Curs SDA (PC2)
Curs 13
Algoritmi

Iulian Năstac

Recapitulare
O nouă tehnică de sortare (**heapsort**)

- Noțiunea de heap a fost introdusă și folosită în anii 60 de Robert W. Floyd și J. W. J. Williams pentru crearea unui algoritm de sortare numit **heapsort**.

```
heapsort (T[1..n])
{
        make_heap(T);
        for( i = n; i ≥ 2; i - -)
                {
                 T[1] ↔ T[i];
                 cerne (T[1..i-1], 1)
                }
}
```

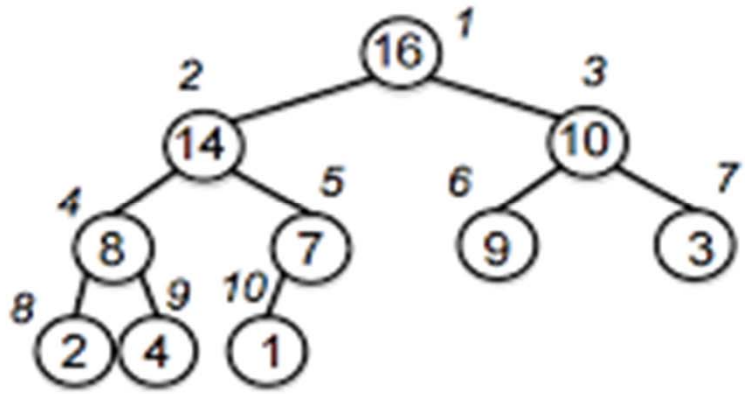
Exemplu

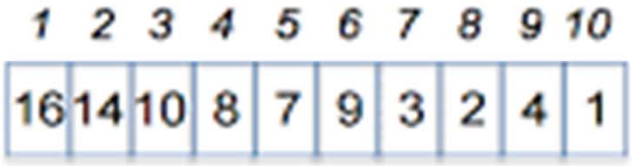
Se dorește obținerea unui arbore heap dintr-un vector nesortat

Pornim de la următorul vector (care nu este un heap):

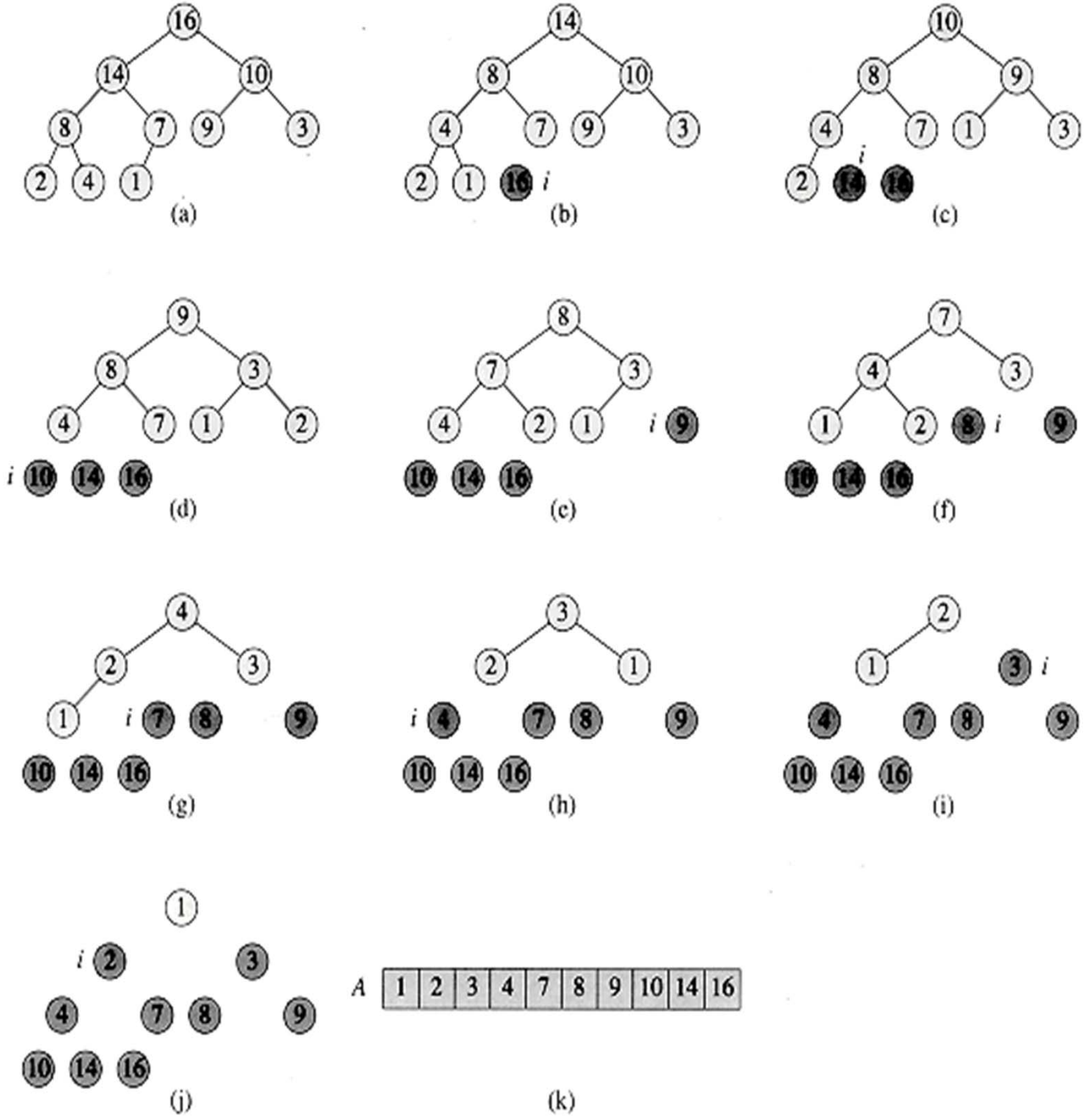
4	1	3	2	16	9	10	14	8	7
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]

După apelarea funcției **make_heap** obținem:





16	14	10	8	7	9	3	2	4	1
T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]	T[10]



Operațiile din HEAPSORT:

(a) Arborele heap imediat după ce a fost construit de funcția **make_heap**.

(b)-(j) Arborele heap imediat după fiecare apel al funcției **sift_down(T[1…i-1], 1)**. Este arătată valoarea în nodul de indice i după fiecare interschimbare. Numai nodurile legate au mai rămas în heap-ul micșorat.

(k) Rezultatul este vectorul sortat **A**

Alți algoritmi de sortare

Recapitulare

- Inserție
- Selecție
- Merge sort
- Quicksort
- etc.

