CURS 7
TEHNICA DE PROGRAMARE "GREEDY"

1. Prezentare generală

Tehnica de programare Greedy este utilizată, de obicei, pentru rezolvarea problemelor de optimizare, adică a acelor probleme în care se cere determinarea unei submulțimi a unei mulțimi date pentru care se minimizează sau se maximizează valoarea unei funcții obiectiv. Formal, o problemă de optimizare poate fi enunțată astfel: "*Fie A o mulțime nevidă și $f: \mathcal{P}(A) \to \mathbb{R}$ o funcție obiectiv asociată mulțimii A, unde prin $\mathcal{P}(A)$ am notat mulțimea tuturor submulțimilor mulțimii A. Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie minimă/maximă pe S (i.e., pentru orice altă submulțime $T \subseteq A, T \neq S$, valoarea funcției obiectiv f va fi cel puțin /cel mult egală cu valoarea funcției obiectiv f pe submulțimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulțime nevidă de numere întregi. Să se determine o submulțime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcția obiectiv nu este dată în formă matematică și nu se precizează explicit faptul că suma elementelor submulțimii S trebuie să fie maximă *în raport cu suma oricărei alte submulțimi*, acest lucru subînțelegându-se. Formal, problema poate fi enunțată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ și $f: \mathcal{P}(A) \to \mathbb{R}$, $f(S) = \sum_{x \in S} x$. Să se determine o submulțime $S \subseteq A$ astfel încât valoarea funcției f să fie maximă pe S, i.e. $\forall T \subseteq A, T \neq S \Rightarrow f(T) \leq f(S)$ sau, echivalent, $\forall T \subseteq A, T \neq S \Rightarrow \sum_{x \in T} x \leq \sum_{x \in S} x$.*"

Evident, orice problemă de acest tip poate fi rezolvată prin *metoda forței-brute*, astfel: se generează, pe rând, toate submulțimile S ale mulțimii A și pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obținută este mai mică/mai mare decât minimul/maximul obținut până în acel moment, atunci se actualizează minimul/maximul și se reține submulțimea S. Deși aceasta rezolvare este corectă, ea are o complexitate exponențială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulțimea S, la fiecare pas, cel mai bun element disponibil din mulțimea A din punct de vedere al optimizării funcției obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcției obiectiv combinând optimele sale locale. Totuși, prin combinarea unor optime locale nu se obține întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre București și Arad (un optim local), precum și cel mai scurt drum posibil dintre Arad și Ploiești (alt optim local). Combinând cele două optime locale nu vom obține un optim global, deoarece, evident, cel mai scurt drum de la București la Ploiești nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însoțită de o demonstrație a corectitudinii (optimalității) criteriului de selecție pe care trebuie să-l îndeplinească un element al mulțimii A pentru a fi adăugat în soluția S.

Revenind la problema determinării unei submulțimi S cu sumă maximă, observăm faptul că aceasta trebuie să conțină toate elementele pozitive din mulțimea A, deci criteriul de selecție este ca elementul curent din A să fie pozitiv (demonstrația optimalității este banală). Dacă mulțimea A nu conține niciun număr pozitiv, care va fi soluția problemei?

În anumite probleme, criteriul de selecție poate fi aplicat mai eficient dacă se realizează o prelucrare inițială a elementelor mulțimii A – de obicei, o sortare a lor. De exemplu, să considerăm următoarea problemă: "*Fie A o mulțime nevidă formată din n numere întregi. Să se determine o submulțime $S \subseteq A$ având exact k elemente $(k \leq n)$ cu proprietatea că suma elementelor sale este maximă.*". Evident, submulțimea S trebuie să conțină cele mai mari k elemente ale mulțimii A, iar acestea pot fi selectate în două moduri:

- de k ori se selectează maximul din mulțimea A și se elimină (sau doar se marchează – important este ca, la fiecare pas, să nu mai luăm în considerare maximul determinat anterior), deci această soluție va avea complexitatea $\mathcal{O}(kn)$, care oscilează între $\mathcal{O}(n)$ pentru valori ale lui k mult mai mici decât n și $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n;
- sortăm crescător elementele mulțimii A și apoi selectăm ultimele k elemente, deci această soluție va avea complexitatea $\mathcal{O}(k + n\log_2 n) \approx \mathcal{O}(n\log_2 n)$, care nu depinde de valoarea k.

