1. PROIECTAREA ALGORITMILOR

1.1.METODA BACKTRACKING

Este una dintre cele mai cunoscute metode de elaborare a algoritmilor. Ea se aplică acelor probleme în care soluția se poate reprezenta sub forma unui vector $x=(x_1,\ldots,x_n)\in A_1x\ldots xA_n$, unde A_i, $i=\overline{1,n}$ sunt finite ($|A_i|=n_i$, $i=\overline{1,n}$). În plus, pentru fiecare problemă în parte este necesar ca soluția $x_1,\ldots,x_n$ sa satisfacă anumite condiții interne $\rho(x_1,\ldots,x_n)$.

Mulțimea $A=A_1x\ldots xA_n$ este spațiul soluțiilor posibile. Elementele $x\in A$ care satisfac condițiile interne se numesc soluții rezultat. Ne propunem determinarea tuturor soluțiilor rezultat, eventual pentru a alege dintre ele pe cea care minimizează sau maximizează o funcție obiectiv.

Metoda Backtracking evită generarea tuturor soluțiilor posibile (toate elementele produsului cartezian $A_1x\ldots xA_n$). Elementului $x_k\in A_k$, $k=\overline{1,n}$ i se atribuie o valoare după ce au fost atribuite valori pentru componentele $x_1\in A_1,\ldots, x_{k-1}\in A_{k-1}$. Metoda trece la atribuirea unei valori pentru $x_{k+1}\in A_{k+1}$ doar dacă x_k împreună cu $x_1,\ldots,x_{k-1}$ verifică condițiile de continuare, notate $\rho_k(x_1,\ldots,x_k)$. Dacă condițiile $\rho_k(x_1,\ldots,x_k)$ sunt îndeplinite se trece la atribuirea unei valori pentru elementul $x_{k+1}\in A_{k+1}$ al soluției. Neîndeplinirea condițiilor exprimă faptul că oricum am alege $x_{k+1}\in A_{k+1},\ldots,x_n\in A_n$ nu vom putea ajunge la o soluție rezultat în care condițiile interne să fie îndeplinite. În acest ultim caz va trebui să facem o altă alegere pentru x_k, iar acest lucru este posibil doar dacă nu am epuizat toate valorile posibile din mulțimea A_k. Dacă

mulțimea A_k a fost epuizată va trebui sa micşorăm pe *k*, *k←k-1* şi să trecem la alegerea unei alte valori pentru elementul $x_{k-1} \in A_{k-1}$.

Algoritmul Backtracking este următorul.

Intrare: - mulțimile $A_1,...,A_n$

-condiția internă $\rho(x_1,...,x_n)$ pentru soluția $(x_1,...,x_n) \in A_1 \times ... A_n$, exprimată prin $\rho_i(x_1,...,x_n)$, $i=\overline{1,n}$, condițiile de la pasul *i*.

Ieşire: soluțiile rezultat (x_1, . . . , x_n).

procedure posibil (x,k;v)

x:vect. Solutiilor rezultat $x=(x_1,...,x_n)$

k: pasul curent, care semnifică căutarea unei noi valori $\alpha \in A_k$ care va fi atribuita componentei x_k a solutiei.

v: variabila booleană cu semnificația v= true ⇔ ($\exists\ \alpha \in A_k$ neatribuită pentru x_k si $\rho_k(x_1,...,x_{k-1},\alpha)$= true) şi false ⇔ caz contrar

w: variabila booleană auxiliară.

repeat

[1] v←(„$\exists\ \alpha \in A_k$ neatribuită înca pentru x_k”);

w←true

if v **then**

[2] $x_k \leftarrow \alpha$

if not $\rho_k(x_1,...,x_k)$ then w←false **end_if**

end_if

until (not v) or (v and w)

```
        v←(v and w)
      end_proc

    function este_sol (k):boolean
        if (k=n) then return true
        else return false
        end_if
    end_function
```

Partea principală a algoritmului este

```
k←1; xk←^3αk initial şi αk initial∉Ak
while (k>0)
  posibil (x, k v)
  if not v then k←k-1
  else
    if este_sol (k) then tipareste (x1,...,xn)
    else
      k←k+1; xk←^3αk initial, αk initial∉Ak
    end_if
  end_if
end_while
```

Un caz particular al problemelor backtracking este acela în care mulțimile A_i, $i=\overline{1,n}$ conțin elemente aflate în progresie aritmetică. Astfel, pentru mulțimea A_i notăm:

a_{i1}: primul element

h_i: rația

b_i: ultimul element

În acest caz algoritmul se modifica după cum urmează:

Instrucțiunea etichetată cu 1 devine $v \leftarrow (x_k < b_k)$

Instrucțiunea etichetată cu 2 devine $x_k \leftarrow x_k + h_k$

Instrucțiunea etichetată cu 3 devine $x_k \leftarrow a_{k1} - h_k$

1.1.1. Exemple de probleme backtracking

Exemplul 1: *Problema colorării hărților*

Să se coloreze o hartă reprezentând n țări folosind m culori etichetate 1,...,m.

Rezolvare: Datele necesare pentru descrierea hărții vor fi reprezentate de o matrice A_{nxn} cu semnificația

$a_{ij} = 1$, dacă țara i are frontieră comună cu țara j

$a_{ii}=0$, în caz contrar

Fie $C=\{c_1,...,c_m\}$ mulțimea culorilor cu $c_i=v$, $v=\overline{1,m}$.

Vectorul soluției rezultat $x=(x_1,...,x_n) \in Cx...xC$ (de n ori), are semnificația: țara i are repartizată culoarea $x_i \in C$, $i=\overline{1,n}$. Inițial, fiecărei țări i se va atribui culoarea $0 \notin C$.

La pasul k încercăm să atribuim o nouă culoare (x_k+1) pentru țara k ($k=\overline{1,n}$), dacă sunt îndeplinite condițiile:

1. $x_k < m \Leftrightarrow$ se mai pot găsi alte culori pentru țara k.
2. $\forall i$, $1 \le i < k$ cu $a_{i,k}=1$ avem $x_i \neq x_k+1$

Dacă condițiile 1 şi 2 sunt îndeplinite, atunci noua culoare pentru țara k va fi x_k+1 şi se poate trece la stabilirea unei noi

culori pentru ţara *k+1*. Astfel, ţării *k* nu i se mai poate atribui altă culoare şi ne întoarcem la alegerea unei noi culori pentru ţara *k-1*. Algoritmul se termină c`nd nu se mai poate atribui culoare pentru ţara *1*.

Exemplul 2: *Problema damelor*

Pe o tablă de şah cu $n \times n$ pătrate, să se aşeze *n* dame aşa încât să nu se atace reciproc.

Rezolvare: Dacă vom considera matricea $A=(a_{i,j})_{i,j}$ $I,j=\overline{1,n}$ care să reprezinte tabla de şah şi două dame în poziţiile a_{ij} si a_{kl}, atunci cele două dame se atacă reciproc dacă $i=k$ sau $j=l$ sau $|i-k|=|j-l|$.

Din prima relaţie deducem faptul că fiecare damă trebuie să fie aşezată pe o linie separată. În continuare vom avea grijă să nu fie îndeplinite celelalte două relaţii.