Metode de sortare

- Prin **metode de sortare** se înţelege o varietate de tehnici în urma cărora sunt **ordonate**, după anumite criterii specificate, diferite *secvenţe de obiecte* (date) care prezintă o trăsătură comună. În acest scop considerăm că datele sunt o *colecţie de elemente* de un anumit tip şi fiecare element conţine o dată sau chiar mai multe, în raport cu care se realizează ordonarea. O astfel de dată se numeşte **cheie**.

Pentru limbajul de programare C, un algoritm de sortare se poate realiza prin una din următoarele metode:

1. Aranjând datele care se sortează în aşa fel încât cheile lor să corespundă ordinii dorite.

2. Ordonând un tablou de pointeri spre datele care trebuiesc sortate în aşa fel încât considerându-i pe aceştia în ordinea crescătoare a indicilor tabloului, datele spre care pointează să formeze o mulţime ordonată în acord cu ordinea dorită.

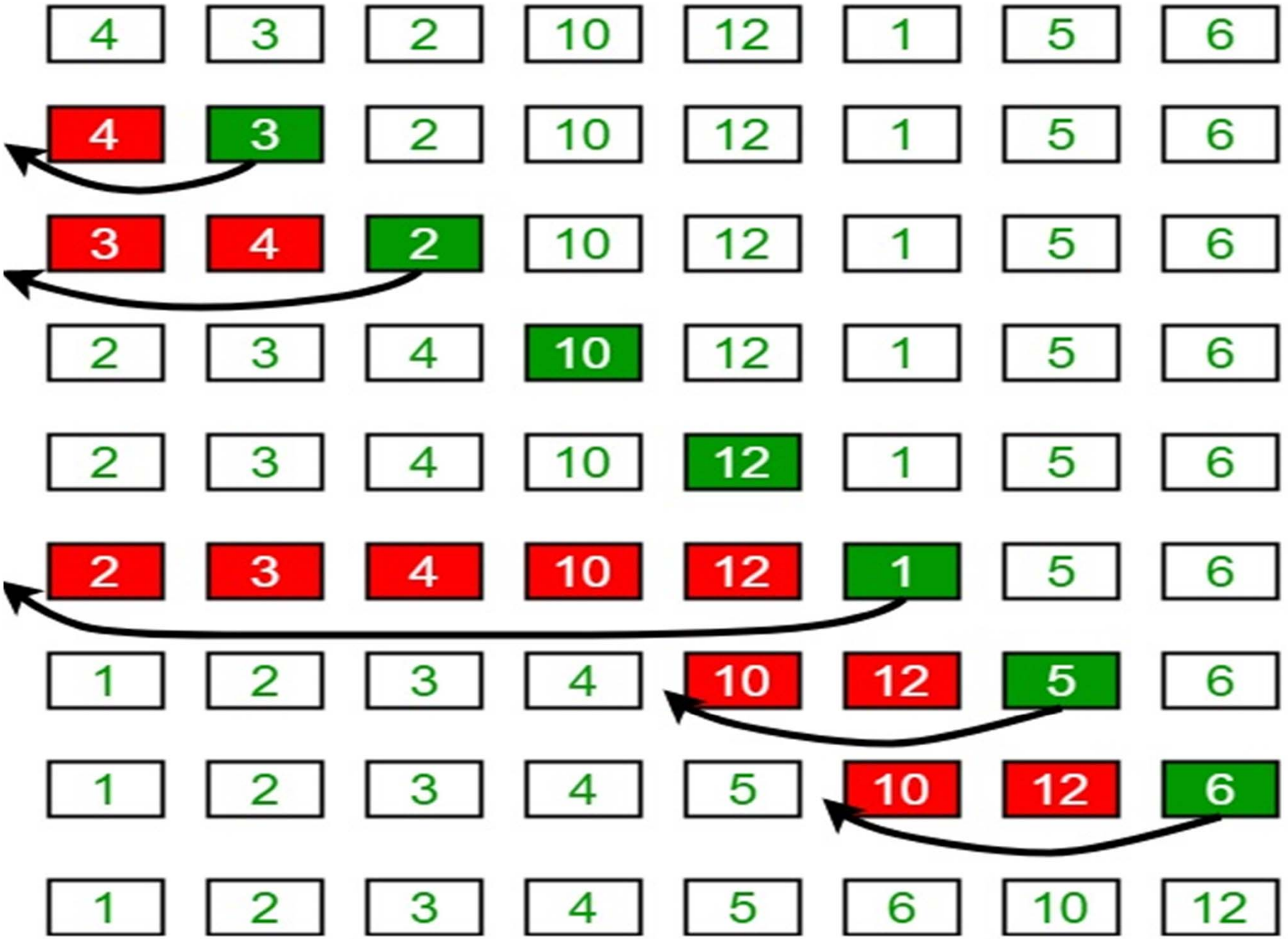
Observație:

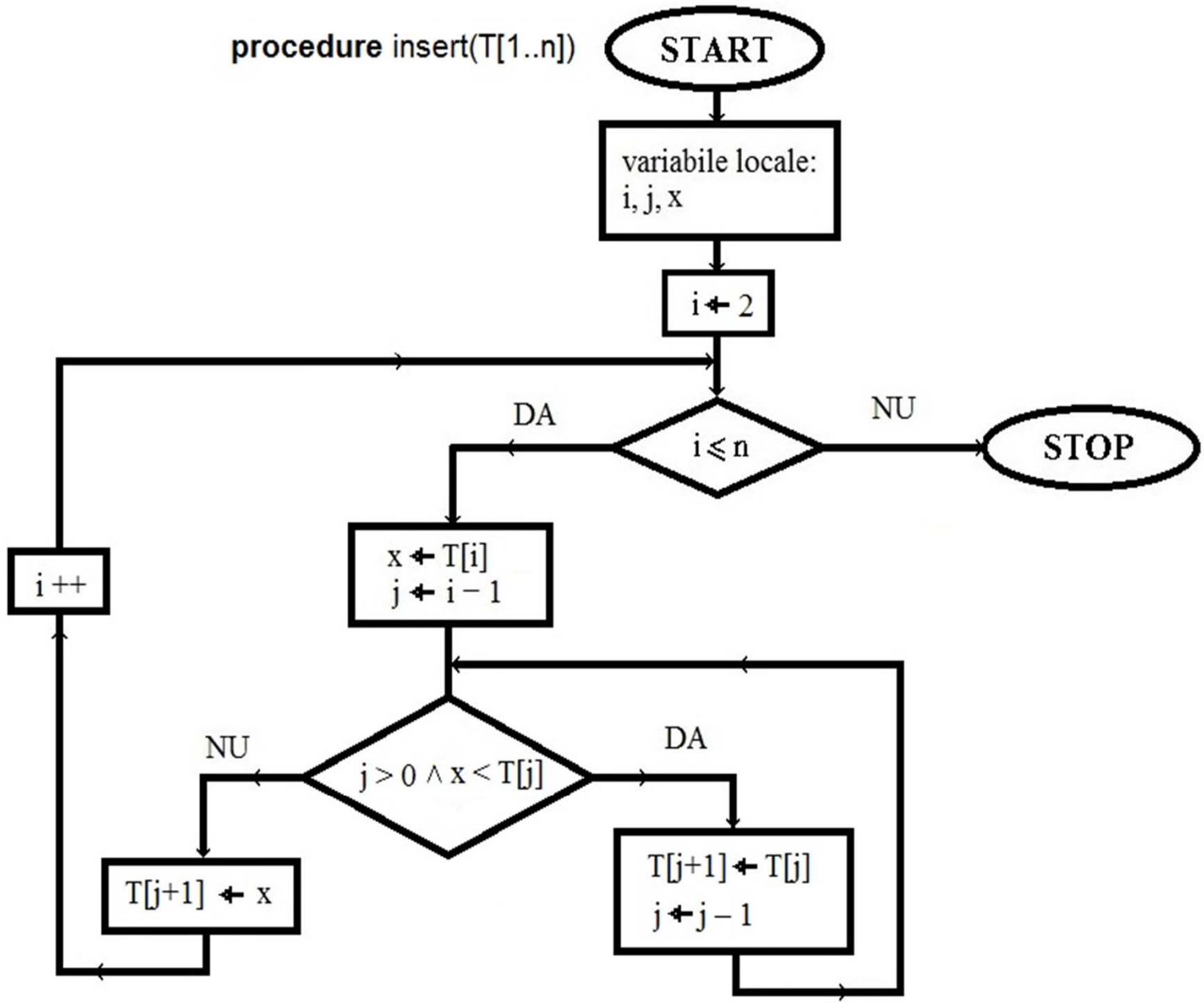
- În continuare ne vom restrânge aria de lucru la sortarea tablourilor unidimensionale (vectori) cu date numerice.

Algoritmul de sortare prin inserţie

Recapitulare

- Ideea generală a sortării ***prin inserţie*** este să considerăm pe rând fiecare element al şirului şi să îl inserăm în subşirul ordonat, creat anterior din elementele precedente. Operaţia de inserare implică deplasarea spre dreapta a unei secvenţe.


4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12


procedure insert(T[1..n])
START
variabile locale:
i, j, x
i ← 2
DA
i ⩽ n
NU
STOP
x ← T[i]
j ← i − 1
i ++
NU
j > 0 ∧ x < T[j]
DA
T[j+1] ← x
T[j+1] ← T[j]
j ← j − 1

O descriere succintă a algoritmului în limbaj pseudocod este următoarea:

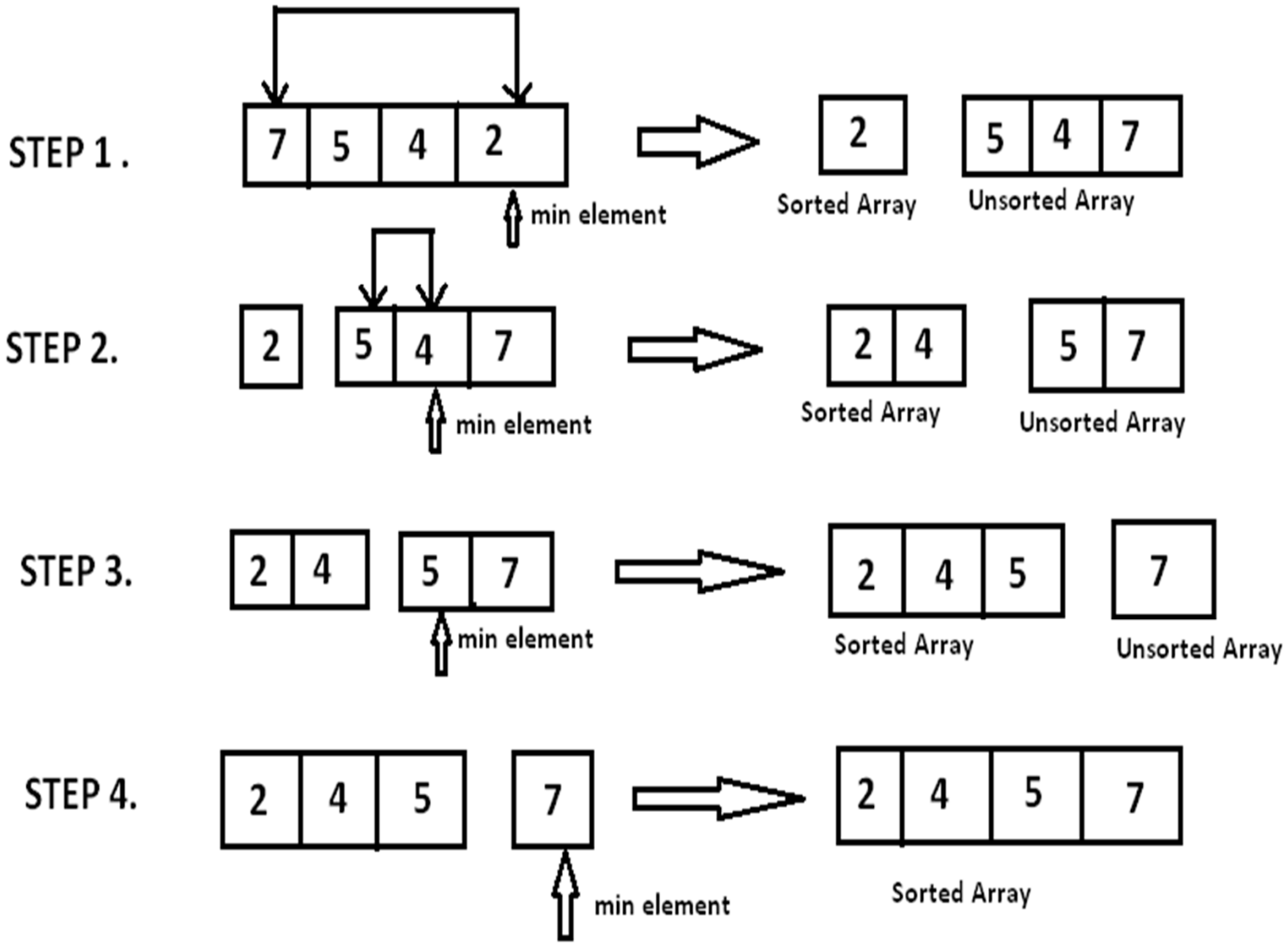
```
procedure insert(T[1..n])
{
        var. loc. i, j, x
        for i ← 2 to n do
        {
                x ← T[i]
                j ← i -1
                while (j>0 and x<T[j]) do
                        {
                         T[j+1] ← T[j]
                          j ← j – 1
                        }
                T[j+1] ← x
        }
}
```

