CURS 8
Complexitatea algoritmilor

Complexitatea unui algoritm:
1. timp de executare (complexitate computațională)
2. memoria utilizată

Se presupune faptul că algoritmii comparați au aceeași dimensiune a datelor de intrare!!!

Complexitate computațională = o estimare a numărului de operații elementare efectuate de către algoritm în funcție de dimensiunile datelor de intrare

Notație (Big O): $\mathcal{O}$(numărul maxim de operații elementare estimat)

Exemplu: $\mathcal{O}(n^2)$ => dimensiunea datelor de intrare este n (variabila din expresie), iar algoritmul efectuează aproximativ n^2 operații elementare (expresia)

Operațiile elementare pe care le efectuează un algoritm sunt:
1. operația de atribuire și operațiile aritmetice
2. operația de decizie și operația de salt
3. operații de citire/scriere

Estimarea complexității unui algoritm

Exemplu 1: Determinarea maximului dintr-o listă

Instrucțiune	Operații elementare	
`n = int(input("Numar elemente: "))`	1 afișare + 1 citire	
`lista = []`	1 atribuire	
`for i in range(n):`	de n ori:	3n operații elementare
`    elem = int(input("Element:"))`	1 afișare + 1 citire	
`    lista.append(elem)`	1 atribuire	
`maxim = lista[0]`	1 atribuire	
`for i in range(1, n):`	de n-1 ori:	n-1 sau 2(n-1) operații elementare
`    if lista[i] > maxim:`	1 operație de decizie	
`        maxim = lista[i]`	1 atribuire?	
`print("Maximul:", maxim)`	1 afișare	
TOTAL:	5n-2+5 operații elementare	

TOTAL = 5n+3 => complexitatea $\mathcal{O}(5n+3) \approx \mathcal{O}(n)$

Reguli de reducere a expresiilor din complexitatea unui algoritm:

1. constantele (multiplicative sau aditive) nu contează
   $\mathcal{O}(5n+3) \approx \mathcal{O}(5n) \approx \mathcal{O}(n)$

2. dintr-o expresie se păstrează doar termenul dominant ($n \to \infty$)
   $\mathcal{O}(3n^2+5n+7) \approx \mathcal{O}(3n^2) \approx \mathcal{O}(n^2)$
   $\mathcal{O}(2^n+3n^2) \approx \mathcal{O}(2^n)$

Exemplu 2: Sortarea prin selecție

<table>
<tr><th>Instrucțiune</th><th colspan="2">Operații elementare</th></tr>
<tr><td>n = int(input("Numar elemente: "))</td><td colspan="2">1 afișare + 1 citire</td></tr>
<tr><td>lst = []</td><td colspan="2">1 atribuire</td></tr>
<tr><td>for i in range(n):</td><td>de n ori:</td><td rowspan="3">3n operații elementare</td></tr>
<tr><td>   elem = int(input("Element: "))</td><td>1 afișare + 1 citire</td></tr>
<tr><td>   lst.append(elem)</td><td>1 atribuire</td></tr>
<tr><td>for i in range(n-1):</td><td colspan="2" rowspan="4">$\frac{n(n-1)}{2}$ operații de decizie

sau

$n(n-1)$ operații de decizie + atribuire</td></tr>
<tr><td>   for j in range(i+1, n):</td></tr>
<tr><td>   if lst[i] > lst[j]:</td></tr>
<tr><td>   lst[i], lst[j] = lst[j], lst[i]</td></tr>
<tr><td>print("Lista sortata:")</td><td colspan="2">1 afișare</td></tr>
<tr><td>for i in range(n):</td><td>de n ori:</td><td rowspan="2">n operații elementare</td></tr>
<tr><td>   print(lst[i], end=" ")</td><td>1 afișare</td></tr>
<tr><td>TOTAL:</td><td colspan="2">$n(n-1)+4n+4 = n^2+3n+4$</td></tr>
</table>

complexitatea $\mathcal{O}(n^2 + 3n + 4) \approx \mathcal{O}(n^2)$

Pentru i = 0 => se execută de n-1 ori operația de decizie
Pentru i = 1 => se execută de n-2 ori operația de decizie
.....
Pentru i = n-2 => se execută de 1 ori operația de decizie

$$\text{TOTAL} = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2}$$

for(i = 0; i < n; i++)
operație cu complexitatea $\mathcal{O}(1)$;
for(j = 0; j < m; j++)
operație cu complexitatea $\mathcal{O}(1)$;
	Complexitatea $\mathcal{O}(n+m)$

for(i = 0; i < n; i++) operație cu complexitatea $\mathcal{O}(m)$;	Complexitatea $\mathcal{O}(nm)$

for(i = 0; i < n; i++)
for(j = 0; j < m; j++)
operație cu complexitatea $\mathcal{O}(1)$;
	Complexitatea $\mathcal{O}(nm)$

for(i = 0; i < n; i++)
for(j = 0; j < m; j++)
operație cu complexitatea $\mathcal{O}(p)$;
	Complexitatea $\mathcal{O}(nmp)$

for(i = 0; i < n; i++)
{
for(i = 0; i < m; i++)
operație cu complexitatea $\mathcal{O}(1)$;
for(j = 0; j < p; j++)
operație cu complexitatea $\mathcal{O}(1)$;
}
	Complexitatea $\mathcal{O}(n(m+p))$

Complexitatea $\mathcal{O}(nm)$ poate fi considerată:

- $\mathcal{O}(n^2)$ dacă $m \approx n$
- $\mathcal{O}(n)$ dacă $m \ll n$

Exemplu: determinarea valorilor distincte dintr-o listă de numere

```
lista = [2, 1, 1, 7, 2, 3, 4, 1, 1, 1, 2, 4]

n = int(input("Numar elemente: "))
lst = []
for i in range(n):
    elem = int(input("Element: "))
    lst.append(elem)

distincte = []
for i in range(n):
    if lst[i] not in distincte: -> O(d = nr val distincte)
        distincte.append(lst[i])

print("Elementele distincte:")
for i in range(len(distincte)):
    print(distincte[i], end=" ")
```

Complexitatea este $\mathcal{O}(nd)$, unde d reprezintă numărul valorilor distincte din listă, deci poate fi:

- $\mathcal{O}(n^2)$ dacă $d \approx n$
- $\mathcal{O}(n)$ dacă $d \ll n$

Clase uzuale de complexitate computațională (în ordine crescătoare)

1. Clasa $\mathcal{O}(1)$ – *complexitate constantă*

Exemple: orice operație elementară, suma a două numere, formule simple (rezolvarea ecuației de gradul I sau II)

2. Clasa $\mathcal{O}(\log_b n)$ – *complexitate logaritmică*

Exemple: suma cifrelor unui număr - $\mathcal{O}(\log_{10} n)$, operația de căutare binară - $\mathcal{O}(\log_2 n)$

Presupunem că numărul n are x cifre => $10^{x-1} \leq n < 10^x$ => $\log_{10} 10^{x-1} \leq \log_{10} n < \log_{10} 10^x$ => $x - 1 \leq \log_{10} n < x$ => $[\log_{10} n] = x - 1$ => $x = [\log_{10} n] + 1$.