Notăm $M=\{1,\dots,n\}$ mulţimea celor *n* coloane ale tablei de şah. Vectorul soluţiei rezultat $x=(x_1,\dots,x_n)\in M\times\dots\times M$ (unde produsul este de *n* ori) are semnificaţia: componenta *i* a vectorului ($i=\overline{1,n}$) reprezintă linia *i* de pe tabla de şah. Valoarea x_i a acestei componente reprezintă coloana în care se aşează dama de pe linia *i*.

Iniţial, toate damele sunt în afara tablei de şah, deci $x_i=0$, $i=\overline{1,n}$.

La pasul *k* vom încerca aşezarea damei *k* pe linia *k* şi în coloana $x[k]+1$. Iniţial, dama *k* este în afara tablei ($x[k]=0\notin M$). În general, vom putea aşeza dama într-o altă coloană dacă vechea poziţie satisface condiţia $x_k<n$. De fiecare dată noua poziţie va trebui să satisfacă cond. interne ρ_k:

$\forall i,\ 1\le i<k$ avem $x_i\neq(x_k+1)$ (nu sunt pe aceeaşi coloană), şi

$|i-k| \neq |x_i-(x_k+1)|$ (nu sunt pe diagonală)

Dacă $x_k<n$ şi ρ_k sunt adevarate, dama de pe linia k se aşează în coloana x_k+1 şi se trece la aşezarea damei $k+1$. Astfel, dama k nu poate fi aşezată pe tabla şi va trebui să reluăm aşezarea damei $k-1$. Algoritmul se termină când dama 1 nu mai poate fi aşezată pe tablă (se încearcă aşezarea damei în coloana 0).

Exemplul 3:*Problema comis-voiajorului*

Un comis-voiajor trebuie sa viziteze n oraşe etichetate $1,\ldots,n$. Va pleca din oraşul 1 şi se va întoarce tot în 1, trecând prin fiecare oraş (în afară de oraşul 1, celelalte oraşe trebuie vizitate o singură dată). Cunoscând legăturile existente între oraşe, să se tipărească toate drumurile posibile.

Rezolvare: Datele referitoare la drumurile dintre oraşe vor fi reprezentate într-o matrice A_{nxn} cu:

$a_{ij}=1$, există drum direct între oraşul i şi oraşul j

$a_{ii}=0$, caz contrar

Mulțimea oraşelor o notăm cu $N=\{1,2,\ldots,n\}$.

Vectorul soluțiilor rezultat va fi $x=(x_1, \ldots , x_n, x_{n+1})$ cu $x_1=x_{n+1}=1$ şi cu semnificația: pe poziția i ($i=\overline{1,n+1}$) se vizitează oraşul $x_i \in N$.

La pasul k $(2 \leq k \leq n)$ se încearcă vizitarea oraşului $x[k]+1$ dacă:

$1 \leq x[k]<n$ (inițial $x[k]=1$) şi

$a[x[k-1], x[k]+1]=1$ (pentru $k=n$ trebuie şi $a[1, x[k]+1]=1$)

În plus trebuie să fie satisfăcute condițiile interne:

ρ_k : $\forall i,\ 1 \leq i<k$; $x_k \neq x_i$

Dacă toate condițiile sunt îndeplinite atunci $x[k] \leftarrow x[k]+1$ şi se poate trece la vizitarea oraşului k+1.

În caz contrar cu oraşul de pe poziția x_k nu ajungem la o soluție rezultat şi va trebui să ne întoarcem la o altă alegere pentru oraşul de pe poziția *k-1*.

Exemplul 4: Să se descompună numărul natural *n* în toate modurile posibile ca sumă de *p* numere diferite ($p \le n$).

Rezolvare: Vectorul soluțiilor rezultat este $x=(x_1, \ldots, x_p)$, $\sum_{i=1}^{p} x_i=n$.

***Observație*:**

1. Cel mai mare număr ce poate fi atribuit componentelor x_i, $i=\overline{1,p}$ este n-p+1, pentru că $n=n-p+1+(1+1+\ldots+1)$ unde suma din paranteză este de *p-1* ori.

2. Trebuie să evităm determinarea unor soluții care se deosebesc între ele doar prin poziția numerelor din descompunere, ex.

5=2+3, 5=3+2, 5=1+4, 5=4+1

Pentru aceasta vom genera pe poziția *k* ($k=\overline{1,p}$) doar valori x_k care satisfac: $k \le x_k \le n-p+1$

Astfel va obține soluții în care componentele sunt în ordine crescătoare.

Fie $N_k=\{k, \ldots, n-p+1\}$, $k=\overline{1,p}$

Atunci $x=(x_1,\ldots,x_p) \in N_1 x N_2 x \ldots x N_p$. Inițial $x_k=k-1$, $k=\overline{1,p}$

La pasul *k* ($k=\overline{1,p}$), valoarea de pe poziția *k* este x_k. Vom putea schimba acea valoare cu x_k+1 doar dacă

(1) $k-1 \le x_k < n-p+1 \Leftrightarrow \exists\, \alpha \in N_k$ *neatribuităîncă lui* x_k

(2) $\rho_k : \forall i,\ 1 \le i < k,\ x_i \ne x_k+1$

Dacă (1) şi (2) sunt adevarate atunci $x_k \leftarrow x_k+1$ şi se trece la poziția $k+1$. Altfel, orice valoare pentru x_k nu ne va conduce la o soluție rezultat şi va trebui să ne întoarcem la atribuirea unei alte valori pentru poziția $k-1$.

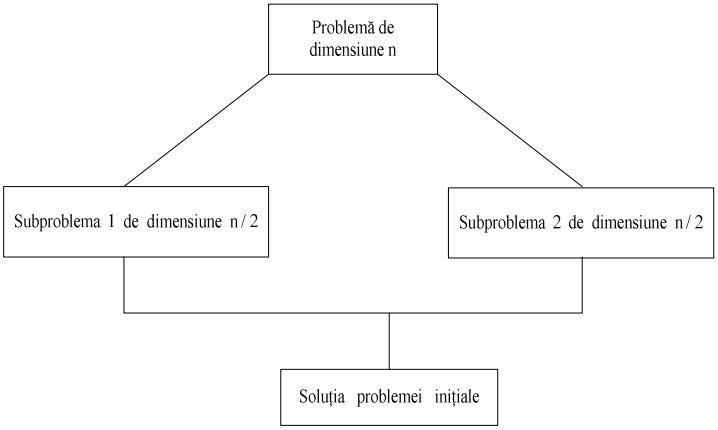
Algoritmul se încheie când epuizăm toate valorile pentru poziția $k=1$.

1.2. METODA DIVIDE-ET-IMPERA

Strategia acestei metode este următoarea:

1. Împarte instanța problemei într-una sau mai multe instanțe mai mici.
2. Rezolvă fiecare instanță mai mică. Dacă o instanță mai mică nu este suficient de mică pentru a putea fi rezolvată, atunci utilizează recursia pentru a face aceasta.
3. Determină soluția problemei inițiale prin combinarea soluției la subprobleme.

Un caz particular este descris de următoarea diagramă.



