil:

5. 12:15 13:00
2. 15:00 18:00
4. 18:00 20:45

Descriere solutie: Vom sorta crescator spectacolele dupa ora de final. Vom selecta initial primul spectacol (cel care se termina cel mai devreme). In continuare vom selecta, la fiecare pas, primul spectacol neselectat, care nu se suprapune peste cele deja selectate.

Spectacole

Exemplu pentru 5 spectacole disponibile. Ora start si ora final.

1. 12:30 16:30
2. 15:00 18:00
3. 10:00 18:30
4. 18:00 20:45
5. 12:15 13:00

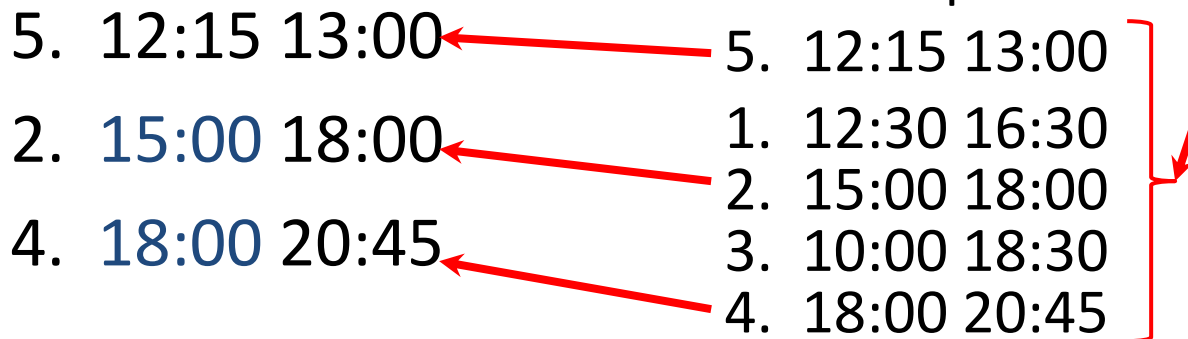
Descriere solutie: Vom sorta crescator spectacolele dupa **ora de final**. Vom selecta initial primul spectacol (cel care se termina cel mai devreme). In continuare vom selecta, la fiecare pas, primul spectacol neselectat, care nu se suprapune peste cele deja selectate.

Rezultat posibil:

5. 12:15 13:00
2. 15:00 18:00
4. 18:00 20:45

Sortare spectacole dupa ora de final

5. 12:15 13:00
1. 12:30 16:30
2. 15:00 18:00
3. 10:00 18:30
4. 18:00 20:45



Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Greedy – cateva remarci generale

- *Metoda Greedy* este una din cele mai directe tehnici de proiectare a algoritmilor care se aplica la o varietate larga de probleme.
- In general,aceasta metoda se aplica problemelor de optimizare.
- Specificul acestei metode consta in faptul ca se construiesc solutiile optime pas cu pas,la fiecare pas fiind selectat (sau „inghitit cu lacomie”) in solutie elementul care pare „cel mai bun” la momentul respectiv, in speranta ca va duce la solutie optima globala.

Componente algoritmi greedy

În general, algoritmi *greedy* au cinci componente:

- O **mulțime de candidați**, din care se creează o soluție.
- **Funcția de selecție**, care alege cel mai bun candidat pentru a fi adăugat la soluție.
- O **funcție de fezabilitate**, care este folosită pentru a determina dacă un candidat poate fi utilizat pentru a contribui la o soluție.
- O **funcție obiectiv**, care atribuie o valoare unei soluții sau unei soluții parțiale, și
- O **funcție de soluție**, care va indica atunci când s-a descoperit o soluție completă.

Metodologie Greedy

- Se da o multime A cu n elemente si se cere sa se determine o submultime a sa (B) care satisface anumite restrictii. Aceasta submultime se numeste **solutie posibila**. Se cere sa se determine o solutie posibila care fie sa **maximizeze** fie sa **minimizeze** o anumita functie obiectiv data. Aceasta solutie posibila se numeste **solutie optima**.
- **Metoda Greedy lucreaza in pasi astfel:**
 1. Multimea B este vida la inceput.
 2. Se alege un element din A care **pare a fi solutia optima** la pasul i .
 3. Se verifica daca elementul ales poate fi adaugat la multimea solutiilor; daca da, atunci va fi adaugat.
 4. Procedeu continua astfel, repetitiv, pana cand au fost determinate toate elementele din multimea solutiilor.

Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Forma generala a metodei greedy

```
Subalgoritm Greedy(A,B) :  
  B  $\leftarrow \emptyset$   
  cât timp nu Soluție(B) și A  $\neq \emptyset$  execută:  
    Alege(A,b)  
    Elimină(A,b)  
    dacă Posibil(B,b) atunci  
      Adaugă(B,b)  
    sfârșit dacă  
  sfârșit cât timp  
  dacă Soluție(B) atunci  
    scrie B  
  altfel  
    scrie 'Nu s-a gasit solutie.'  
  sfârșit dacă  
sfârșit subalgoritm
```

```

Subalgoritm Greedy(A,B) :
  B ← ∅
  cât timp nu Soluție(B) și A ≠ ∅ execută:
    Alege(A,b)
    Elimină(A,b)
    dacă Posibil(B,b) atunci
      Adaugă(B,b)
    sfârșit dacă
  sfârșit cât timp
  dacă Soluție(B) atunci
    scrie B
  altfel
    scrie 'Nu s-a gasit solutie.'
  sfârșit dacă
sfârșit subalgoritm

```

In acest algoritm am utilizat următorii subalgoritmi:

- **Soluție(B)** de tip funcție verifică dacă B este soluție optimă a problemei.
- **Alege(A,b)** extrage cel mai promițător element b din mulțimea A pe care îl poate alege la un moment dat;
- **Posibil(B,b)** de tip funcție verifică dacă este posibil să se obțină o soluție, nu neapărat optimă, prin adăugarea lui b la mulțimea B.

```

Subalgoritm Greedy(A,B) :
  B  $\leftarrow \emptyset$ 
  cât timp nu Soluție(B) și A  $\neq \emptyset$  execută:
    Alege(A,b)
    Elimină(A,b)
    dacă Posibil(B,b) atunci
      Adaugă(B,b)
    sfârșit dacă
  sfârșit cât timp
  dacă Soluție(B) atunci
    scrie B
  altfel
    scrie 'Nu s-a gasit solutie.'
  sfârșit dacă
sfârșit subalgoritm

```

1. La fiecare pas se alege cel mai promițător element la momentul respectiv. Dacă un element se introduce în mulțimea B, el nu va fi niciodată eliminat de acolo.
2. Dacă se alege un element din A care nu se poate adăuga mulțimii B, el se elimină din A și nu se mai testează ulterior.
3. În rezolvarea problemelor, de multe ori este utilă ordonarea mulțimii A înainte ca algoritmul propriu-zis să fie aplicat în funcție de cerințele problemei. Datorită ordonării, elementele din A vor fi testate pe rând, începând cu primul.

Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Greedy - observatii

- Metoda Greedy **nu cauta sa determine toate solutiile posibile** (care ar putea fi prea numeroase) si apoi sa aleaga din ele pe cea optima, ci cauta sa introduca direct un element **x** in solutia optima.
- Acest lucru duce la eficienta algoritmilor Greedy, insa **nu conduce in mod necesar la o solutie optima** si nici nu este posibila formularea unui criteriu general conform caruia sa putem stabili exact daca metoda Greedy rezolva sau nu o anumita problema de optimizare.

De avut în vedere

- Algoritmii de tip greedy **nu asigură** întotdeauna găsirea soluției optime, **chiar dacă la fiecare pas se determină soluția optimă pentru pasul respectiv.**
- De aceea, în momentul în care rezolvăm o problemă prin această metodă, va trebui să găsim o demonstrație matematică riguroasă referitoare la optimalitatea globală a soluției.
- De cele mai multe ori ea se face prin inducție matematică sau prin reducere la absurd.

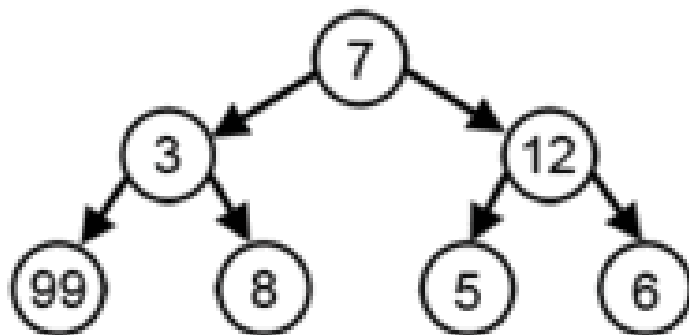
Greedy – de avut in vedere

- Putem face orice alegere pare mai bună pe moment și apoi se pot rezolva subproblemele care apar mai târziu.
- Alegerea făcută de către un algoritm *greedy* poate depinde de alegerile făcute până atunci, dar **nu de viitoarele alegeri** sau de toate soluțiile subproblemelor.
- El face iterativ o alegere *greedy* după alta, reducând fiecare problemă dată într-una mai mică.
- Cu alte cuvinte, **un algoritm greedy nu își reconsideră alegerile.**

Exemplu de esec greedy

<https://upload.wikimedia.org/wikipedia/commons/8/8c/Greedy-search-path-example.gif>

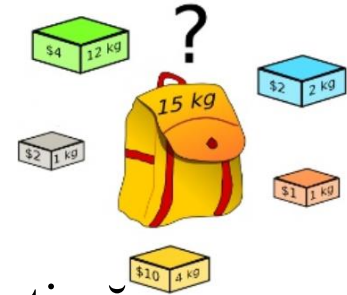
Cu obiectivul de a ajunge la suma cea mai mare, la fiecare pas, algoritmul *greedy* va alege ceea ce pare a fi soluția, așa că va alege 12 în loc de 3 la pasul al doilea, și nu va ajunge niciodată la soluția optimă, care conține 99.