Operația de căutare binară: să se verifice dacă o valoare x apare sau nu într-un șir format din n numere sortate crescător.
Exemplu: $v = (2, 3, 3, 7, 10, 15, 15, 25, 100, 101)$ și $x = 3$

$$\log_2 8 = 3 \iff 2^3 = 8 \Leftrightarrow \frac{8}{2} = 4; \frac{4}{2} = 2; \frac{2}{2} = 1$$

Algoritmul de căutare binară:

- citirea tabloului sortat crescător și a valorii x căutate - $\mathcal{O}(n + 1)$
- operația de căutare binară - $\mathcal{O}(\log_2 n)$
- afișarea rezultatului - $\mathcal{O}(1)$
- COMPLEXITATEA ALGORITMULUI: $\mathcal{O}(n + \log_2 n) \approx \mathcal{O}(n)$

3. Clasa $\mathcal{O}(n)$ – *complexitate liniară*

Exemple: citirea/scrierea/o singură parcurgere a unui tablou unidimensional cu n elemente, suma primelor n numere naturale (fără formulă) etc.

4. Clasa $\mathcal{O}(n \log_2 n)$

Exemple: Quicksort (sortarea rapidă), Mergesort (sortarea prin interclasare), Heapsort (sortarea cu ansamble)

5. Clasa $\mathcal{O}(n^2)$ – *complexitate pătratică*

Exemple: sortarea prin interschimbare, Bubblesort, compararea fiecărui element al unui tablou unidimensional cu n elemente cu toate celelalte elemente din tablou, citirea/scrierea/o singură parcurgere a unui tablou bidimensional cu n linii și n coloane

6. Clasa $\mathcal{O}(n^k), k \geq 3$ – *complexitate polinomială*

Exemple: sortarea fiecărei linii dintr-o matrice pătratică de dimensiune n folosind sortarea prin interschimbare sau Bubblesort - $\mathcal{O}(n^3)$

7. Clasa $\mathcal{O}(a^n), a \geq 2$ – *complexitate exponențială*

Exemple: generarea tuturor submulțimilor unei mulțimi cu n elemente - $\mathcal{O}(2^n)$

Teoremă: O mulțime cu n elemente are 2^n submulțimi.

Exemplu:

$$A = \{1,2,3\} \Longrightarrow \mathcal{P}(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$
$$\Longrightarrow |\mathcal{P}(A)| = 8 = 2^3$$

$\mathcal{P}(A) =$ mulţimea părţilor lui $A =$ mulţimea tuturor submulţimilor lui A

Observație: Complexitatea unui algoritm NU poate fi mai mică decât complexitatea citirii datelor de intrare și/sau scrierii datelor de ieșire!!!

Exemplu 1:

```
n = int(input("n = "))
i = 0
p = 1
while i <= n:
    j = 1
    while j <= p:
        print(j, end= " ")
        j = j + 1
    print()
    i = i + 1
    p = p * 2
```

```
n = 3
1                       2^0
1 2                     2^1
1 2 3 4                 2^2
1 2 3 4 5 6 7 8         2^3
```

$$2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1$$

$\mathcal{O}(2^n)$

Exemplu 2:

```
a = int(input("a = "))
b = int(input("b = "))

p = 1
while p < a:
    p = p * 2
while p <= b:
    print(p, end=" ")
    p = p * 2
```

Afișează puterile lui 2 cuprinse între a și b

$$2^k \leq b => k \leq \log_2 b$$

$\mathcal{O}(\log_2 b)$

Exemplu 3:

```
v = [int(x) for x in input("Valorile: ").split()]
i = 0
j = len(v) - 1
while i < j:
    while i < len(v) and v[i] < 0:
        i = i + 1
    while j >= 0  and v[j] >= 0:
        j = j - 1
    if i < j:
        v[i], v[j] = v[j], v[i]

print("\nValorile:\n", v, sep="")
```

$\mathcal{O}(n)$

Sortare parțială: **numerele negative înaintea celor pozitive**

				j	i				
-10	-15	-21	-30	-1	8	19	20	10	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 submulţimi a unei mulţimi date pentru care se minimizează sau se maximizează valoarea unei funcţii obiectiv.

Forma generală a unei probleme de optimizare:

"*Fie A o mulţime nevidă şi $f: \mathcal{P}(A) \to \mathbb{R}$ o funcţie obiectiv asociată mulţimii A, unde prin $\mathcal{P}(A)$ am notat mulţimea tuturor submulţimilor mulţimii A. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie minimă/maximă pe S (i.e., pentru orice altă submulţime $T \subseteq A, T \neq S$, valoarea funcţiei obiectiv f va fi cel puţin /cel mult egală cu valoarea funcţiei obiectiv f pe submulţimea S).*"

Exemplu:
"*Fie A o mulţime nevidă de numere întregi. Să se determine o submulţime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*"

"*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ şi $f: \mathcal{P}(A) \to \mathbb{R}$,*

$$f(S) = \sum_{x \in S} x.$$

Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie maximă pe S, i.e. $\forall T \subseteq A, T \neq S \Rightarrow f(T) \leq f(S)$ sau, echivalent, $\forall T \subseteq A, T \neq S \Rightarrow \sum_{x \in T} x \leq \sum_{x \in S} x$."