Altfel spus, fie vectorul $A = \{a_1, ..., a_n\}$ reprezentând o mulțime ale cărei elemente trebuie prelucrate conform unei anumite probleme. Presupunem că pentru $\forall p,q \in \{1, ..., n\}$ indici cu $p < q$, $\exists\ m \in \{p, p+1, ..., q\}$ astfel încât prelucrarea secvenței $\{a_p, ..., a_q\}$ se poate face prelucrând separat secvențele $\{a_p, a_{p+1}, ..., a_m\}$ şi $\{a_{m+1}, ..., a_q\}$. Apoi, combinând soluțiile se determină soluția pentru cazul secvenței $\{a_p, ..., a_q\}$.

Algoritmul este următorul.

procedure divide_et_impera *(p, q, α)*

p,q : indici din mulțimea $\{1, ..., n\}$, $p < q$

α : soluția problemei la pasul curent

prel (p, q, α): procedura care obține soluția problemei pentru submulțimea *{a_p, ... , a_q}*

împarte (p, q, m): procedura care determină indicele $p \le m \le q$ necesar pentru impărțirea problemei

combină (β, γ, α): combină soluțiile β si γ obținând soluția α.

if $(q - p \le eps)$ **then** *prel* (p, q, α)

else

împarte (p, q, m)

divide_et_impera (p, m, β)

divide_et_impera $(m+1, q, \gamma)$

combină (β, γ, α)

end_if

end_proc

Cazul general este de a împărți problema de dimensiune n în a instanțe de dimensiune $\frac{n}{b}$ şi având nevoie de un timp $f(n)$

pentru operațiile de împărțire şi de combinare a soluțiilor de la subprobleme.

Dacă notăm cu $T(n)$ timpul necesar problemei de dimensiune n, formula generală de recurență pentru metoda Divide-et-Impera este:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

unde $f(n) = f_D(n) + f_C(n)$, şi unde f_D este timpul necesar pentru impărțirea in subprobleme iar f_C este timpul necesar pentru interclasare. Putem considera că în general avem $f(n) \in O\left(n^k\right)$.

Exemplu: Căutarea binară: $a = 1$, $b = 2$, $f(n) = 1 \Leftrightarrow \mathrm{O}(1)$. Deci $T(n) = T\left(\frac{n}{2}\right) + \mathrm{O}(1)$.

1.2.1. Exemple de probleme Divide-et-Impera

1.2.1.1. Problema turnurilor din Hanoi

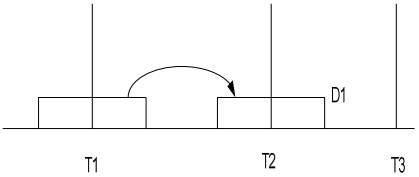
Se dau 3 tije numerotate 1, 2, 3 şi n discuri perforate şi de dimensiuni diferite. Inițial toate discurile sunt aşezate pe tija *1* în ordinea descrescătoare a diametrelor considerând sensul de la baza tijei spre vârful ei. Să se mute toate discurile pe tija *2* în aceeaşi ordine, folosind tija *3* ca tijă intermediară şi respectând următoarele reguli:

– la fiecare pas, se mută un singur disc din vârful unei tije;

– pe fiecare tijă, la orice pas, deasupra oricărui disc, există numai discuri de diametre mai mici.

Rezolvare Simbolizăm mutarea unui disc de pe tija i pe tija j, unde $i, j \in \{1, 2, 3\}$ prin perechea (i, j), $i \neq j$.

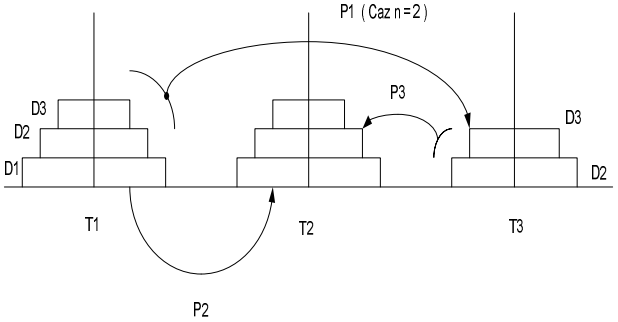
Cazul $n = 1$:



$\Rightarrow (1,2)$

Cazul $n=2$: necesită trei paşi.
P1 : se mută discul D_2 de pe tija T_1 pe tija T_3;
P2 : se mută discul D_1 de pe tija T_1 pe tija T_2
P3 : se mută discul D_2 de pe tija T_3 pe tija T_2.

Cazul $n=3$:



Observăm că se poate face abstracție inițial de discul D_1 şi se mută $n=2$ discuri (D_2 şi D_3) de pe T_1 pe T_3 folosind ca tijă intermediară, tija $k=6-1-3=2$. Revenim la discul D_1 mutându-l pe tija T_2. În final, se procedează din nou ca în cazul $n=2$ pentru discurile D_2 şi D_3, mutându-le de această dată de pe tija T_3 pe tija T_2 şi utilizând ca tijă intermediară tija $k=6-3-2=1$. Dacă notăm cu $H(n,i,j)$ şirul mutărilor necesare la pasul n (mutarea de pe tija i pe tija j), atunci $H(n,\ i,\ j)=\begin{cases}(i,j) & dacă\ n=1\\ H(n-1,\ i,\ k),\ (i,j),\ H(n-1,k,\ j) & dacă\ n>1\end{cases}$, unde $k=n-i-j.$

Dacă notăm cu $H(n)$ numărul de mutări efectuate, atunci:

$$H(n) = H(n-1)+1+H(n-1) = 2H(n-1)+1 =$$
$$= 2\cdot(2\cdot H(n-2)+1)+1 = 2\cdot(2\cdot(2\cdot H(n-3)+1)+1)+1 = ...$$
$$= 2^i \cdot S(n-i) + \sum_{j=0}^{i-1} 2^j$$

Pentru $i = n$ avem $S_{(0)} = 0$ şi $S_{(1)} = 1$.

Deci $H(n) = 2^n \cdot S_{(0)} + 2^n - 1 = 2^n - 1$.

Complexitatea este $O(2^n)$.