În plus, a doua varianta are avantajul unei implementări puțin mai simple decât prima.

În concluzie, pentru o mulțime A cu n elemente, putem considera următoarea formă generală a unui algoritm de tip Greedy:

```
prelucrarea inițială a mulțimii A (o sortare a sa, de obicei)
j = 0;
for(i = 0; i < n; i++)
   if(A[i] verifică criteriul de selecție)
   {
         S[j] = A[i];
         j++;
   }
afișarea elementelor mulțimii S
```

Complexitățile specifice ale unui algoritm de tip Greey sunt mici, cele mai des întâlnite fiind următoarele:

- $\mathcal{O}(n)$, dacă se prelucrează mulțimea A cu un algoritm având complexitatea maximă $\mathcal{O}(n)$, iar verificarea criteriului de selecție pentru fiecare dintre cele n elemente ale mulțimii A se realizează cu complexitatea $\mathcal{O}(1)$;
- $\mathcal{O}(n\log_2 n)$, dacă se sortează elementele mulțimii A folosind un algoritm cu complexitatea $\mathcal{O}(n\log_2 n)$, iar verificarea criteriului de selecție pentru fiecare dintre cele n elemente ale mulțimii A se realizează cu complexitatea $\mathcal{O}(1)$;
- $\mathcal{O}(n^2)$, dacă se prelucrează mulțimea A cu un algoritm având complexitatea maximă $\mathcal{O}(n^2)$ sau verificarea criteriului de selecție pentru fiecare dintre cele n elemente ale mulțimii A se realizează cu complexitatea $\mathcal{O}(n)$.

Evident, acestea nu sunt toate complexitățile posibile pentru un algoritm de tip Greedy, ci doar sunt cele mai des întâlnite!

Deoarece selecția unui element din mulțimea A se realizează într-o manieră statică, respectiv fără a ține cont de elementele deja selectate sau de elementele care ar mai putea fi selectate ulterior, este obligatorie demonstrarea corectitudinii criteriului de selecție. În caz contrar, respectivul algoritm Greedy s-ar putea să furnizeze soluții incorecte! Practic, demonstrația corectitudinii unui algoritm de tip Greedy trebuie să ne garanteze faptul că prin selectarea și combinarea unor optime locale vom obține optimul global!

De exemplu, să considerăm *problema plății unei sume folosind un număr minim de monede*: "Fie n tipuri de monede cu valorile $v_1, v_2, \ldots, v_n$, din fiecare tip de monedă fiind disponibil un număr nelimitat de monede. Fiind dată o sumă de bani S, să se determine o modalitate de plată a sa folosind un număr minim de monede." O rezolvare de tip Greedy ar putea fi următoarea: *la fiecare pas utilizăm un număr maxim de monede din moneda cu valoare maximă care nu a fost utilizată anterior*. Astfel, pentru monede cu valorile 8, 7 și 5 RON, putem plăti suma $S = 23$ RON în următorul mod: 2×8 RON $+ 1 \times 7$ RON. Se observă ușor faptul că soluția obținută, folosind 3 monede, este optimă (nu există nicio posibilitate de plată a sumei $S = 23$ RON folosind doar două monede). Dacă vom considera acum suma $S = 14$ RON, algoritmul va folosi mai întâi o monedă de 8 RON, apoi una de 5 RON (nu poate folosi moneda de 7 RON!) și va rămâne cu un rest de 1 RON pe care nu îl poate plăti, deci algoritmul va considera faptul că problema nu are soluție. Evident, problema are soluția optimă 2×7 RON, deci algoritmul de tip Greedy propus este incorect! De asemenea, putem observa faptul că existența monedei cu valoarea de 1 RON garantează găsirea unei variante de plată a sumei S, dar care nu este neapărat optimă. De exemplu, pentru monede cu valorile 8, 7 și 1 RON, suma $S = 14$ RON va fi plătită folosind 1×8 RON $+ 6 \times 1$ RON, dar, evident, această modalitate nu este optimă! Totuși, pentru anumite valori ale monedelor, algoritmul Greedy prezentat mai sus, denumit *algoritmul casierului*, va furniza întotdeauna soluția optimă (https://tylermoore.ens.utulsa.edu/courses/cse3353/slides/l13-handout.pdf, respectiv https://arxiv.org/pdf/0809.0400.pdf).