A = {-5, 7, -1, 10, 3} => S = {7, 10, 3}

A = {-5, -7, -1, -10, -3} => S = {-1}

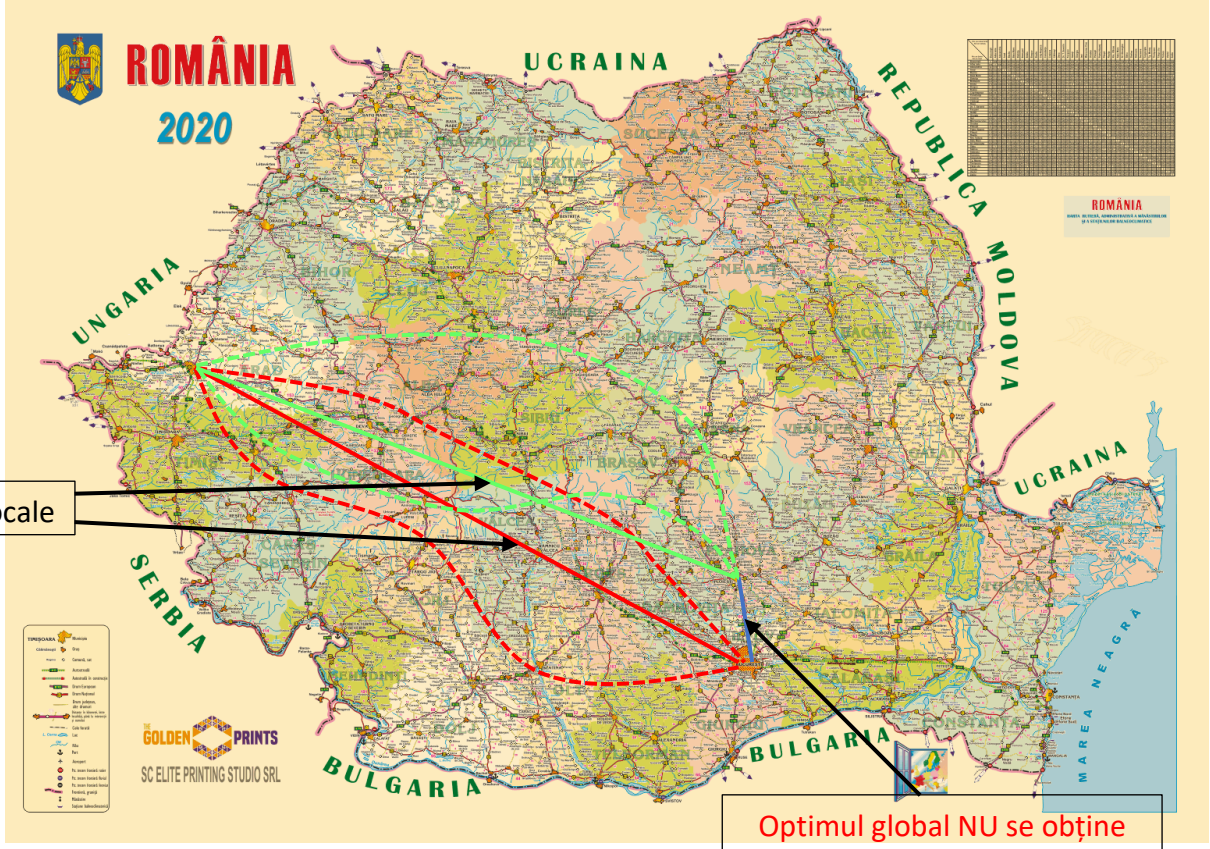
Evident, orice problemă de acest tip poate fi rezolvată prin *metoda forţei-brute*, dar soluţia va avea o complexitate exponenţială, respectiv $\mathcal{O}\left(2^{|A|}\right)$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulţimea *S*, la fiecare pas, cel mai bun element disponibil din mulţimea *A* din punct de vedere al optimizării funcţiei obiectiv.

Practic, metoda Greedy încearcă să găsească optimul global al funcţiei obiectiv combinând optimele sale locale.

Principiul de optim:

1. *Optimul global => optime locale* (forma directă) şi este întotdeauna adevărată
2. *Optime locale => optim global* (forma inversă) şi NU este întotdeauna adevărată



Revenind la problema determinării unei submulţimi S cu sumă maximă, observăm faptul că aceasta trebuie să conţină toate elementele pozitive din mulţimea A, deci criteriul de selecţie este ca elementul curent din A să fie pozitiv (demonstraţia optimalităţii este banală). Dacă mulţimea A nu conţine niciun număr pozitiv, care va fi soluţia problemei?

În anumite probleme, criteriul de selecţie poate fi aplicat mai eficient dacă se realizează o prelucrare iniţială a elementelor mulţimii A – de obicei, o sortare a lor.

Exemplu:
Fie A o mulţime nevidă formată din n numere întregi. Să se determine o submulţime $S \subseteq A$ având exact k elemente $(1 \leq k \leq n)$ *cu proprietatea că suma elementelor sale este maximă.*

A = { 1, -3, -2, ~~5~~, -7, 2, ~~3~~, -5} n = 8 şi k = 5
S = {5, 3, 2, 1, -2}

sorted(A) = {-7, -5, -3, -2, 1, 2, 3, 5}

Soluția:
- *fără sortare* (*implementare directă*): $\mathcal{O}(kn)$
- *cu sortare*: $\mathcal{O}(k + n\log_2 n) \approx \mathcal{O}(n\log_2 n)$

Forma generală a unui algoritm de tip Greedy:

```
prelucrarea inițială a elementelor mulțimii A (opțional)

S = []
for x in A
  if x verifică criteriul de selecție:
      S.append(x)

afișarea elementelor soluției S
```

Complexitățile specifice metodei Greedy:

- $\mathcal{O}(n)$ -> fără sortare și verificarea criteriului de selecție pentru un element x în $\mathcal{O}(1)$
- $\mathcal{O}(n\log_2 n)$ -> cu sortare în $\mathcal{O}(n\log_2 n)$ și verificarea criteriului de selecție pentru un element x în $\mathcal{O}(1)$
- $\mathcal{O}(n^2)$ sau $\mathcal{O}(n^2\log_2 n)$ -> cu sortare în $\mathcal{O}(n\log_2 n)$ și verificarea criteriului de selecție în $\mathcal{O}(n)$

Selecția unui element x din mulțimea A este statică, deci trebuie demonstrată corectitudinea criteriului de selecție!!!