1.2.1.2. Sortare prin interclasare

```
procedure MergeSort (A[ ], temp[ ], left, right)
  Intrare: tabloul A[0....n] care va fi sortat crescător
           tabloul temp[0....n] ajutător
           left, right – indici în tablouri
  Ieşire: tabloul A sortat.
  begin
     if (right > left) then
        mid ← (right + left) / 2
        MergeSort (A, temp, left, mid)
        MergeSort (A, temp, mid + 1, right)
        Merge (A, temp, left, mid + 1, right)
     end_if
  end
end_proc
```

procedure Merge (*A[ ], temp[ ], left, mid, right*)

Intrare / Ieşire: tabloul A cu elementele interclasate între părţile left

```
            şi right
Intrare: tabloul ajutător temp[ ]
        left, mid, right – indici în A
begin
    left_end ← mid – 1
    tmp_pas ← left
    nr_elem ← right – left + 1
    while ((left ≤ left_end) and (mid ≤ right)) do
        if (A [left] ≤ A [mid]) then
              temp[tmp_pas] ← A[left]
              tmp_pas ← tmp_pas + 1
              left ← left + 1
        else
              temp[tmp_pas] ← A[mid]
              tmp_pas ← tmp_pas + 1
              mid ← mid + 1
        end_if
    end_while
    while (left ≤ left_end) do
        temp[tmp_pas] ← A[left]
        left ← left + 1
        tmp_pas ← tmp_pas + 1
    end_while
    while (mid ≤ right) do
        temp [tmp_pas] ← A[mid]
        mid ← mid + 1
        tmp_pas ← tmp_pas + 1
    end_while
    for i=0 to nr_elem do
         A[right] ← temp [right];
        right ← right – 1
end_proc
```

Observații:

1. Algoritmul necesită o memorie externă – tablou temporar - pentru a memora datele interclasate.

2. Algoritmul necesită şi acțiuni suplimentare de copiere din tabloul temporar în tabloul inițial.

Teorema Master: *Fie* $a \geq 1$, $b > 1$ *constante întregi. Fie* $f(n)$ *o funcție cu* $f(n) \in O(n^k)$ *şi fie* $T(n)$: $N_+ \to R$ *(sau* N*) definită prin recurența* $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

Atunci $T(n)$ *poate fi delimitată asimptotic după cum urmează:*

1. *Dacă* $a > b^k$, *atunci* $T(n) \in O\left(n^{\log_b^a}\right)$.

2. *Dacă* $a = b^k$, *atunci* $T(n) \in O\left(n^k \log n\right)$.

3. *Dacă* $a < b^k$, *atunci* $T(n) \in O\left(n^k\right)$.

Vom aplica teorema master pentru a calcula complexitatea algoritmului de sortare prin interclasare.

Complexitatea împărțirii problemei este $f_D(n) \in O(1)$ iar complexitatea interclasării este $O(n)$. Rezultă $f(n) \in O(n)$.

Deci $k = 1$.

Numărul de subprobleme este $a = 2$.

Dimensiunea tablourilor corespunzătoare subproblemelor este $\left\lfloor \frac{n}{2} \right\rfloor$ sau $\left\lceil \frac{n}{2} \right\rceil \Rightarrow b = 2$.

Formula de recurență este $T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(n)$, iar dacă $n = 2^p$ avem $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$. Din $a = 2$, $b = 2$, $k = 1$ rezultă $T(n) \in O(n \log n)$.

1.3. METODA GREEDY

Această metodă se aplică problemelor în care se dă o mulțime A cu n elemente şi se cere să se determine o submulțime $B \subseteq A$ care să îndeplinească anumite condiții pentru a putea fi acceptată. Este posibil ca să existe mai multe mulțimi acceptabile (*soluții posibile*). De aceea se mai furnizează şi un criteriu prin care, dintre soluțiile posibile, se alege una singură numită *soluție optimă*.

Soluțiile posibile au proprietatea: dacă B este soluție posibilă şi $C \subset B$, atunci şi C este soluție posibilă. Prin definiție, Φ este soluție posibilă.

1.3.1. Algoritmul Greedy - 1

***Intrare*:** mulțimea A, $|A| = n$.

***Ieşire*:** mulțimea B – soluția optimă.

Alege (A, i, x): alege la pasul i din mulțimea A, elementul x. Alegerea se face conform unui criteriu care, în final să conducă la soluția optimă.

Posibil (B, x, v): furnizează în $v \in \{true, false\}$ răspunsul la întrebarea: „ $B \cup \{x\}$ este sau nu soluție posibilă ? ”

Adaug (x, B): adaugă la B elementul x, $B \leftarrow B \cup \{x\}$.

$B \leftarrow \Phi$

for $i = 1$ to n **do**

Alege (A, i, x)

Posibil (B, x, v)

if (v) **then** A*daug* (x, B)

end_if

next *i*

În această versiune, criteriul de alegere este aplicat la fiecare pas. Algoritmul poate suferi o modificare. Vom ordona elementele mulțimii A conform criteriului de alegere stabilit. Urmează ca apoi elementele din A să fie extrase în această ordine.

1.3.2. Algoritmul Greedy – 2

***Intrare*:** mulțimea A, $|A| = n$.

***Ieşire*:** mulțimea B – soluție optimă.

Perm (A): permută elementele lui A ordonându-le conform criteriului de alegere stabilit.

Posibil (B, x, v) şi *Adaug* (x, B) sunt aceleaşi ca la algoritmul precedent.

$B \leftarrow \Phi$; perm (A)

for $i = 1$ to n **do**

P*osibil* (B, a_i, v)

if (v) **the**n *Adaug* (a_i, B)

end_if

next *i*

1.3.3. Exemple de probleme Greedy

1.3.3.1. Memorarea textelor pe benzi

a) Fiind date *n* texte de lungimi diferite $L_1,.....,L_n$ să se înmagazineze aceste texte pe o bandă suficient de lungă. Ştiindu-se că citirea textului k se face după citirea textelor $1,.....,k-1$ să se aşeze aceste texte aşa încât timpul de citire pentru fiecare din ele să fie minim.

Rezolvare: Fie o poziționare a textelor pe bandă dată de permutarea $p(1,2,....,n)=(p_1,....,p_n)$ în care textul k apare al p_k–lea.

În general, citirea unui text de lungime L_k necesită un timp direct proporțional cu $L_k : T_k \sim L_k$.

Atunci citirea textului k, care se află pe poziția p_k, se va face după textele $p_1,....,p_{k-1}$ şi va avea timpul $\sum_{i=1}^{k} Lp_i$.

Timpul mediu de regăsire a unui text va fi:

$$\frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{n} L_{pi} = \frac{1}{n}\sum_{k=1}^{n}(n-k+1)\,L_{p_k}$$

Considerăm funcția: $\rho(p) = \sum_{k=1}^{n}(n-k+1)\,L_{pk}$, unde p este permutare pentru $\{1,...,n\}$ şi să determinăm permutarea p care minimizează această funcție.

Criteriul de alegere: Dacă am ales textele $p_1,....,p_{k-1}$ şi le–am aşezat pe bandă în această ordine, atunci cea mai bună

alegere pentru textul p_k va fi textul cel mai scurt dintre textele $\{1,2,...,n\}\setminus\{p_1, p_2,..., p_{k-1}\}$. Astfel suntem conduşi la alegerea textelor în ordinea crescătoare a lungimilor lor.

Propoziție. *Fie* $p=(1,2,...,n)$ *o permutare dată de ordinea crescătoare* $L_1 \le L_2 \le .... \le L_n$. *Atunci poziționarea în această ordine este optimă.*

Demonstrație: Fie $p=(p_1,...,p_n)$ o poziționare optimă. Presupunem, prin absurd, că $\exists\, i<j$ astfel încât $L_{pi} \ge L_{pj}$. Fie p' obținută din p schimbând p_i cu p_j. Cele două permutări diferă doar cu două poziții. Atunci:

$$\rho(p') - \rho(p) = (n-i+1)(L_{pj} - L_{pi}) + (n-j+1)(L_{pi} - L_{pj}) =$$
$$= (j-i)(L_{pj} - L_{pi}) \le 0$$

Rezultă $\rho(p') \le \rho(p)$, dar $\rho(p)$ este minimă. Rezultă $\rho(p') = \rho(p)$. Deci p' este poziționare optimă.

