# Metoda Greedy

”Lacomul” care ne poate duce, uneori, la soluția bună

# Descriere

- **Metoda Greedy** propune o strategie de rezolvare a problemelor de optim în care se poate obţine optimul global prin alegeri succesive ale optimului local, ceea ce permite rezolvarea problemei fără nici o revenire la deciziile luate deja.

- În general, această metodă se aplică **problemelor de optimizare**.

- Majoritatea acestor probleme constau în determinarea unei submulţimi **B**, a cărei elemente aparțin unei mulţimi **A**, care să îndeplinească anumite condiţii pentru a fi acceptată.

- Orice astfel de submulţime care respectă aceste restricţii se numeşte ***soluţie posibilă***.

# Descriere

- Din mulţimea tuturor soluţiilor posibile se doreşte determinarea unei soluţii care maximizează sau minimizează o funcţie de cost.

- O soluţie posibilă care realizează acest lucru se numeşte ***soluţie optimă***.

- Se construieşte **soluţia optimă pas cu pas**, la fiecare pas fiind selectat (sau "înghiţit") în soluţie elementul care pare "cel mai bun" la momentul respectiv, în speranţa că va duce la soluţia optimă globală.

- Considerăm că soluţiile posibile au următoarea proprietate: dacă **B** este o soluţie posibilă, atunci orice submulţime a sa este soluţie posibilă.

# Funcționare

1. Se inițializează mulțimea soluțiilor **B** cu mulțimea vidă;
2. Se alege un element **x϶A**, *cel mai promițător element la momentul respectiv*, care poate conduce la o soluție optimă;
3. Se verifică dacă elementul poate fi adăugat la mulțimea **B**, dacă da **B=B U {x}**. Dacă un element este adăugat atunci nu mai poate fi eliminat. Dacă un element e refuzat atunci nu va mai fi testat/prelucrat ulterior;
4. Dacă nu s-a terminat de prelucrat mulțimea **A** și/sau nu s-a ajuns la soluție se reia de la pasul 2;

OBS: În rezolvarea problemelor, de multe ori este utilă ordonarea mulţimii **A** înainte ca algoritmul propriu-zis să fie aplicat.

# Descriere algoritm

```
B = multimea vidă
for (i=1; i<=n; i++)    //parcurgem elementele mulțimii A
  {
    x = alege(A);       // Alegem un element
    if (posibil(B, x))  // Dacă elementul poate face parte din B
      adaug(B, x);      // Îl adăugăm la mulțimea B
  }
```

# Descriere algoritm 2

```
B = multimea vidă
prelucreaza(A, v)// prelucrăm mulțimea A și obținem vectorul v
for (i=1; i<=n; i++)    // parcurgem elementele vectorului v
  {
    x = v[i];          // Luăm un element din vectorul v
    if (posibil(B, x))    // Dacă elementul poate fi ales
     adaug(B, x);      // Îl adăugăm la mulțimea B

}
```

# Observații

- Metoda Greedy nu caută să determine toate soluţiile posibile şi să aleagă pe cea optimă conform criteriului de optimizare dat, ci constă în a alege pe rând câte un element urmând să-l introducă eventual în soluţia optimă.
- Acest lucru duce la eficienţa algorimilor Greedy, însă nu conduce în mod necesar la o soluţie optimă şi nici nu este posibilă formularea unui criteriu general conform căruia să putem stabili exact dacă metoda **Greedy** rezolvă sau nu o anumită problemă de optimizare.
- Acest motiv duce la completarea fiecărei rezolvări prin metoda **Greedy** cu o demonstraţie matematică.

☺

# Analiza unei probleme

- O stație de benzinărie trebuie să satisfacă cererile a **n** clienți. Pentru fiecare client *i* se cunoaște timpul de servire $t_i$. Scrieți un program care minimizează timpul de așteptare.

- Exemplu: pentru **n**=3, $t_1$=5, $t_2$=7, $t_3$=2. Dacă clienții sunt serviți în ordinea în care vin timpul de așteptare este 5 pentru primul client, 12 (5+7) pentru al doilea client, 14 (5+7+2) pentru ultimul client, timpul total de așteptare fiind 31 (5+12+14).

- OBS: Cei **n** clienți pot fi serviți în **n!** moduri. Să calculăm toate modurile posibile și apoi să alegem ar fi nepractic:
  - pentru **n**=10 numărul de combinații ar fi 3628800
  - pentru **n**=20 numărul de combinații ar fi 2432902008176640000

# Analiza unei probleme 2

- Cei 3 clienți pot fi serviți în ordinea:

| | | |
|---|---|---|
| 1 2 3 | T=5+(5+7)+(5+7+2)=31 | |
| 1 3 2 | T=5+(5+2)+(5+2+7)=26 | |
| 2 1 3 | T=7+(7+5)+(7+5+2)=33 | |
| 2 3 1 | T=7+(7+2)+(7+2+5)=30 | |
| 3 1 2 | T=2+(2+5)+(2+5+7)=23 | minim |
| 3 2 1 | T=2+(2+7)+(2+7+5)=25 | |

Se observă că minimul se obține când timpii sunt în ordine crescătoare.