Contraexemplu Greedy

Plata unei sume S folosind n tipuri de monede cu valorile $v_1, v_2,\ldots,v_n$

Dorim să plătim suma S folosind un număr minim de monede!

Idee de tip Greedy: folosim un număr maxim de monede cu valoare maximă la momentul respectiv

a) v = [8, 7, 5] $ și S = 23 $

S = 2*8$ + 7$ = 2*8$ + 1*7$ => Nr. monede = 3 => soluție optimă

b) v = [8, 7, 1] $ și S = 14 $

S = 1*8$ + 6$ = 1*8$ + 6*1$ => Nr. monede = 7 => soluție corectă, dar care nu este optimă (2*7$)

c) v = [8, 7, 5] $ și S = 14 $

S = 1*8$ + 6$ = 1*8$ + 1*5$ + 1$ (rest neplătibil) => nu există soluție!

2. Minimizarea timpului mediu de aşteptare

La un ghişeu, stau la coadă n persoane $p_1, p_2, \dots, p_n$ şi pentru fiecare persoană p_i se cunoaşte timpul său de servire t_i. Să se determine o modalitate de reaşezare a celor n persoane la coadă, astfel încât timpul mediu de aşteptare să fie minim.

Exemplu:

Persoana	**Timpul de servire (t_i)**	**Timp de aşteptare (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 aşteptare (TMA):		$\frac{7+13+16+26+32+35}{6} = \frac{129}{6} = 21.5$

Rearanjarea optimă (în ordinea crescătoare a timpilor de servire):

Persoana	**Timpul de servire (t_i)**	**Timp de aşteptare (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 aşteptare (TMA):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

minimizarea timpului mediu de aşteptare <=>
minimizarea timpului de aşteptare al fiecărei persoane <=>
minimizarea timpilor de aşteptare ai persoanelor aflate înaintea sa

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \dots, p_i, \dots, p_j, \dots, p_n$ 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 $t_1, t_2, \dots, t_n$ 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 $P_2 \neq P_1$ de aranjare $p_1, p_2, \dots, p_j, \dots, p_i, \dots p_n$ 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 TMA_1 este egal cu:

$$TMA_1 = \frac{t_1 + (t_1 + t_2) + \cdots + (t_1 + \cdots + t_i) + \cdots + (t_1 + \cdots + t_j) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_i + \cdots + (n-j+1)t_j + \cdots + 2t_{n-1} + t_n}{n}$$

În cazul celei de-a doua modalități de aranjare P_2, timpul mediu de așteptare TMA_2 este egal cu:

$$TMA_2 = \frac{t_1 + (t_1 + t_2) + \cdots + (t_1 + \cdots + t_j) + \cdots + (t_1 + \cdots + t_i) + \cdots + (t_1 + \cdots + t_n)}{n} =$$
$$= \frac{nt_1 + (n-1)t_2 + \cdots + (n-i+1)t_j + \cdots + (n-j+1)t_i + \cdots + 2t_{n-1} + t_n}{n}$$

Comparăm acum TMA_1 cu TMA_2, calculând diferența dintre ele:

$$TMA_1 - TMA_2 = \frac{(n-i+1)t_i + (n-j+1)t_j - (n-i+1)t_j - (n-j+1)t_i}{n} =$$
$$= \frac{t_i(n-i+1-n+j-1) + t_j(n-j+1-n+i-1)}{n} =$$
$$= \frac{t_i(-i+j) + t_j(-j+i)}{n} = \frac{-t_i(i-j) + t_j(i-j)}{n} = \frac{(t_j - t_i)(i-j)}{n}$$

Deoarece $i < j$ și $t_j > t_i$, obținem faptul că $TMA_1 - TMA_2 = \frac{(t_j - t_i)(i-j)}{n} < 0$ (evident, $n \geq 1$), ceea ce implică $TMA_1 < TMA_2$. Contradicție!!!

Detalii de implementare + complexitate!

$\mathcal{O}(n \log_2 n)$

Forma generală a problemei:

"*Se consideră n activități cu duratele $t_1, t_2, \ldots, t_n$ care partajează o resursă comună. Știind faptul că activitățile trebuie efectuate sub excludere reciprocă* (i.e., *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.*".

3. Planificarea optimă a unor spectacole într-o singură sală

Considerăm n spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$, toate dintr-o singură zi. Având la dispoziție o singură sală, în care putem să planificăm un singur spectacol la un moment dat, să se determine numărul maxim de spectacole care pot fi planificate fără suprapuneri. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, să considerăm $n = 7$ spectacole având următoarele intervale de desfășurare:

$S_1: [10^{00}, 11^{20})$

$S_2: [09^{30}, 12^{10})$

$S_3: [08^{20}, 09^{50})$

$S_4: [11^{30}, 14^{00})$

$S_5: [12^{10}, 13^{10})$

$S_6: [14^{00}, 16^{00})$

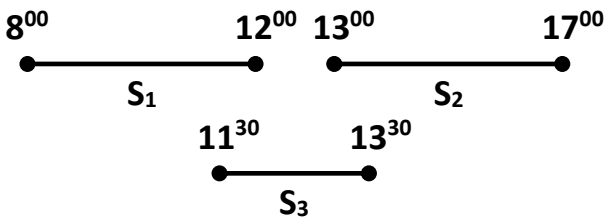
$S_7: [15^{00}, 15^{30})$

În acest caz, numărul maxim de spectacole care pot fi planificate este 4, iar o posibilă soluție este S_3, S_1, S_5 și S_7. Soluția nu este unică (S_3, S_1, S_5 și S_6)!!!