Aplicând acest raționament de mai multe ori, trecem din permutare în permutare până ajungem la permutarea identică $p=(1,2,...,n)$ dată de $L_1 \le L_2 \le .... \le L_n$.

Rezolvare practică: Un tablou de şiruri de caractere care se va ordona crescător.

b) Cele n texte se vor aşeza pe m benzi.

Rezolvare: Poziționarea celor n texte pe m benzi va corespunde unei partiții a mulțimii $\{1,....,n\} = P_1 \cup .... \cup P_m$, iar timpul mediu de regăsire a unui text pe una din benzi este:

$\frac{1}{m}\sum_{k=1}^{m}\rho(Pk)$. Vom lua $\rho(P)=\sum_{k=1}^{m}\rho(Pk)$, $P=P_1\cup....\cup P_m$. Dorim să găsim partiția P care minimizează $\rho(P)$.

Criteriu de alegere: Presupunem că am poziționat $k-1$ texte. Vom alege din cele $n-k+1$ texte rămase pe cel mai scurt. Îl vom aşeza pe banda cea mai liberă.

Vom considera textele în ordinea crescătoare a lungimilor $L_1\le....\le L_n$ şi vom aşeza textul i pe banda $\bmod(i-1,m)+1$ în continuarea celor deja aflate pe această bandă. Rezultă că pe această bandă vor fi aşezate textele $i+m,\ i+2m,.....$, în total $\bmod(n-i,m)+1$ texte.

Propoziție. *Fie textele ordonate crescător $L_1\le....\le L_n$. Aşezarea textului i pe banda $\bmod(i-1,m)+1$, $i=\overline{1,n}$ este o poziționare optimă.*

Demonstrație: Aplicând metoda de mai sus rezultă că pe fiecare bandă textele sunt aşezate în ordine crescătoare (cele mai lungi m texte vor fi ultimele pe cele m benzi).

Fie $u_i\equiv$ numărul de texte ce urmează textului i pe banda unde acesta se află. Fie P o partiționare optimă.

Dacă $n\ge m$ atunci nici o bandă nu este goală (am putea muta un text de pe o bandă cu minim două texte pe banda goală şi am obține o partiționare P' mai bună decât P).

În condiția $n\ge m$ cele mai lungi m texte apar pe ultima poziție de pe cele m benzi. Valoarea n corespunzătoare lor este 0. Să dovedim acest lucru. Presupunem prin absurd ca există un text i, ultimul pe o bandă şi mai există pe o altă bandă textele j şi k (ultimele în această ordine) astfel încât: $L_i<L_j$ şi $L_i<L_k$ (unde

$L_j < L_k) \Leftrightarrow L_i < L_j$ şi $u_i = 0$ dar $u_j = 0$. Schimbând textele i şi j între ele am obține o partiționare P' strict mai bună decât P, contrazicând ipoteza.

Deci cele mai lungi m texte au $u = 0$. Următoarele m texte ca lungime vor avea $u = 1$. În general, textul i din ordinea $L_1 \leq .... \leq L_n$ va avea $u_i = \bmod (n-i,\ m)$.

Rezolvare practică: Cele m benzi vor fi m linii ale unui tablou bidimensional de şiruri de caractere (string-uri).

1.3.3.2. Problema rucsacului

Cu ajutorul unui rucsac în care se poate încărca greutatea maximă G, trebuie transportate obiectele $O_1, ..., O_n$. Obiectul i are greutatea g_i şi pentru transportul lui în întregime se câştigă $c_i \left(i = \overline{1,n}\right)$. Să se determine ce obiecte trebuie transportate pentru a se realiza câştigul maxim.

Rezolvare: Se disting două cazuri:

1. *Cazul continuu*: Putem lua din obiectul i orice parte $x_i \in [0,1]$.
2. *Cazul discret*: Orice obiect i poate fi încărcat doar în întregime sau deloc, $x_i \in \{0,1\}$.

Fie $x = (x_1, ..., x_n)$ vectorul soluție în ambele cazuri. Soluția optimă va trebui să satisfacă:

Cazul continuu: $\sum_{i=1}^{n} g_i x_i = G$

şi

Cazul discret : $\sum_{i=1}^{n} g_i x_i \le G$

În ambele cazuri soluția optimă trebuie să maximizeze funcția de câştig:

$$f(x) = \sum_{i=1}^{n} c_i x_i$$

Pentru a ajunge la soluția optimă vom considera în ambele cazuri obiectele în ordinea descrescătoare a câştigurilor unitare $\frac{c_i}{g_i}$, $i = \overline{1,n}$.

Studiem pe rând cele două cazuri.

Cazul continuu

Presupunem ordinea $\frac{c_1}{g_1} \ge \frac{c_2}{g_2} \ge .... \ge \frac{c_n}{g_n}$ şi vom încerca să încărcăm pe cât posibil obiectele în întregime, în ordinea considerată, până când ajungem la greutatea G. Vectorul $x = (x_1, ..., x_n)$ este *soluție posibilă* dacă $x_i \in [0,1]$, $i = \overline{1,n}$; $\sum_{i=1}^{n} g_i x_i \le G$. Soluția posibilă este *optimă* dacă maximizează funcția de câştig (1).

procedure Rucsac (n, G, g, c, x, C)

n: numărul de obiecte; G: greutatea maximă ce poate fi dusă în rucsac;

g: array [1...n] - vectorul greutăților obiectelor;

c: array [1...n] - vectorul câştigurilor corespunzătoare fiecarui obiect;

x: array [1...n] - vectorul soluției optime;

C: câştigul total.

$C \leftarrow 0,\ i = 1$

while $(G > 0)$ and $(i \le n)$ **do**

if $g_i < G$ **then**

$x_i \leftarrow 1,\ G \leftarrow G - g_i,\ C \leftarrow C + c_i,\ i \leftarrow i + 1$

else

$x_i \leftarrow \dfrac{G}{g_i},\quad G \leftarrow 0$

for $j = i+1$ to n **do** $x_j \leftarrow 0$

next j

end_if

end_while

end_proc

Observăm că în urma procedurii *Rucsac* vectorul soluție este $x = (1, \ldots, 1, x_j, 0, \ldots, 0)$, unde $1 \le j \le n$, j-unic şi $x_j = (0, 1]$. Avem $\sum_{i=1}^{n} g_i x_i = G$. Ne propunem să arătăm că x este soluție optimă.

Demonstrație. Fie $y=(y_1,....,y_n)$ soluția optimă $\Leftrightarrow \sum_{i=1}^{n} g_i x_i = G$ şi presupunem $y \neq x$. Atunci, fie $k \leq n$ cel mai mic indice, astfel încât $y_k \neq x_k$ $\left(y_i = x_i,\ i = \overline{1,\ k-1},\ y_k \neq x_k\right)$. Pornind de la soluția y vom construi o nouă soluție optimă y' cu proprietatea $y_i' = x_i$, $i = \overline{1,\ n}$. Repetând procedeul până când $k=n$ vom ajunge la o soluție optimă care coincide cu x.