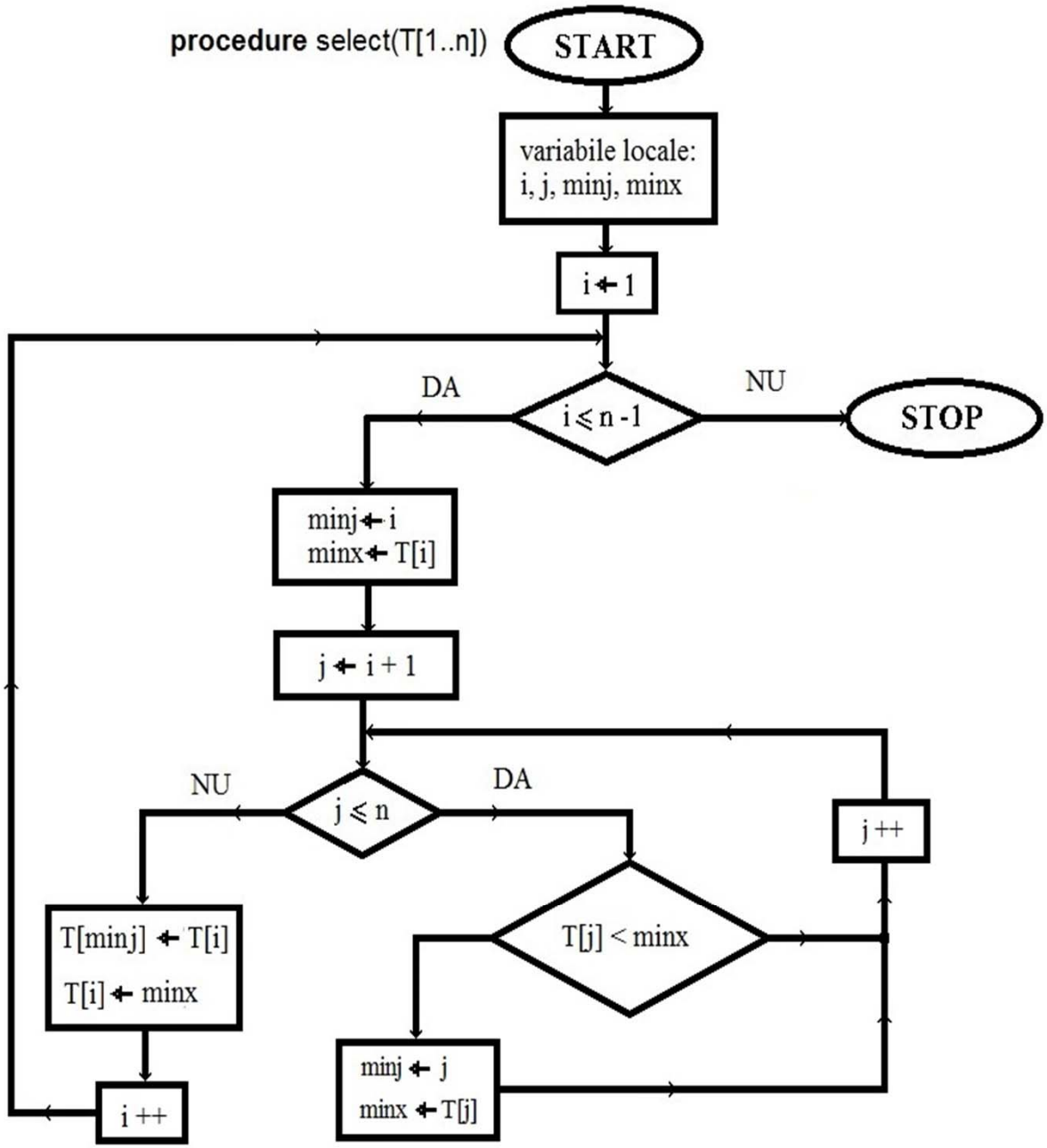
Observaţie: La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

Algoritmul de sortare prin selecţie

Recapitulare

- În cadrul algoritmului de sortare prin selecţie se plasează la fiecare pas câte un element al vectorului direct pe poziţia sa finală. Şirul sortat creşte treptat pe masură ce partea nesortată, din care se selectează noi elemente, scade corespunzător.


STEP 1 .
7
5
4
2
min element
2
Sorted Array
5
4
7
Unsorted Array
STEP 2.
2
5
4
7
min element
2
4
Sorted Array
5
7
Unsorted Array
STEP 3.
2
4
5
7
min element
2
4
5
Sorted Array
7
Unsorted Array
STEP 4.
2
4
5
7
min element
2
4
5
7
Sorted Array


procedure select(T[1..n])
START
variabile locale:
i, j, minj, minx
i ← 1
DA
i ⩽ n -1
NU
STOP
minj ← i
minx ← T[i]
j ← i + 1
NU
j ⩽ n
DA
j ++
T[j] < minx
T[minj] ← T[i]
T[i] ← minx
minj ← j
minx ← T[j]
i ++

În pseudocod, algoritmului de sortare prin selecţie este descris de următoarea secvenţă:

```
procedure select(T[1..n])
{
   var. loc. i, j, min_j, min_x,
   for i ← 1 to n-1 do
            { min_j ← i
              min_x ← T[i]
              for j ← i+1 to n do
                        if T[j] < min_x then
                                  { min_j ← j
                                    min_x ← T[j]
                                  }
              T[min_j] ← T[i]
              T[i] ← min_x
            }
}
```

Observaţie: La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

Algoritmul Bubblesort

Recapitulare

- Acest algoritm face parte din cadrul metodelor de interschimbare, în care ideea generală este de a parcurge o serie de înregistrări şi de a le interschimba pe cele care nu verifică ordinea impusă. Cea mai simplă metodă este cea de interschimbare directă, care se mai numeşte **Bubblesort** sau **metoda bulelor**.

- Metoda bulelor are la bază compararea a două chei învecinate (de indici succesivi sau din noduri succesive). Dacă acestea nu sunt în ordinea cerută, atunci se permută elementele respective sau pointerii spre ele. Deşi eficientă, această metodă prezintă un număr mare de permutări.