Agenda

- Probleme din viata reala. Stiluri de abordare.
- Greedy – metodologie
- Forma generala a metodei greedy
- Discutii greedy, succese si esecuri
- Implementare concreta greedy

Problema rucsacului - greedy



- Deoarece se pot lua obiecte tăiate, se obține o încărcare optimă a rucsacului dacă la fiecare pas se ia obiectul cel mai valoros.
- În algoritm se calculează pentru fiecare obiect raportul valoare/greutate și se ordonează descrescător elementele șirului în funcție de acesta.
- Se extrag obiecte din șirul ordonat până când acestea vor umple rucsacul sau până se vor epuiza.
- În situația în care s-a ajuns la un obiect cu o greutate prea mare, acesta se taie.
- În algoritmul din slide-ul următor:
 - **nr_alese** contorizează numărul obiectelor alese de hoț;
 - **y** este un șir de indici, reprezentând numerele de ordine ale obiectelor alese,;
 - **Câștig** este variabila care reține rezultatul final, reprezentând câștigul total realizat de hoț.
- În momentul apelării subalgoritmului **Alege**, șirul **x** este deja ordonat descrescător în funcție de raportul valoare/greutate.

Pseudocod greedy – problema rucsacului

Subalgoritm Alege(nr_alese, y, Câștig) :

```
i ← 1                                     { primul obiect }
G_parțial ← 0                             { deocamdată greutatea împachetată este 0 }
sfârșit ← fals                             { încă nu s-a împachetat rucsacul }
cât timp (G_parțial < G) și (i ≤ n) și nu sfârșit execută:
    nr_alese ← nr_alese + 1                 { se selectează un obiect }
    y[nr_alese] ← x[i].nr
                                         { în șirul soluției se păstrează numărul de ordine al obiectului }
    dacă x[i].gr ≤ G - G_parțial atunci      { poate mai încapă ceva... }
        G_parțial ← G_parțial + x[i].gr
        Câștig_parțial ← Câștig_parțial + x[i].v
    altfel                                  { ultimul obiect se taie dacă este cazul }
        Câștig ← Câștig_parțial + (G - G_parțial) * x[i].gv
                                         { nu mai încapă nimic, intrerupem reluarea while-ului }
    sfârșit ← adevărat
    sfârșit dacă
        i ← i + 1
    sfârșit cât timp
sfârșit subalgoritm
```


Greedy rucsac - observatii

```
Subalgoritm Alege(nr_alese, y, Câștig):  
    i ← 1 { primul obiect }  
    G_parțial ← 0 { deocamdată greutatea împachetată este 0 }  
    sfârșit ← fals { încă nu s-a împachetat rucsacul }  
    cât timp (G_parțial < G) și (i ≤ n) și nu sfârșit execută:  
        nr_alese ← nr_alese + 1 { se selectează un obiect }  
        y[nr_alese] ← x[i].nr  
        { în șirul soluției se păstrează numărul de ordine al obiectului }  
        dacă x[i].gr ≤ G - G_parțial atunci { poate mai încape ceva... }  
            G_parțial ← G_parțial + x[i].gr  
            Câștig_parțial ← Câștig_parțial + x[i].v  
        altfel { ultimul obiect se taie dacă este cazul }  
            Câștig ← Câștig_parțial + (G - G_parțial) * x[i].gv  
            { nu mai încape nimic, intrerupem reluarea while-ului }  
        sfârșit ← adevărat  
    sfârșit dacă  
    i ← i + 1  
    sfârșit cât timp  
sfârșit subalgoritm
```

- Pentru anumite date de intrare problema admite mai multe soluții. Pot exista obiecte care conduc la același câștig. În acest caz ele se selectează în ordinea în care apar în șirul ordonat. Afișarea tuturor variantelor de selectare a obiectelor se poate face folosind metoda backtracking.
- Pot exista situații în care se aleg toate obiectele, dar nu se completează capacitatea rucsacului. La scrierea algoritmului trebuie să ținem cont și de acest caz.

Remarci de final

- Spre deosebire de algoritmi "lacomi", tehnicile Backtracking revin mereu inapoi, la nivel de predecesor, de aici rezultand timpul mare de executie in comparatie cu Greedy.
- Un lucru pe care il au comun ambele metode este acela ca solutia se construiește progresiv, pas cu pas.

Bibliografie suplimentara

- Antti Laaksonen, 2018, “Competitive Programmer’s Handbook”, chapter 6, pp. 57-64