Observații:

1. $k \leq j$. Dacă presupunem $k > j$, atunci $\sum_{i=1}^{n} g_i x_i > G$ ($\sum_{i=1}^{n} g_i x_i > \sum_{i=1}^{n} g_i x_i = G$) şi y nu ar mai fi soluția posibilă.

2. $y_k < x_k$. Când $k > j$, avem $x_k = 1$ şi $y_k \neq x_k \Rightarrow y_k < 1 = x_k$. Când $k = j$, putem presupune $y_k > x_k \Rightarrow \sum_{i=1}^{n} g_i y_i > \sum_{i=1}^{n} g_i x_i > G$ (fals). Deci $y_k < x_k$.

Fie $y' = (y_1, ..., y_{k-1}, x_k, y'_{k+1}, ..., y'_k) = (1, ..., 1, x_k, y'_{k+1}, ..., y'_n)$ cu $y'_i = \varepsilon\, y_i$, $i = \overline{k+1, n}$, astfel încât greutatea maximă obținută în cazul lui y să fie menținută şi pentru y'. Notăm $\rho(\varepsilon)=[g_k x_k+\varepsilon \sum_{i>k} g_i y_i]-[g_k y_k+ \sum_{i>k} g_i y_i]$

$\rho(1)=g_k(x_k-y_k)>0$

$\rho(0)=g_k x_k-g_k y_k-\sum_{i>k} g_i y_i=\sum_{i=1}^{k-1} g_i x_i+g_k x_k-\sum_{i=1}^{k-1} g_i y_i-g_k y_k-$

$\sum_{i>k} g_i y_i=\sum_{i=1}^{k} g_i x_i-\sum_{i=1}^{n} g_i y_i \le G-G=0$

$\rho(1)>0$ şi $\rho(0)\le 0 \Rightarrow \exists\ \varepsilon\in[0,1)$ astfel încât $\rho(\varepsilon)=0$

$\Rightarrow g_k x_k+\sum_{i>k} g_i y_i = g_k y_k+\sum_{i>k} g_i y_i$

$\Rightarrow \sum_{i=1}^{n} g_i y'_i=\sum_{i=1}^{n} g_i y_i \Rightarrow$ y' este soluția posibilă.

Ştim că *f(y)* este maximă.

$f(y') - f(y)=c_k x_k+\sum_{i>k} c_i y_i'-c_k y_k-\sum_{i>k} c_i y_i=c_k(x_k-y_k)-\sum_{i>k} c_i(y_i-y_i')=c_k/g_k[g_k(x_k-y_k)-\sum_{i>k} c_i g_k/c_k(y_i-y_i')]$

Dar, cum $\frac{c_1}{g_1}\ge ....\ge\frac{c_n}{g_n} \Rightarrow \frac{g_1}{c_1}\le ....\le\frac{g_n}{c_n}$. Rezultă că pentru $i>k$, avem $\frac{g_i}{c_i}>\frac{g_k}{c_k}$. Înlocuind $\frac{g_k}{c_k}$ cu $\frac{g_i}{c_i}$ obținem:

$f(y')-$

$f(y)\ge c_k/g_k[g_k(x_k y_k)+\sum_{i>k} g_i(y_i y_i')]=c_k/g_k[(g_k x_k+\sum_{i>k} g_i y_i')-$

$(g_k y_k+\sum_{i>k} g_i y_i)]=c_k/g_k\rho(\varepsilon)=0$

Rezultă $f(y')\ge f(y)$ şi $f(y)$ maximă. Rezultă $f(y')=f(y)$. Deci y' este soluția optimă.

Cazul discret

Vectorul $y=(y_1,....,y_n)$ este soluție posibilă dacă $y_i \in \{0,1\}$ şi $\sum_{i=1}^{n} g_i y_i > G$. Soluția posibilă y este optimă dacă $f(y)=\sum_{i=1}^{n} c_i y_i$ este maximă ($\forall\ k=\overline{1,n}$ cu $y_k=0$, dacă $y_k<1 \Rightarrow \sum_{i=1}^{n} g_i y_i > G+g_k > G,\ i \neq k$).

procedure Rucsac_discret $(n, G, g, c; x, c)$

$G_i \leftarrow 0,\ C \leftarrow 0$

for $i=1$ to n **do**

if $(G_1+g_i<G)$ **then**

$x_i \leftarrow 1$

$G_1 \leftarrow G_1+g_i,\ \ C \leftarrow C+c_i$

else $x_i \leftarrow 0$

end_if

next *i*

end_proc.

Din algoritmul *Rucsac_discret* rezultă că $\forall\ k=\overline{1,n}$ cu $x_k=0$, avem $\sum_{i=1}^{n} g_i x_i \le G$ şi $\sum_{i=1}^{n} g_i x_i + g_k > G$. Ne propunem să arătăm că vectorul $x=(x_1,....,x_n)$ obținut de algoritmul *Rucsac_discret* este soluția optimă.

Fie $y=(y_1,....,y_n)$ soluție optimă. Presupunem $y \neq x$.

Fie k primul indice astfel încât $x_k \neq y_k \Leftrightarrow x_i = y_i$, $i = \overline{1, k-1}$. Atunci sunt două posibilități:

1. $\begin{cases} x_k = 0 \Rightarrow \sum_{i=1}^{k-1} g_i x_i + g_k > G \\ y_k = 1 \Rightarrow \sum_{i=1}^{k-1} g_i y_i + g_k \leq G \end{cases}$ Contradicție,

deci acest caz nu este posibil.

2. $\begin{cases} x_k = 1 \Rightarrow \sum_{i=1}^{k-1} g_i x_i + g_k \leq G \\ y_k = 0 \Rightarrow \sum_{i=1}^{k-1} g_i y_i + g_k > G \end{cases}$ Contradicție,

deci nici cazul doi nu este posibil. Atunci $x_k = y_k \Rightarrow x = y \Rightarrow x$ este soluția optimă.

1.4. METODA PROGRAMĂRII DINAMICE

Această metodă se utilizează la rezolvarea unor probleme de optimizare care se referă la un proces. Procesul parcurge stările $s_0, s_1, ..., s_n$ (s_n este stare finală, s_0 este stare inițială).

La fiecare trecere din starea s_{i-1} în starea s_i se ia decizia d_i, $i = \overline{1, n}$ pentru a se realiza acest lucru. La fiecare pas i, decizia d_i poate fi aleasă din mai multe decizii posibile, dar cea care se ia trebuie să fie optimă. În general deciziile $d_1, ..., d_n$ care conduc la soluția problemei trebuie să fie optime satisfăcând principiul optimalității (R. Bellmann). Acest principiu spune că, dacă stărilor $s_0, s_1, ..., s_n$ le corespund deciziile $d_1, ..., d_n$ optime atunci, oricare ar fi i, la şirul de stări $s_0, s_1, ..., s_n$ le corespund aceleaşi decizii optime $d_1, d_2, ..., d_i$, iar la şirul de stări $s_i, s_{i+1}, ..., s_n$ corespund aceleaşi decizii optime $d_{i+1}, ..., d_n$.