2. Minimizarea timpului mediu de așteptare

La un ghișeu, stau la coadă n persoane $p_1, p_2, \ldots, p_n$ și pentru fiecare persoană p_i se cunoaște timpul său de servire t_i. Să se determine o modalitate de reașezare a celor n persoane la coadă, astfel încât timpul mediu de așteptare să fie minim.

De exemplu, să considerăm faptul că la ghișeu stau la coadă $n = 6$ persoane, având timpii de servire $t_1 = 7,\ t_2 = 6,\ t_3 = 5,\ t_4 = 10,\ t_5 = 6$ și $t_6 = 4$. Evident, pentru ca o persoană să fie servită, aceasta trebuie să aștepte ca toate persoanele aflate înaintea sa la coadă să fie servite, deci timpii de așteptare ai celor 6 persoane vor fi următorii:

Persoana	**Timpul de servire (t_i)**	**Timp de așteptare (a_i)**
p_1	7	7
p_2	6	7 + 6 = 13
p_3	3	13 + 3 = 16
p_4	10	16 + 10 = 26
p_5	6	26 + 6 = 32
p_6	3	32 + 3 = 35
Timpul mediu de așteptare (M):		$\frac{7+13+16+26+32+35}{6} = \frac{129}{6} = 21.5$

Deoarece timpul de servire al unei persoane influențează timpii de așteptare ai tuturor persoanelor aflate după ea la coadă, se poate intui foarte ușor faptul că minimizarea timpului mediu de așteptare se obține rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	**Timpul de servire (t_i)**	**Timp de așteptare (a_i)**
p_3	3	3
p_6	3	3 + 3 = 6
p_2	6	6 + 6 = 12
p_5	6	12 + 6 = 18
p_1	7	18 + 7 = 25
p_4	10	25 + 10 = 35
Timpul mediu de așteptare (M):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

Practic, minimizarea timpului mediu de așteptare este echivalentă cu minimizarea timpului de așteptare al fiecărei persoane aflate la coadă, iar minimizarea timpului de așteptare al unei persoane se obține minimizând timpii de servire ai persoanelor aflate înaintea sa la coadă!

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \ldots, p_i, \ldots, p_j, \ldots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \leq t_2 \leq \cdots \leq t_i \leq \cdots \leq t_j \leq \cdots \leq t_n$. De asemenea, vom presupune faptul că timpii individuali de servire $t_1, t_2, \ldots, t_n$ nu sunt toți egali între ei (în acest caz, problema ar fi trivială), deci există $i < j$ astfel încât $t_i < t_j$. În continuare, presupunem faptul că această modalitate P_1 de aranjare a persoanelor la coadă (o permutare, de fapt) nu este optimă, deci există o altă modalitate optimă P_2 de aranjare $p_1, p_2, \ldots, p_j, \ldots, p_i, \ldots p_n$ diferită de cea inițială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i și p_j din varianta inițială, adică persoana p_j se află acum pe poziția i în coadă, iar persoana p_i se află acum pe poziția j, unde $i < j$).

În cazul primei modalități de aranjare P_1, timpul mediu de așteptare M_1 este egal cu:

$$M_1 = \frac{t_1 + (t_1 + t_2) + \cdots + (t_1 + \cdots + t_i) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_i + \cdots + (n-j+1)t_j + \cdots + 2t_{n-1} + t_n}{n}$$

În cazul celei de-a doua modalități de aranjare P_2, timpul mediu de așteptare M_2 este egal cu:

$$M_2 = \frac{t_1 + (t_1 + t_2) + \cdots + \left(t_1 + \cdots + t_j\right) + \cdots + (t_1 + \cdots + t_i) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_j + \cdots + (n-j+1)t_i + \cdots + 2t_{n-1} + t_n}{n}$$

Comparăm acum M_1 cu M_2, calculând diferența dintre ele:

$$M_1 - M_2 = \frac{(n-i+1)t_i + (n-j+1)t_j - (n-i+1)t_j - (n-j+1)t_i}{n} =$$
$$= \frac{t_i(n-i+1-n+j-1) + t_j(n-j+1-n+i-1)}{n} =$$
$$= \frac{t_i(-i+j) + t_j(-j+i)}{n} = \frac{-t_i(i-j) + t_j(i-j)}{n} = \frac{(t_j - t_i)(i-j)}{n}$$

Deoarece $i < j$ și $t_j > t_i$, obținem faptul că $M_1 - M_2 = \frac{(t_j - t_i)(i-j)}{n} < 0$ (evident, $n \geq 1$), ceea ce implică $M_1 < M_2$. Acest fapt contrazice optimalitatea modalității de aranjare P_2, deci presupunerea că modalitatea de aranjare P_1 (în ordinea crescătoare a timpilor de servire) nu ar fi optimă este falsă!