Criterii posibile de selecție:

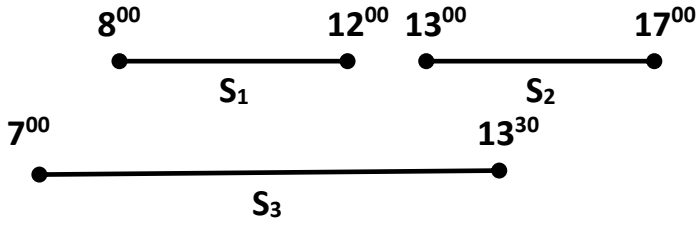
a) în ordinea crescătoare a duratelor
b) în ordinea crescătoare a orelor de început
c) în ordinea crescătoare a orelor de terminare

Criteriul a)



Programare: S3 (optim: S1, S2)

Criteriul b)



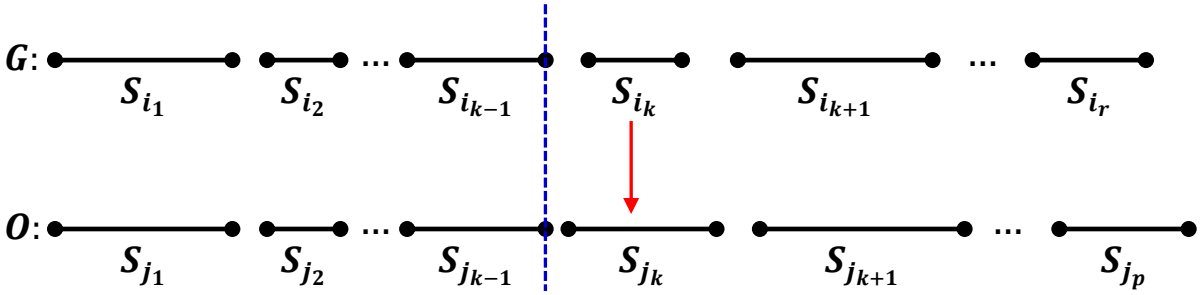
Programare: S3 (optim: S1, S2)

Criteriul c)

.....

Demonstrația optimalității - *exchange argument*:

Fie G soluția furnizată de algoritmul de tip Greedy și o soluție optimă O, diferită de G, obținută folosind orice alt algoritm:



Algoritmul Greedy:

- sortăm spectacolele în ordinea crescătoare a orelor de terminare;
- planificăm primul spectacol (problema are întotdeauna soluție!);
- pentru fiecare spectacol rămas, verificăm dacă începe după ultimul spectacol programat și, în caz afirmativ, îl planificăm și pe el.

Complexitate: $\mathcal{O}(n \log_2 n)$.

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 submulţimi a unei mulţimi date pentru care se minimizează sau se maximizează valoarea unei funcţii obiectiv. Formal, o problemă de optimizare poate fi enunţată astfel: "*Fie A o mulţime nevidă şi $f: \mathcal{P}(A) \to \mathbb{R}$ o funcţie obiectiv asociată mulţimii A, unde prin $\mathcal{P}(A)$ am notat mulţimea tuturor submulţimilor mulţimii A. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei f să fie minimă/maximă pe S (i.e., pentru orice altă submulţime $T \subseteq A, T \neq S$, valoarea funcţiei obiectiv f va fi cel puţin /cel mult egală cu valoarea funcţiei obiectiv f pe submulţimea S).*"

O problemă foarte simplă de optimizare este următoarea: "*Fie A o mulţime nevidă de numere întregi. Să se determine o submulţime $S \subseteq A$ cu proprietatea că suma elementelor sale este maximă.*". Se observă faptul că funcţia obiectiv nu este dată în formă matematică şi nu se precizează explicit faptul că suma elementelor submulţimii S trebuie să fie maximă *în raport cu suma oricărei alte submulţimi*, acest lucru subînţelegându-se. Formal, problema poate fi enunţată astfel: "*Fie $A \subseteq \mathbb{Z}, A \neq \emptyset$ şi $f: \mathcal{P}(A) \to \mathbb{R}$, $f(S) = \sum_{x \in S} x$. Să se determine o submulţime $S \subseteq A$ astfel încât valoarea funcţiei 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 forţei-brute*, astfel: se generează, pe rând, toate submulţimile S ale mulţimii A şi pentru fiecare dintre ele se calculează $f(S)$, iar dacă valoarea obţinută este mai mică/mai mare decât minimul/maximul obţinut până în acel moment, atunci se actualizează minimul/maximul şi se reţine submulţimea S. Deşi aceasta rezolvare este corectă, ea are o complexitate exponenţială, respectiv $\mathcal{O}(2^{|A|})$!

Tehnica de programare Greedy încearcă să rezolve problemele de optimizare adăugând în submulţimea S, la fiecare pas, cel mai bun element disponibil din mulţimea A din punct de vedere al optimizării funcţiei obiectiv. Practic, metoda Greedy încearcă să găsească optimul global al funcţiei obiectiv combinând optimele sale locale. Totuşi, prin combinarea unor optime locale nu se obţine întotdeauna un optim global! De exemplu, să considerăm cel mai scurt drum posibil dintre Bucureşti şi Arad (un optim local), precum şi cel mai scurt drum posibil dintre Arad şi Ploieşti (alt optim local). Combinând cele două optime locale nu vom obţine un optim global, deoarece, evident, cel mai scurt drum de la Bucureşti la Ploieşti nu trece prin Arad! Din acest motiv, aplicare tehnicii de programare Greedy pentru rezolvarea unei probleme trebuie să fie însoţită de o demonstraţie a corectitudinii (optimalităţii) criteriului de selecţie pe care trebuie să-l îndeplinească un element al mulţimii A pentru a fi adăugat în soluţia S.

De exemplu, să considerăm *problema plății unei sume folosind un număr minim de monede*. O rezolvare de tip Greedy a acestei probleme ar putea consta în utilizarea, la fiecare pas, a unui număr maxim de monede cu cea mai mare valoare admisibilă. Astfel, pentru monede cu valorile de 8$, 7$ şi 5 $, o sumă de 23$ va fi plătită în următorul mod: 23$ = 2*8$ + 7$ = 2*8$ + 1*7$, deci se vor utiliza 3 monede, ceea ce reprezintă o soluție optimă. Dacă vom considera monede cu valorile de 8$, 7$ şi 1 $, o sumă de 14$ va fi plătită în următorul mod: 14$ = 1*8$ + 6$ = 1*8$ + 6*1$, deci se vor utiliza 7 monede, ceea ce nu reprezintă o soluție optimă (i.e., 2*7$). Mai mult, pentru monede cu 8$, 7$ şi 5 $, o sumă de 14$ nu va putea fi plătită deloc: 14$ = 1*8$ + 6$ = 1*8$ + 1*5$ + 1$, deoarece restul rămas, de 1$, nu mai poate fi plătit (evident, soluția optimă este 2*7$). În concluzie, acest algoritm de tip Greedy, numit *algoritmul casierului*, nu furnizează întotdeauna o soluție optimă pentru plata unei sume folosind un număr minim de monede. Totuşi, pentru anumite valori ale monedelor, el poate furniza o soluție optimă pentru orice sumă dată (de exemplu, pentru monedele din Statele Unite ale Americii: https://personal.utdallas.edu/~sxb027100/cs6363/coin.pdf)

Revenind la problema determinării unei submulţimi S cu sumă maximă, observăm faptul că aceasta trebuie să conţină toate elementele pozitive din mulţimea A, deci criteriul de selecţie este ca elementul curent din A să fie pozitiv (demonstraţia optimalităţii este banală). Dacă mulţimea A nu conţine niciun număr pozitiv, care va fi soluția problemei?