Dacă principiul optimalității este satisfăcut, metoda programării dinamice presupune scrierea unei relații de recurență pentru decizia de la pasul. În general relațiile de recurență sunt de două tipuri:

- $d_i = f(d_1, d_2, ..., d_{i-1})$ reprezentând *procedeul înapoi*
- $d_i = f(d_{i+1}, d_{i+2}, ..., d_n)$ reprezentând *procedeul înainte*

Funcția f determină decizia de la pasul i ținând seama de trecutul procesului (în primul caz) sau de viitorul procesului (în al doilea caz).

1.4.1. Procedeul înapoi

***Problemă*:** Fie două cuvinte s respectiv t cu m şi respectiv n litere. Să se transforme cuvântul s în cuvântul t utilizând operațiile:

"a": adăugarea unei litere;

"m": modificarea unei litere;

"s": ştergerea unei litere.

Transformarea să se facă prin utilizarea unui număr minim de operații. Să se afişeze costul transformării (numărul de operații) şi cuvintele intermediare.

Rezolvare: O soluție banală, care bineînțeles nu este optimă, este cea în care la sfârşitul cuvântului s se adaugă literele cuvântului t şi apoi se şterg primele m litere ale cuvântului s, inițial. O altă soluție, care în continuare nu este optimă dar mai bună decât prima, s-ar aplica în cazul în care $m > n$. În acest caz primele n litere din s s-ar modifica prin literele din t şi apoi s-ar şterge ultimele $m-n$ litere. Vom rezolva problema folosind procedeul înapoi al programării dinamice.

Considerăm o matrice $\text{cost}(m+1) \times (n+1)$ ale cărei elemente cost_{ij} au semnificația: (numărul de transformări) costul transformării primelor i caractere ale cuvântului s în primele j caractere ale cuvântului t. În final ne interesează $\text{cost}(m,n)$, aceasta fiind valoarea minimă de operații cerută de problemă.

Să urmărim modul de calcul al valorii cost_{ij} :

– dacă litera de pe poziția i a cuvântului s este egală cu litera de pe poziția j a cuvântului $t (s_i = t_j)$ atunci $\cos t_{ij} = \cos t_{i-1, j-1}$ (nu se face nici o operație în plus față de pasul anterior);

– dacă $s_i \neq t_j$ atunci:

1. Transformarea primelor i litere din s în primele j litere din t poate deriva din transformarea primelor $i-1$ litere din s în primele $j-1$ litere din t la care se adaugă operația de modificare a literei i din s, s_i, cu litera j din t, t_j. Atunci: $\cos t_{ij} = \cos t_{i-1, j-1}$.

2. Transformarea primelor i litere din s în primele j litere din t poate deriva din transformarea primelor $i-1$ litere din s în primele j litere din t la care se adaugă operația de ştergere a literei s_i (litera de pe poziția i a cuvântului s). Atunci: $\cos t_{ij} = \cos t_{i-1, j} + 1$ (s_i se şterge).

3. Transformarea primelor i litere din s în primele j litere din t poate deriva din transformarea primelor i litere din s în primele $j-1$ litere din t la care se adaugă operația de *"adăugare"* a unei litere. Mai precis, la cuvântul intermediar obținut în s la pasul anterior, se adaugă pe poziția j, litera t_j.

Atunci: $\cos t_{ij} = \cos t_{i-1, j} + 1$. În final:

$$\cos t_{ij} = \begin{cases} \cos t_{i-1, j-1} & \text{dacă } s_i = t_j \\ 1 + \min \{\cos t_{i-1, j-1}, \cos t_{i-1, j}, \cos t_{i, j-1}\} \end{cases}$$

Se observă că la fiecare pas se alege varianta care conduce la un număr minim de transformări. Altfel spus, la pasul curent valoarea $\cos t_{ij}$ este minimă posibilă. Pentru realizarea algoritmului vom considera liniile şi coloanele matricei $\cos t$ numerotate $0, \ldots, m$ respectiv $0, \ldots, n$.

Coloana 0 va avea valorile:

$\cos t_{0,0} = 0$: costul transformării cuvântului vid în cuvântul vid;

$\cos t_{1,0} = 1$: costul transformării cuvântului s_1 în cuvântul vid (o ştergere);

$\cos t_{m,0} = m$ costul transformării cuvântului $s_1, \ldots, s_m$ în cuvântul vid (m ştergeri).

Linia 0 va avea valorile:

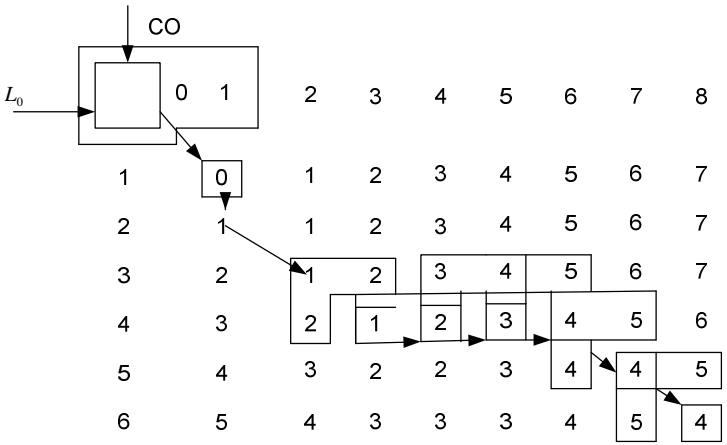
$\cos t_{0,0} = 0$;

$\cos t_{0,1} = 0$ costul transformării cuvântului vid în cuvântul t_1 (o adăugare);

$\cos t_{0,n} = n$ costul transformării cuvântului vid în cuvântul $t_1, \ldots, t_n$ (n adăugiri)

Exemplu. Să transformăm cuvântul *"Corina"* în cuvântul *"Cristina"*.

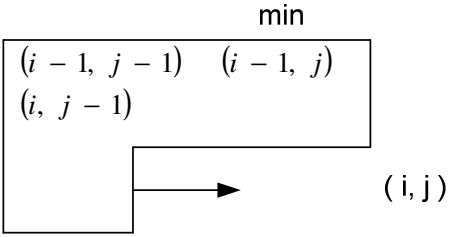
Matricea $\cos t$ va fi:



$\cos t_{6,8} = 4$ reprezintă numărul minim de transformări ale cuvântului *"Corina"* în cuvântul *"Cristina "*.

Să urmărim şirul de transformări.

***Observaţie*:**



Calculul se face astfel:

$\min\ (i-1,\ j-1)$ şi se compară pe rând cu poziția $(i-1,\ j)$ şi $(i,\ j-1)$.