Algoritmul Shell

Recapitulare

- Metoda Shell a fost propusă de către Donald Shell în 1959 în urma căutării unei îmbunătăţiri a metodei bulelor. Astfel, în algoritmul Bubblesort, se compară elementele vecine şi dacă nu sunt în ordinea cerută, atunci ele se permută. De aceea, elementele care nu sunt în ordinea corectă se deplasează cu o singură poziţie. Pentru a îmbunătăţi acest procedeu, ar trebui să realizăm deplasări cu mai multe locuri ale elementelor, la o permutare a lor. Acest lucru se poate realiza dacă în loc de compararea elementelor învecinate se compară elemente aflate la o anumită distanţă între ele. În cazul în care ele nu sunt în ordinea cerută, acestea se vor permuta. Astfel, elementele se deplasează făcând salturi mai mari decât o poziţie.

- Distanţa dintre elementele comparate se numeşte increment. Incrementul se micşorează după o parcurgere a şirului de elemente care se sortează şi se reia parcurgerea de la începutul şirului. De aici rezultă şi denumirea alternativă de ***sortare cu micşorarea incrementului***.

Metoda de sortare Shell poate fi definită astfel:

1) Pornim cu un increment **inc = n/2**, unde prin **n** am notat numărul elementelor care se sortează.
2) Se realizează o parcurgere a şirului de elemente care se sortează.
3) Se înjumătăţeşte incrementul inc = inc/2 .
4) Dacă inc > 0, atunci se reia de la pasul *2)*, altfel algoritmul se opreşte.

Observaţie *Parcurgerea şirului de elemente implică următorii paşi*:

1) i = inc.
2) j = i – inc + 1 .
3) Dacă j > 0 şi elementele de ordine j şi j+inc nu satisfac criteriul de ordonare, atunci elementele respective se permută. Altfel se continuă cu pasul 6.
4) j = j – inc.
5) Se reia de la pasul 3.
6) i = i + 1.
7) Dacă i > n, se termină parcurgerea curentă a şirului. Altfel se reia de la pasul 2.

Observaţie:

- La implementarea algoritmului de mai sus în cod C va trebui avut în vedere faptul că tablourile pornesc de la indicele 0.

- Astfel iniţializarea de la punctul *2)* devine **j = i – inc** .

O perspectivă mai generală

- În informatică un algoritm înseamnă o metodă sau o procedură de calcul, alcătuită din pașii elementari necesari pentru rezolvarea unei probleme sau categorii de probleme.

- De obicei algoritmii se implementează în mod concret prin programarea adecvată a unui calculator, sau a mai multora.

- Din diverse motive există și algoritmi încă neimplementați, teoretici.

- Practic, există o nesfârșită varietate de algoritmi (pentru a le clasifica - a se vedea cărți scrise de Donald Knuth).

Un algoritm poate fi intuitiv reprezentat printr-o schemă logică

- Un algoritm este o metodă eficientă, care poate fi exprimată într-o cantitate finită de memorie și de timp, folosind o schemă logică sau un limbaj formal de calcul.

- Pornind de la o stare inițială, instrucțiunile algoritmului descriu o succesiune de calcule (stări finite) care se termină cu un rezultat (sau o varietate de rezultate).

- Trecerea de la o stare la alta, nu este în mod necesar deterministă. Unii algoritmi încorporează tranziții aleatoare.

Alți algoritmi clasici

- **Divide et Impera**
  - Divide et impera se bazează pe principiul descompunerii problemei în două sau mai multe subprobleme (mai ușoare de rezolvat), iar soluția pentru problema inițială se obține combinând soluțiile subproblemelor. De multe ori, subproblemele sunt de același tip și pentru fiecare din ele se poate aplica aceeași tactică a descompunerii în (alte) subprobleme, până când (în urma descompunerilor repetate) se ajunge la probleme care admit rezolvare imediată.
- **Greedy**
  - Greedy este un algoritm care face alegerea optima locală în fiecare etapă, cu speranța de a găsi un optim global. În multe probleme, o strategie greedy nu produce, în general, o soluție optimă, dar cu toate acestea metoda, în sine, poate oferi soluții pe plan local optime care conduc, în principiu, la o soluție globală optimală, într-un timp rezonabil.

Algoritmul divide et impera

- Divide et impera este o tehnică ce admite o implementare recursivă. Principiul general prin care se elaborează algoritmi recursivi este: "ce se întâmplă la un nivel, se întâmplă la orice nivel" (având grijă să asigurăm condițiile de terminare).
  Așadar, un algoritm prin divide et impera se elaborează astfel (la un anumit nivel avem două posibilități):

1. s-a ajuns la o problemă care admite o rezolvare imediată (condiția de terminare), caz în care se rezolvă și se revine din apel;

2. nu s-a ajuns în situația de la punctul 1, caz în care problema curentă este descompusă în (două sau mai multe) subprobleme, pentru fiecare din ele urmează un apel recursiv al funcției, după care combinarea rezultatelor are loc fie pentru fiecare subproblemă, fie la final, înaintea revenirii din apel.

Observații:

- **Divide et impera** este baza de rezolvare pentru anumite probleme, cum ar fi sortarea (quicksort, merge sort, etc.), înmulțirea numerelor mari, analiză sintactică, și de calculul transformatei Fourier discrete.
- Această tehnică poate fi uneori asociata cu algoritmul de căutare binară.
- Totuși, nu toate problemele pot fi rezolvate prin utilizarea **divide et impera**, tocmai datorită cerinței ca aplicația implicată să admită o descompunere repetată.

Algoritmul greedy

- Un exemplu clasic pentru algoritmul greedy este „problema comisului voiajor" (de complexitate NP – nonpolinomială – adică necesită un timp practic incalculabil de rezolvare completă).
- În general, algoritmii tip greedy au cinci componente:

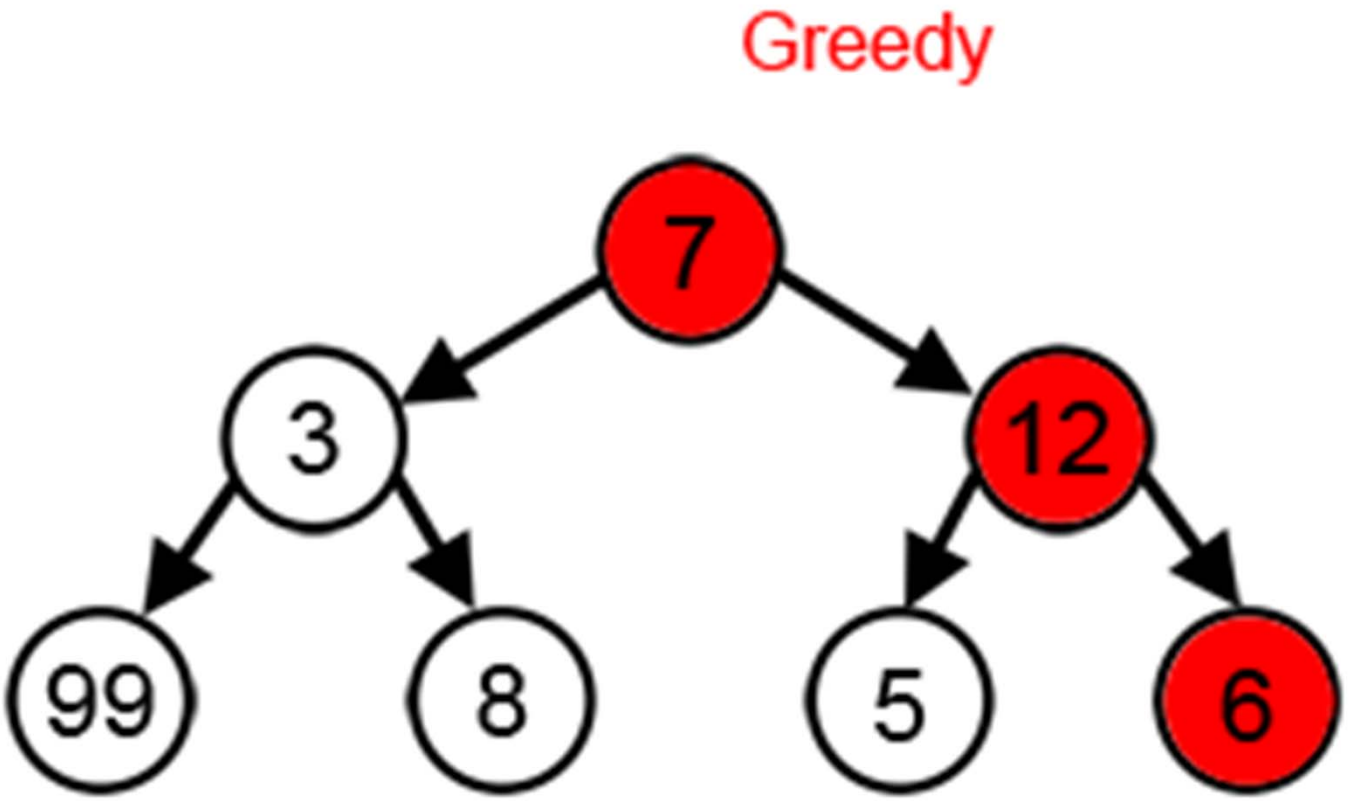
1. Un set candidat, din care este selectată o soluție
2. O funcție de selecție, care alege cel mai bun candidat la o anumită etapă
3. O funcție de fezabilitate, care este folosită pentru a determina dacă un candidat poate fi utilizată pentru a contribui la o soluție
4. O funcție obiectiv, care atribuie o valoare unei soluții
5. O funcție soluție, care va indica când am descoperit o soluție completă

Observații:

- Cu **greedy** putem face orice alegere care pare cea mai bună în prezent si apoi rezolva subproblemele care apar mai târziu.
- Alegerea momentană făcută de un **algoritm greedy** poate depinde doar de alegerile de alegerile anterioare, dar nu și cele viitoare (adică nu este o strategie globală, ci mai degrabă una de conjunctură).
- Se face iterativ câte o alegere, una după alta, reducând fiecare problema dată într-una mai simplificată. **Algoritmul greedy** nu reconsideră opțiunile sale anterioare.
- Există o diferență majoră față de **programarea dinamică**, care este exhaustivă și caută soluția globală.
- În fiecare moment, **programarea dinamică** ia decizii bazate pe toate deciziile luate în etapele anterioare și poate reconsidera cale algoritmică dintr-o etapă anterioară.

ATENȚIE!

- Algoritmii de tip Greedy (de obicei, dar nu întotdeauna) nu reușesc să găsească soluția optimă la nivel global, pentru că nu funcționează în mod exhaustiv pe toate variantele.
- Se pot face alegeri grăbite la anumite etape, prea devreme, care ulterior împiedică găsirea celei mai bune soluții de ansamblu.
- De exemplu, toți algoritmi de colorare de tip greedy pentru probleme de colorare a grafurilor (dar și alte probleme NP-complete) nu găsesc în mod constant soluții optime.
- Cu toate acestea, soluțiile greedy pot fi sunt utile, atunci când se cere un răspuns rapid.



Pornind de la rădăcină, se caută a ajunge la cea mai mare sumă finală. Algoritmul greedy va selecta ceea ce pare a fi optimul local (instantaneu), asa că va alege 12 în loc de 3 la etapa a doua, și nu va ajunge la cea mai bună soluție, care conține 99.

Perspective

- Algoritmii **greedy** sunt adesea folosiți în crearea de rețele de conexiune, ad-hoc, pentru a crea rute (trasee) eficiente pentru pachetele de date, în cel mai scurt timp posibil.
- Ideile provenite din algoritmii **greedy** sunt folosite în noi direcții cum ar fi: machine learning, business intelligence (BI) și inteligență artificială (AI).

Backtracking

- **Backtracking** este numele generic al unui grup de algoritmi de descoperire a tuturor soluțiilor unei probleme de calcul.
- Un astfel de algoritm se bazează pe construirea incrementală de soluții posibile (denumite **candidat**), abandonând fiecare candidat parțial imediat ce devine clar că acesta nu are șanse să devină o soluție validă.

Observații:

- Un backtracking la limită este o problemă de căutare într-un arbore binar (**pre**, **in** sau **post** - ordine).

- Exemplul de bază folosit în numeroase publicații este problema reginelor, care cere să se găsească toate modurile în care pot fi așezate pe o tablă de șah opt regine astfel încât să nu se atace. (vezi problema de laborator de pe site http://www.euroqual.pub.ro/wp-content/uploads/sda_lab_06_backtracking.pdf)

Backtracking
(alte observații)

- Backtracking depinde de:

  - procedurile de tip "cutie neagră" construite de utilizator care definesc problema care trebuie rezolvată,

  - de natura candidaților parțiali,

  - de modul în care acestea sunt extinse în candidați complete.

- Este un algoritm metaheurist mai degrabă decât un algoritm specific.

- Totusi spre deosebire de multe alte meta-heuristici, este garantată găsirea tuturor soluțiilor la o problemă finită într-o perioadă limitată de timp.

Analiza eficienței algoritmilor

- Analiza eficienței unui algoritm are ca scop estimarea volumului de *resurse de calcul* necesare pentru execuția algoritmului.