# Analiza unei probleme 3

- Demonstrație:

Fie $I=\{i_1, i_2, \ldots, i_n\}$ o permutare a mulțimii {1, 2,…, n}

Dacă servirea are loc în ordinea *I* atunci timpul de așteptare este $T=t_{i1}+(t_{i1}+t_{i2})+(t_{i1}+t_{i2}+t_{i3})+\ldots=$

$=n*t_{i1}+(n-1)*t_{i2}+(n-2)*t_{i3}+\ldots=\sum_{k=1}^{n}(n-k+1)*t_{ik}$

Dacă găsim în permutarea *I* doi termeni a,b a<b pentru care $t_{ia}>t_{ib}$ interschimbăm pe $i_a$ cu $i_b$, adică schimbăm clientul care a fost servit al *a*-lea cu clientul care a fost servit al *b*-lea atunci obținem o nouă permutare *J*.

# Analiza unei probleme 4

$T(J)=(n-a+1)*t_{ib}+(n-b+1)*t_{ia}+\sum_{\substack{k=1\\k\neq a,b}}^{n}(n-k+1)*t_{ik}$

$T(I)-T(J)=(n-a+1)*t_{ia}+(n-b+1)*t_{ib}-(n-a+1)*t_{ib}-(n-b+1)*t_{ia}=(b-a)*(t_{ib}-t_{ia})>0$

Deci prin interchimbare s-a îmbunătățit timpul (a scăzut).

Nu se mai pot face îmbunătățiri dacă pentru oricare a,b cu $a<b$ $t_{ia}<t_{ib}$, adică timpii sunt în ordine crescătoare.

# Problema 1

- Se dă o mulțime **A** cu **n** valori reale. Determinați o submulțime **B** astfel încât suma elementelor să fie maximă.

- Idee: se aleg toate elementele strict pozitive, dacă nu există nici un element pozitiv se alege maximul dintre ele.

- Temă: de implementat algoritmul.

# Problema 2

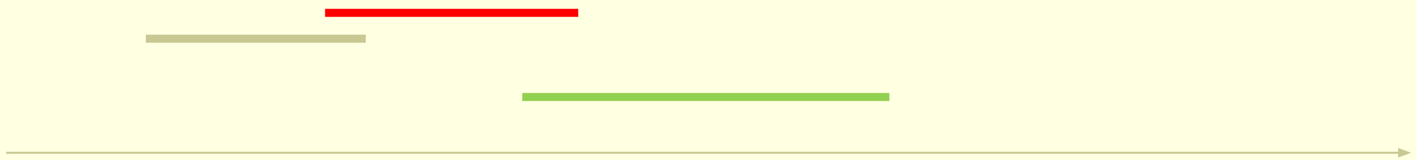
- Se dă o mulțime **A** cu **n** valori reale. Determinați o submulțime **B**, cu **m** elemente, astfel încât suma elementelor să fie maximă.

- Idee: Se ordonează descrescător elementele mulțimii **A**, iar apoi se iau primele **m** elemente.

- Temă: de implementat algoritmul

# Problema 3

- Fie un set **S** de **n** segmente pe axa **OX**. Pentru fiecare segment $\mathbf{s}_i$ este cunoscută abscisa extremităţii stângi $\mathbf{x}_i$ şi lungimea sa $\mathbf{l}_i$.
  Să se scrie un program, care va determina un subset cu un număr maxim de segmente, $\mathbf{B} \subseteq \mathbf{S}$, astfel încât segmentele din **B** nu se vor intersecta între ele.



- Idee:
  - un segment poate fi adăugat doar după ce se sfârșește precedentul
  - Sortezi după capătul din dreapta.

- Temă: de implementat algoritmul

# Problema 4 – Problema rucsacului v1.0

- Un hoț intră într-o încăpere unde se află **n** obiecte de aur de greutate și volum cunoscute. Știind că are un rucsac de volum **V** cunoscut și că poate tăia obiectele, dacă e necesar, să se determine care este greutatea maximă a obiectelor ce se pot încărca în rucsac.

- Idee: Se pun obiectele în rucsac în ordine descrescătoare a densității până când rucsacul se umple. E posibil ca din ultimul obiect să trebuiască tăiată o bucată.

# Problema 5 – Problema rucsacului v2.0

- Un hoț intră într-o încăpere unde se află **n** obiecte de aur de greutate și volum cunoscute. Știind că are un rucsac de volum **V** cunoscut să se determine care este greutatea maximă a obiectelor ce se pot încărca în rucsac.

- OBS. Problema nu se poate rezolva folosind metoda (anterior prezentată) Greedy. Ex.pentru datele (greutate, volum) : (9,6), (6,5), (5,5) și volumul rucsacului V=10 metoda anterioară (fără tăierea obiectelor) va alege obiectul de greutate 9 si volum 6 (9,6), iar soluția corectă ar fi fost (6,5)+(5,5).
- Problema se poate rezolva folosind programare dinamică