Transformarea primelor i caractere din s în primele j caractere din t	=	***Transformarea primelor*** i_1 ***caractere din s în primele*** j_1 ***caractere din t***	***Operația***
(6, 8)	=	(5,7)	$"-"$
(5,7)	=	(4,6)	$"-"$
(4,6)	=	1 + (4,5)	$"a" \leftarrow lit\ "i" = t_6$
(4,5)	=	1 + (4,4)	$"a" \leftarrow lit\ "i" = t_5$
(4,4)	=	1 + (4,3)	$"a" \leftarrow lit\ "i" = t_4$
(4,3)	=	(3,2)	$"-"$
(3,2)	=	(2,1)	$"-"$
(2,1)	=	1 + (1,1)	$"s" \leftarrow lit\ "o" = s_2$
(1,1)	=	(0,0)	$" "$

Deci Corina → Crina → Crisna → Cristna → Cristina

1.4.2. Procedeul înainte

Problemă: Fie şirul $A = (a_{i1}, a_{i2}, ..., a_{ik})$. Să se determine cel mai lung subşir crescător al lui $A\ (a_{i_1}, a_{i_2}, ..., a_{i_k})$ unde: $a_{i_1} \le a_{i_2} \le .... \le a_{i_k}$, $i_1 < i_2 < .... < i_k$.

Rezolvare: Pentru a respecta principiul programării dinamice facem următoarea presupunere: dacă $(a_{i_1}, ..., a_{i_k})$ este subşir crescător de lungime maximă, atunci $\forall\, j \in \{1, 2, ..., k-1\}$ subşirul $(a_{i_j}, a_{i_2}, ..., a_{i_k})$ este subşir crescător de lungime maximă care începe în a_{i_j}.

Notăm cu l_i, numărul elementelor celui mai lung subşir crescător care începe cu a_i. Atunci:

$$\begin{cases} l_n = 1 \\ l_i = 1 + \max\{l_k / a_i < a_k,\ i+1 \le k \le n\},\ i = \overline{n-1, 1} \end{cases}$$

Prin convenție, $\max\{\Phi\} = 0$.

Dacă vom determina $nr = \max\{l_i / 1 \le i \le n\}$, atunci problema este rezolvată pentru că $l_{i0} = nr$ este lungimea maximă a subşirului crescător iar acesta începe cu a_{i_0}. Determinarea efectivă a elementelor şirului se face începând cu a_{i_0} şi considerând toate elementele din A care respectă ordinea crescătoare.

```
procedure Secvmax (n, a[1.....n], nr, i0)
    var i, k, max` : integer;
    l[1...n] de tip integer
    ln = 1, nr = 0
    for i = n-1 downto 1 do
       max = 0
      for k = i + 1 to n do
```

```
            if (a_i < a_k) and (l_k > max) then max = l_k
            end_if
        next k
        l_i = l + max
        if l_i > nr then
            nr = l_i, i_0 = i
        end_if
    next i
end_proc.
```

1.4.3. Procedeul combinat

Problemă: Să se calculeze produsul $A_1 \times A_2 \times .... \times A_n$, unde $\forall\ i = \overline{1, n}$. A_i este o matrice de ordin $d_i \times d_{i+1}$, astfel încât să se efectueze un număr minim de înmulțiri.

Rezolvare: În general, pentru două matrici B_{pxq} şi C_{qxr}, numărul de înmulțiri pentru a calcula produsul $B \times C$ este $p.q.r.$. Produsul matricilor nu este comutativ dar este asociativ. În cazul celor n matrici există $(n-1)!$ posibilități de a le asocia. Dintre toate aceste asocieri trebuie aleasă cea în care numărul total de înmulțiri să fie minim. Iată un exemplu:

$A_1(100, 1) \times A_2(1, 100) \times A_3(100, 1) \times A_4(1, 100)$

În asocierea $(A_1 \times A_2) \times (A_3 \times A_4)$ se execută 1.020.000 înmulțiri, iar în asocierea $(A_1 \times (A_2 \times A_3)) \times A_4$, care este optimă, se execută 10.200 înmulțiri.

Vom utiliza metoda programării dinamice pentru a găsi asocierea optimă dintre cele $(n-1)!$ posibile.

Pentru $1 \le i \le j \le n$ introducem valoarea $cost(i, j)$ = numărul minim de înmulțiri necesar pentru a calcula produsul $A_i \times A_{i+1} \times .... \times A_j$.

Vom considera $cost(i,j) = 0$. Observăm că $cost(i,i+1) = d_i, d_{i+1}, d_{i+2}$, $1 \le i \le n-1$.

În general, $cost(i,j) = \min\{cost(i,k) + cost(k+1,j) + d_i, d_{k+1}, d_{j+1}\}$, unde minimul este luat pentru $i \le k \le j$.

Această ultimă formulă poate fi înțeleasă astfel:

$$A_{ij} = A_i \times A_{i+1} \times ... \times A_j = \overbrace{(A_i \times A_{i+1} ... \times A_k)}^{A_{ik}} \times \overbrace{(A_{k+1} \times ... A_j)}^{A_{k+1,j}}$$

unde k este valoarea pentru care se realizează minimul de mai sus. Primele două valori din sumă corespund factorilor $(A_i \times .... \times A_k)$ respectiv $(A_{k+1} \times .... \times A_j)$, iar $d_i \cdot d_{k+1} \cdot d_{j+1}$ este numărul de înmulțiri pentru produsul $A_{ik}(d_i, d_{k+1}) \times A_{k+1,j}(d_{k+1}, d_{j+1})$.

Formula costului permite calculul valorilor $cost(i,j)$ în ordinea descrescătoare a diferenței $j-1$. Pentru calculul tuturor valorilor $cost(i,j)$ folosim procedura:

procedure Prodmat ($d[1....n+1]$; $cost[1....n, 1....n]$)

for $i = 1$ to n **do**

$cost(i,i) = 0$

for $k = 1$ to $n-1$ **do**

for $i = 1$ to $n-k$ **do**

$j = i + k$

$cost(i,j) = \min\{cost(i,l) + cost(l+1,j) + d_i, d_{l+1}, d_{j+1}\}$, $1 \le l \le j$

$cost(j,i) = 1$, unde *l* este valoarea pentru care se realizeaza minimul

next *i*

next *k*

end_proc

Observăm că toate valorile $cost(i,j) = 0$ sunt scrise deasupra diagonalei principale, iar $cost(i,n) = 0$ reprezintă costul minim

cerut de problemă, adică numărul minim de înmulțiri în cazul asocierii optime.

Asocierea optimă o vom găsi datorită valorilor $\cos t\ (i, j)$ înscrise sub diagonala principală. Astfel, pentru $i < j$, fie $k = \cos t\ (j, i)$.

Atunci ştim sigur că $A_i \times .... \times A_j = (A_i \times .... \times A_k) \times (A_{k+1} \times .... \times A_j)$. Vom continua cu determinarea asocierilor din fiecare paranteză cercetând $\cos t\ (i, k)$ şi $\cos t\ (k+1, j)$. Ne vom opri când $\cos t\ (i, j) = 0$, acest lucru desemnând un singur factor, matricea $A_i\ (i, j)$. Formula completă a asocierilor o vom determina începând cu $\cos t(n, 1)$.

Iată o procedură care realizează acestă asociere.

```
procedure Asociere (i, j), i ≤ j
    k = cost (j, i)
    if cost (i, k) ≠ 0 then
      print (' (');
      Asociere (i, k);
      print (') ')
    else print ('A', i)
    end_if
    print ('x')
    if cost (k+1, j) ≠ 0 then
      print (' (');, asociere (k+1, j), print (') ')
    else print ('A', j)
    end_if
end_proc
Apel : Asociere (1, n)
```