Atenție, soluția acestei probleme constă într-o rearanjare a persoanelor $p_1, p_2, \dots, p_n$, deci în implementarea acestui algoritm nu este suficient să sortăm crescător timpii de servire, ci trebuie să memorăm perechi de forma (p_i, t_i), folosind, de exemplu, o structură, iar apoi să le sortăm crescător după componenta t_i. Evident, complexitatea algoritmului este dată doar de complexitatea operației de sortare utilizate, deci complexitatea sa optimă este $\mathcal{O}(n \log_2 n)$.

O implementarea a acestui algoritm în limbajul C este următoarea (numărul n de persoane și timpii lor de servire se vor citi din fișierului text `persoane.in`, iar rezultatele vor fi scrise în fișierul text `persoane.out`:

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

typedef struct
{
    //ID = numarul de ordine initial al persoanei la coada
    //ts = timpul de servire individual
    int ID, ts;
} Persoana;

//functie comparator bazata pe timpii de servire a doua persoane
int cmpPersoane(const void *a, const void *b)
{
    Persoana pa = *(Persoana*)a;
    Persoana pb = *(Persoana*)b;

    if(pa.ts < pb.ts) return -1;
    if(pa.ts > pb.ts) return  1;
    return 0;
}

int main()
{
    int n;
    Persoana *p;
```

```
//citirea datelor de intrare din fisierul de intrare
FILE* fin = fopen("persoane.in", "r");

fscanf(fin, "%d", &n);
p = (Persoana*)malloc(n * sizeof(Persoana));
for(int i = 0; i < n; i++)
{
    p[i].ID = i+1;
    fscanf(fin, "%d", &p[i].ts);
}

fclose(fin);

//scrierea timpilor initiali in fisierul de iesire
FILE* fout = fopen("persoane.out", "w");
fprintf(fout, "Timpii initiali:\n");
fprintf(fout, "Persoana\tTimp de servire\t\t
                                        Timp de asteptare\n");

//ta = timpul de asteptare al persoanei curente
int ta = 0;
//tma = timpul mediu de asteptare al celor n persoane
float tma = 0;
for(int i = 0; i < n; i++)
{
    ta = ta + p[i].ts;
    tma = tma + ta;
    fprintf(fout, "\t%d\t\t\t\t%d\t\t\t\t\t%d\n",
                                        p[i].ID, p[i].ts, ta);
}
tma = tma / n;
fprintf(fout, "Timpul mediu de asteptare initial: %.2f\n", tma);

//sortam persoanele in ordinea crescatoare a timpilor de servire
qsort(p, n, sizeof(Persoana), cmpPersoane);

//scriem in fisierul de iesire timpii optimi
fprintf(fout, "\n\nTimpii optimi:\n");
fprintf(fout,"Persoana\tTimp de servire\t\tTimp de asteptare\n");

ta = 0;
tma = 0;
for(int i = 0; i < n; i++)
{
    ta = ta + p[i].ts;
    tma = tma + ta;
    fprintf(fout, "\t%d\t\t\t\t%d\t\t\t\t\t%d\n",
                                        p[i].ID, p[i].ts, ta);
}
```

```
    tma = tma / n;
    fprintf(fout, "Timpul mediu de asteptare minim: %.2f\n", tma);

    fclose(fout);

    free(p);

    return 0;
}
```

Încheiem prezentarea acestei probleme precizând faptul că este o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră* n *activități cu duratele* $t_1, t_2, \dots, t_n$ *care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă* (respectiv, *la un moment dat resursa comună poate fi alocată unei singure activități*)*, să se determine o modalitate de planificare a activităților astfel încât timpul mediu de așteptare să fie minim.*".