Prin resurse se poate înțelege:

- ***Spațiul de memorie*** necesar pentru stocarea datelor pe care le prelucrează algoritmul.

- ***Timpul*** necesar pentru execuția tuturor prelucrărilor specificate în algoritm.

Studiul algoritmilor cuprinde mai multe aspecte:

- Elaborarea algoritmilor
- Exprimarea algoritmilor
- Validarea algoritmilor
- Testarea programelor:
  - **Depanarea** (debugging) – corectare erori
  - **Trasarea** – procesul executării unui program corect pe diferite date de test pentru a determina timpul de calcul și memoria necesară (rezultatele pot fi introduse în analiza algoritmilor)
- **Analiza algoritmilor**

Problema testării

- Se poate dovedi a fi extrem de dificilă pentru un soft de mari dimensiuni.
- Cazurile folosite pentru testare nu pot acoperi întotdeauna situațiile posibile.
- Necesită un timp apreciabil și resurse importante de calcul.

“Prin depanare putem evidenția prezența erorilor, dar nu și absența lor. O demonstrație a faptului că un program este corect poate fi mai valoroasă decât o mie de teste, deoarece garantează faptul că acel program va funcționa corect în orice situație.”

Edsger Wybe Dijkstra
(considerat unul dintre fondatorii Metodelor Formale)

Eficiența algoritmilor

- Ideal este, ca pentru o problemă dată, să găsim mai mulți algoritmi, iar apoi să-l alegem pe cel optim…

- … adică să-l găsim pe cel mai eficient implementabil la momentul curent (folosind resursele disponibile).

Un algoritm se poate analiza:

- **Aposteriori** (empiric) – comportarea algoritmului după implementarea și rularea pe sistemul de calcul a unor cazuri diferite.

- **Apriori** (teoretic) – înaintea implementării prin determinarea cantitativă a resurselor (timp, memorie, etc.)

Observații:

- Dezavantajul analizei aposteriori: din motive practice un algoritm nu poate fi testat pentru cazuri oricât de mari.

- Analiza apriori:
  - nu depinde de calculatorul sau limbajul ales
  - salvează timpul pierdut cu programarea și rularea unui algoritm ineficient
  - permite studiul eficienței algoritmului pentru cazuri de orice mărime

Creșterea vitezei

- Schimbarea calculatorului (cu unul mai performant) poate conduce la rezolvarea unei probleme de 100 de ori mai repede (pentru aceleași date de intrare) $\rightarrow$ creșterea este liniară

- … dar numai un algoritm eficient poate îmbunătăți dramatic rezolvarea unei probleme de mari dimensiuni.

Ne concentrăm asupra timpului

- Obiectivul major al eficienței algoritmilor este criteriul **timp de execuție**.

- Alte resurse necesare (cum ar fi memoria) pot fi estimate teoretic în mod similar.

Definiție:

- Vom spune că un algoritm necesită un timp de ordinul lui ***t*** dacă există o constantă pozitivă ***c*** și o ***implementare*** (cod program) a acestui algoritm, capabilă să rezolve fiecare caz al problemei în cel mult ***c·t(n)*** secunde, unde ***n*** este mărimea cazului considerat.

Observație:

Constanta ***c*** depinde de fapt (în cea mai mare măsură) de frecvența de ceas a mașinii de calcul.

Notația asimptotică

- Notăm cu:

R_+ - mulțimea numerelor reale pozitive

N - mulțimea numerelor naturale

Fie $f : N \to R_+$ o funcție arbitrară

Definim mulțimea (numită ***ordinul lui f***):

$$O(f) = \{t : N \to R_+ \mid (\exists c \in R_+^*)\ si\ (\exists n_0 \in N^*)\ \ astfel\ \ încât\ \ (\forall n \geq n_0)\ \ avem\ \ \ t(n) \leq c \cdot f(n)\}$$

Observaţie:

Se poate vorbi despre ordinul lui ***f*** chiar dacă ***f(n)*** este negativă sau nedefinită pentru $n \leq n_0$

Principiul invarianţei

Două implementări diferite ale aceluiaşi algoritm nu diferă în eficienţă cu mai mult decât o constantă multiplicativă.

Exemplu:

Având două implementări diferite care necesită $t_1(n)$, respectiv $t_2(n)$ secunde pentru a rezolva un caz de mărime n, rezultă că:

$$\exists c \in R_+^* \quad astfel \quad încât \quad t_1(n) \leq c \cdot t_2(n) \quad \forall n \geq n_0$$

Observaţie

$$t \in O(t)$$

Uzual se va căuta cea mai simplă funcţie f astfel încat $t \in O(f)$

Se observă imediat că:

$n \in O(n)$, $n^2 \in O(n^2)$, $n^2 \in O(n^3)$, $n^3 \notin O(n^2)$

Putem spune că:

$$\forall\ f : N \to R_+^*\ \ dacă\ \ f \in O(n) \Rightarrow\ f^2 \in O(n^2)$$

$$\forall\ f : N \to R_+^*\ \ dacă\ \ f \in O(n) \Rightarrow\ 2^f \in O(2^n)$$

Definim relaţii de ordine parţială:

- $f \leq g$ dacă $O(f) \subseteq O(g)$

- $f < g$ dacă $O(f) \subset O(g)$

Proprietăţi:

P1: Tranzitivitate

Dacă $f \in O(g)$ şi $g \in O(h)$ atunci $f \in O(h)$

P2: Dacă $f \in O(g)$ atunci $O(f) \subseteq O(g)$

P3: $\forall\ f\ şi\ g : N \to R_+^*\ avem:$

a) $O(f) = O(g) \Leftrightarrow f \in O(g)$ şi $g \in O(f)$

b) $O(f) \subset O(g) \Leftrightarrow f \in O(g)$ şi $g \notin O(f)$

P4:

$$O(f+g) = O(\max(f,g))\ pentru\ \forall\ f\ şi\ g : N \to R_+^*$$

unde suma şi produsul se iau punctual.

Definim relaţia de echivalenţă

$$f \equiv g \quad \text{dacă} \quad O(f) = O(g)$$

Observaţie: În mulţimea $O(f)$ putem înlocui pe f cu orice funcţie echivalentă cu f

Exemplu:

lg(n) ≡ ln(n) ≡ log_2(n) sau ***O(lg(n))=O(ln(n))=O(log_2n)***

Demonstraţia este simplă fiindcă arătăm că există o constantă ***c*** astfel încât: ***lg(n) ≤ c·ln(n)***

Deoarece: $$\log_a x = \frac{\log_b x}{\log_b a}$$

Rezultă: $$c \geq \frac{\lg(n)}{\ln(n)} = \frac{\frac{\ln(n)}{\ln(10)}}{\ln(n)} = \frac{1}{\ln(10)}$$

Dacă notăm cu O(1) ordinul funcţiilor mărginite superior de o constantă, atunci putem obţine uşor ierarhia:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset ... \subset O(2^n)$$

Această ierarhie corespunde unui criteriu al performanţelor.