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

- de k ori se selectează maximul din mulţimea A şi 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 şi $\mathcal{O}(n^2)$ pentru valori ale lui k apropiate de n;
- sortăm crescător elementele mulţimii A şi 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 de implementare are avantajul unei implementări 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 elementelor mulțimii A
S = []
for x in A:
   if elementul x verifică criteriul de selecție:
        S.append(x)
afișarea elementelor mulțimii S
```

Se observă faptul că, de obicei, un algoritm de acest tip are o complexitate relativ mică, de tipul $\mathcal{O}(n \log_2 n)$, dacă prin sortarea (prelucrarea) elementelor mulțimii A cu complexitatea $\mathcal{O}(n \log_2 n)$ se poate ulterior testa criteriul de selecție în $\mathcal{O}(1)$. Dacă nu se realizează prelucrarea inițială a elementelor mulțimii A, atunci algoritmul (care trebuie puțin adaptat) va avea complexități de tipul $\mathcal{O}(n)$ sau $\mathcal{O}(n^2)$, induse de complexitatea verificării criteriului de selecție. Evident, acestea nu sunt toate complexitățile posibile pentru un algoritm de tip Greedy, ci doar sunt cele mai des întâlnite!

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 aşteptare se obţine rearanjând persoanele la coadă în ordinea crescătoare a timpilor de servire:

Persoana	**Timpul de servire** (t_i)	**Timp de aşteptare** (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 aşteptare (*M*):		$\frac{3+6+12+18+25+35}{6} = \frac{99}{6} = 16.5$

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

Pentru a demonstra mai simplu corectitudinea algoritmului, mai întâi vom renumerota persoanele $p_1, p_2, \dots, p_i, \dots, p_j, \dots, p_n$ în ordinea crescătoare a timpilor de servire, astfel încât vom avea $t_1 \le t_2 \le \cdots \le t_i \le \cdots \le t_j \le \cdots \le t_n$. De asemenea, vom presupune faptul că timpii individuali de servire $t_1, t_2, \dots, t_n$ nu sunt toţi 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 $p_1, p_2, \dots, p_j, \dots, p_i, \dots p_n$ diferită de cea iniţială, în care $t_j > t_i$ (practic, am interschimbat persoanele p_i şi p_j din varianta iniţială, adică persoana p_j se află acum pe poziţia i în coadă, iar persoana p_i se află acum pe poziţia j, unde $i < j$).

În cazul primei modalităţi de aranjare P_1, timpul mediu de servire 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 modalităţi de aranjare P_2, timpul mediu de servire 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, un tuplu, iar apoi să le sortăm crescător după componenta t_i.

```
# functie folosita pentru sortarea crescătoare a persoanelor
# în raport de timpii de servire (cheia)
def cheieTimpServire(t):
    return t[1]

# funcția afișează, într-un format tabelar, timpii de servire
# și timpii de așteptare ai persoanelor
# ts = o listă cu timpii individuali de serivre
def afisareTimpi(ts):
    print("Persoana\tTimp de servire\tTimp de asteptare")
    # timpul de așteptare al persoanei curente
    tcrt = 0
    # timpul total de așteptare
    ttotal = 0
    for t in ts:
        tcrt = tcrt + t[1]
        ttotal = ttotal + tcrt
        print(str(t[0]).center(len("Persoana")),
              str(t[1]).center(len("Timp de serivire")),
              str(tcrt).center(len("Timp de așteptare")), sep="\t")
    print("Timpul mediu de așteptare:", round(ttotal/len(ts), 2))


# timpii de servire ai persoanelor se citesc de la tastatură
aux = [int(x) for x in input("Timpii de servire: ").split()]
# asociem fiecărui timp de servire numărul de ordine al persoanei
tis = [(i+1, aux[i]) for i in range(len(aux))]
```

```
print("Varianta inițială:")
afisareTimpi(tis)