Pentru o problemă dată, dorim mereu să obţinem un algoritm corespunzător cu un ordin cât mai la stânga ierarhiei menţionate.

Exemplu:

$$n^3 + 3 \cdot n^2 + n + 8 \in O(n^3 + (3 \cdot n^2 + n + 8)) = O(max(n^3, (3 \cdot n^2 + n + 8))) = O(n^3)$$

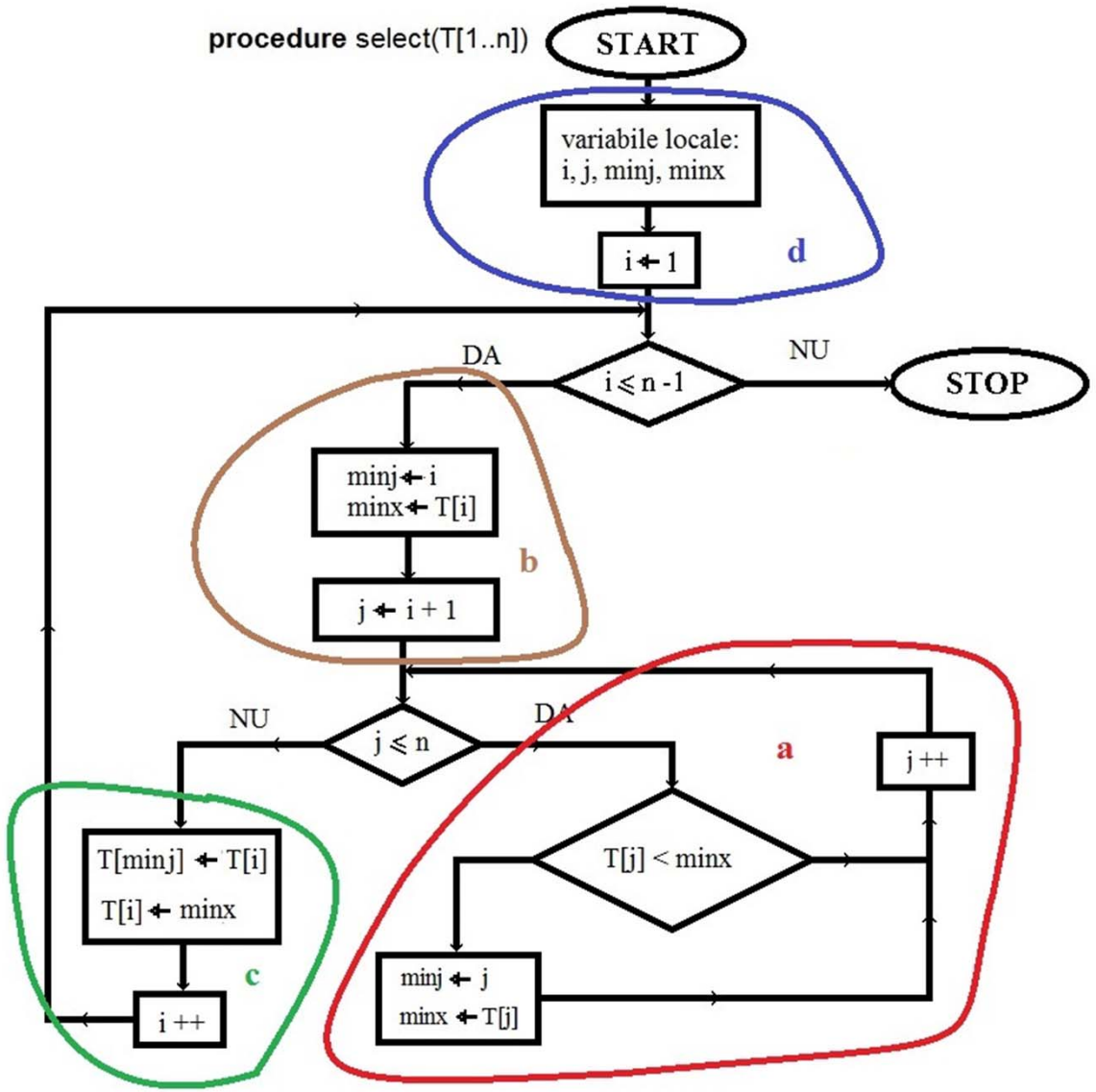
Observaţie:

max(n^3 , ($3 \cdot n^2$ + n + 8)) va fi ***n^3*** pentru un număr suficient de mare (creșterea, pe grafic, a lui ***n^3*** este mai rapidă decât a oricărui polinom de gradul 2, chiar dacă pornește de la valori mai mici)

Tehnici de analiză a algoritmilor

- Nu există o formulă generală pentru analiza eficienţei algoritmilor.
- Aceasta este o chestiune de raţionament, intuiţie şi experienţă.
- Schema logică poate fi utilă pentru a găsi ordinul unui algoritm.

Sortarea prin selecţie



- Timpul pentru o singura execuţie a buclei interioare poate fi mărginit superior de o constantă ***a***.

- Pentru un ***i*** dat, bucla interioară necesită un timp de cel mult

$$b + a\cdot(n - i) \text{ unităţi}$$

unde ***b*** este o constantă ce reprezintă iniţializarea buclei.

-O singură execuţie a buclei exterioare are loc în cel mult

$$c + b + a\cdot(n - i) \text{ unităţi}$$

unde ***c*** este o altă constantă.

Algoritmul durează în final:

$$T_{\max} = d + \sum_{i=1}^{n-1} (c + b + a(n - i)) \quad \text{unităţi}$$

$$T_{\max} = a \cdot n \cdot \sum_{i=1}^{n-1} 1 - a \cdot \sum_{i=1}^{n-1} i + (c+b) \cdot (n-1) + d$$

dar

$$\sum_{i=1}^{n-1} i = \frac{(n-1+1) \cdot (n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \qquad \text{şi} \qquad \sum_{i=1}^{n-1} 1 = n-1$$

Rezultă:

$$T_{\max} = \tfrac{a}{2} \cdot n^2 - \tfrac{a}{2} \cdot n + (b+c-a) \cdot (n-1) + d - c - b$$

Prin urmare algoritmul de sortare prin inserţie necesită un timp de execuţie de ordinul ***O(n²)***