# sortăm persoanele în ordinea crescătoare a timpilor de servire
tis.sort(key=cheieTimpServire)

print("\nVarianta optimă:")
afisareTimpi(tis)
```

Evident, complexitatea algoritmului este dată de complexitatea operației de sortare utilizate, deci complexitatea sa, optimă, este $\mathcal{O}(n \log_2 n)$.

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.*".

3. Planificarea optimă a unor spectacole într-o singură sală

Considerăm n spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaștem intervalele lor de desfășurare $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$, toate dintr-o singură zi. Având la dispoziție o singură sală, în care putem să planificăm un singur spectacol la un moment dat, să se determine numărul maxim de spectacole care pot fi planificate fără suprapuneri. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, să considerăm $n = 7$ spectacole având următoarele intervale de desfășurare:

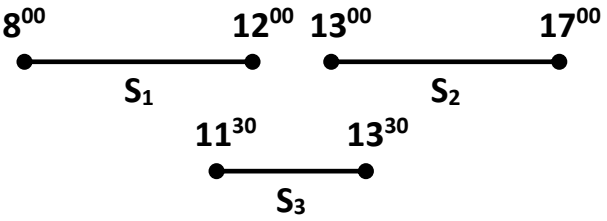
S_1: $[10^{00}, 11^{20})$
S_2: $[09^{30}, 12^{10})$
S_3: $[08^{20}, 09^{50})$
S_4: $[11^{30}, 14^{00})$
S_5: $[12^{10}, 13^{10})$
S_6: $[14^{00}, 16^{00})$
S_7: $[15^{00}, 15^{30})$

Se observă faptul că numărul maxim de spectacole care pot fi planificate este 4, iar o posibilă soluție este S_3, S_1, S_5 și S_7. Atenție, soluția nu este unică (de exemplu, o altă soluție optimă este S_3, S_1, S_5 și S_6)!

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca planificarea spectacolelor folosind unul dintre următoarele criterii:

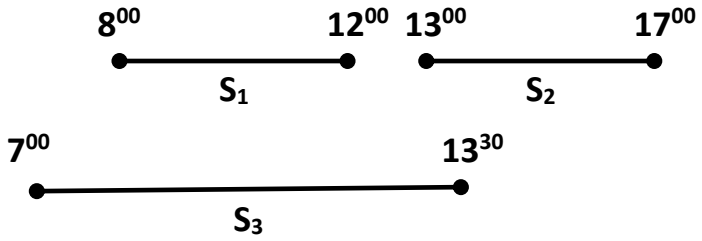
a) în ordinea crescătoare a duratelor;
b) în ordinea crescătoare a orelor de început;
c) în ordinea crescătoare a orelor de terminare.

În cazul utilizării criteriului a), se observă uşor faptul că nu vom obţine întotdeauna o soluţie optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul a), vom planifica prima dată spectacolul S_3 (deoarece durează cel mai puţin), iar apoi nu vom mai putea planifica nici spectacolul S_1 şi nici spectacolul S_2, deoarece ambele se suprapun cu spectacolul S_3, deci vom obţine o planificare formată doar din S_3. Evident, planificarea optimă, cu număr maxim de spectacole, este S_1 şi S_2.

De asemenea, în cazul utilizării criteriului b), se observă uşor faptul că nu vom obţine întotdeauna o soluţie optimă. De exemplu, să considerăm următoarele 3 spectacole:



Aplicând criteriul b), vom planifica prima dată spectacolul S_3 (deoarece începe primul), iar apoi nu vom mai putea planifica nici spectacolul S_1 şi nici spectacolul S_2, deoarece ambele se suprapun cu el, deci vom obţine o planificare formată doar din S_3. Evident, planificarea optimă este S_1 şi S_2.

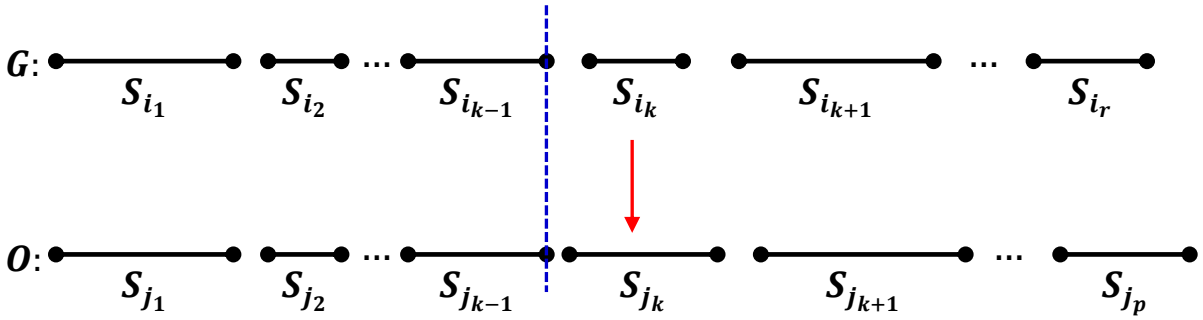
În cazul utilizării criteriului c), se observă faptul că vom obţine soluţiile optime în ambele exemple prezentate mai sus:

- în primul exemplu, vom planifica mai întâi spectacolul S_1 (deoarece se termină primul), apoi nu vom putea planifica spectacolul S_3 (deoarece se suprapune cu S_1), dar vom putea planifica spectacolul S_2, deci vom obţine planificarea optimă formată din S_1 şi S_2;
- în al doilea exemplu, vom proceda la fel şi vom obţine planificarea optimă formată din S_1 şi S_2.

Practic, criteriul c) este o combinaţie a criteriilor a) şi b), deoarece un spectacol care durează puţin şi începe devreme se va termina devreme!

Pentru a demonstra optimalitatea criteriului c) de selecţie, vom utiliza o demonstraţie de tipul *exchange argument*: vom considera o soluţie optimă furnizată de un algoritm oarecare (nu contează metoda utilizată!), diferită de soluţia furnizată de algoritmul de tip Greedy, şi vom demonstra faptul că aceasta poate fi transformată, element cu element, în soluţia furnizată de algoritmul de tip Greedy. Astfel, vom demonstra faptul că şi soluţia furnizată de algoritmul de tip Greedy este tot optimă!

Fie G soluţia furnizată de algoritmul de tip Greedy şi o soluţie optimă O, diferită de G, obţinută folosind orice alt algoritm:



Deoarece soluţia optimă O este diferită de soluţia Greedy G, rezultă că există un cel mai mic indice k pentru care $S_{i_k} \neq S_{j_k}$. Practic, este posibil ca ambii algoritmi pot să selecteze, până la pasul $k-1$, aceleaşi spectacole în aceeaşi ordine, adică $S_{i_1} = S_{j_1}, \dots, S_{i_{k-1}} = S_{j_{k-1}}$. Spectacolul S_{j_k} din soluţia optimă O poate fi înlocuit cu spectacolul S_{i_k} din soluţia Greedy G fără a produce o suprapunere, deoarece:

- spectacolul S_{i_k} începe după spectacolul $S_{j_{k-1}}$, deoarece spectacolul S_{i_k} a fost programat după spectacolul $S_{i_{k-1}}$ care este identic cu spectacolul $S_{j_{k-1}}$, deci $s_{i_k} \geq f_{i_{k-1}} = f_{j_{k-1}}$;
- spectacolul S_{j_k} se termină după spectacolul S_{i_k}, adică $f_{j_k} \geq f_{i_k}$, deoarece, în caz contrar ($f_{j_k} < f_{i_k}$) algoritmul Greedy ar fi selectat spectacolul S_{j_k} în locul spectacolului S_{i_k};
- spectacolul S_{i_k} se termină înaintea spectacolului $S_{j_{k+1}}$, adică $f_{i_k} \leq s_{j_{k+1}}$, deoarece am demonstrat anterior faptul că $f_{i_k} \leq f_{j_k}$ şi $f_{j_k} \leq s_{j_{k+1}}$ (deoarece spectacolul $S_{j_{k+1}}$ a fost programat după spectacolul S_{j_k}).

Astfel, am demonstrat faptul că $f_{j_{k-1}} \leq s_{i_k} < f_{j_k} \leq s_{j_{k+1}}$, ceea ce ne permite să înlocuim spectacolul S_{j_k} din soluţia optimă O cu spectacolul S_{i_k} din soluţia Greedy G fără a produce o suprapunere. Repetând raţionamentul anterior, putem transforma primele r elemente din soluţia optimă O în soluţia G furnizată de algoritmul Greedy.

Pentru a încheia demonstraţia, trebuie să mai demonstrăm faptul că ambele soluţii conţin acelaşi număr de spectacole, respectiv $r = p$. Presupunem prin absurd faptul că $r \neq p$. Deoarece soluţia O este optimă, rezultă faptul că $p > r$ (altfel, dacă $p < r$, ar însemna că soluţia optimă O conţine mai puţine spectacole decât soluţia Greedy G, ceea ce i-ar contrazice optimalitatea), deci există cel puţin un spectacol $S_{j_{r+1}}$ în soluţia optimă

O care nu a fost selectat în soluția Greedy G. Acest lucru este imposibil, deoarece am demonstrat anterior faptul că orice spectacol S_{j_k} din soluția optimă se termină după spectacolul S_{i_k} aflat pe aceeași poziție în soluția Greedy (adică $f_{j_k} \geq f_{i_k}$), deci am obține relația $f_{i_r} \leq f_{j_r} \leq s_{j_{r+1}}$, ceea ce ar însemna că spectacolul $S_{j_{r+1}}$ ar fi trebuit să fie selectat și în soluția Greedy G! În concluzie, presupunerea că $r \neq p$ este falsă, deci $r = p$.

Astfel, am demonstrat faptul că putem transforma soluția optimă O în soluția G furnizată de algoritmul Greedy, deci și soluția furnizată de algoritmul Greedy este optimă!

În concluzie, algoritmul Greedy pentru rezolvarea problemei programării spectacolelor este următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de terminare;
- planificăm primul spectacol (problema are întotdeauna soluție!);
- pentru fiecare spectacol rămas, verificăm dacă începe după ultimul spectacol programat și, în caz afirmativ, îl planificăm și pe el.

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, programarea primului spectacol are complexitatea $\mathcal{O}(1)$, testarea spectacolelor rămase are complexitatea $\mathcal{O}(n-1)$, iar afișarea planificării optime are cel mult complexitatea $\mathcal{O}(n)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.

În continuare, vom prezenta implementarea algoritmului în limbajul Python:

```
# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de sfârșit (cheia)
def cheieOraSfârșit(sp):
    return sp[2]

# citim datele de intrare din fișierul text "spectacole.txt"
fin = open("spectacole.txt")
# lsp = lista spectacolelor, fiecare spectacol fiind memorat
# sub forma unui tuplu (ID, ora de început, ora de sfârșit)
lsp = []
crt = 1
for linie in fin:
    aux = linie.split("-")
    # aux[0] = ora de început a spectacolului curent
    # aux[1] = ora de sfârșit a spectacolului curent
    lsp.append((crt, aux[0].strip(), aux[1].strip()))
    crt = crt + 1
fin.close()

# sortăm spectacolele în ordinea crescătoare a timpilor de sfârșit
lsp.sort(key=cheieOraSfârșit)
```

```
# posp = o listă care conține o programare optima a spectacolelor,
# inițializată cu primul spectacol
posp = [lsp[0]]
# parcurgem restul spectacolelor
for sp in lsp[1:]:
    # dacă spectacolul curent începe după ultimul spectacol
    # programat, atunci îl programăm și pe el
    if sp[1] >= posp[len(posp)-1][2]:
        posp.append(sp)

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numarul maxim de spectacole: "+str(len(posp))+"\n")
fout.write("\nSpectacolele programate:\n")
for sp in posp:
    fout.write(sp[1]+"-"+sp[2]+" Spectacol "+str(sp[0])+"\n")
fout.close()
```

Pentru exemplul de mai sus, fişierele text de intrare şi de ieşire sunt următoarele:

spectacole.txt	**programare.txt**
10:00-11:20 09:30-12:10 08:20-09:50 11:30-14:00 12:10-13:10 14:00-16:00 15:00-15:30	Numarul maxim de spectacole: 4 Spectacolele programate: 08:20-09:50 Spectacol 3 10:00-11:20 Spectacol 1 12:10-13:10 Spectacol 5 15:00-15:30 Spectacol 7

Încheiem prezentarea acestei probleme precizând faptul că este tot o problemă de planificare, forma sa generală fiind următoarea: "*Se consideră n activităţi pentru care se cunosc intervalele orare de desfăşurare şi care partajează o resursă comună. Ştiind faptul că activităţile trebuie efectuate sub excludere reciprocă* (*respectiv, la un moment dat resursa comună poate fi alocată unei singure activităţi*)*, să se determine o modalitate de planificare a unui număr maxim de activităţi care nu se suprapun.*".

4. Planificarea unor spectacole folosind un număr minim de săli

Considerăm n spectacole $S_1, S_2, \dots, S_n$ pentru care cunoaştem intervalele lor de desfăşurare $[s_1, f_1), [s_2, f_2), \dots, [s_n, f_n)$, toate dintr-o singură zi. Să se determine numărul minim de săli necesare astfel încât toate spectacolele să fie programate astfel încât să nu existe suprapuneri în nicio sală. Un spectacol S_j poate fi programat după spectacolul S_i dacă $s_j \geq f_i$.

De exemplu, cele 7 spectacolele de mai jos pot fi planificate folosind minim 3 săli:

spectacole.txt	programare.txt
10:00-11:20 09:30-12:10 08:20-09:50 11:30-14:00 12:10-13:10 11:15-13:15 15:00-15:30	Numar minim de sali: 3 Sala 1: 11:15-13:15 Spectacol 6 Sala 2: 08:20-09:50 Spectacol 3 10:00-11:20 Spectacol 1 11:30-14:00 Spectacol 4 Sala 3: 09:30-12:10 Spectacol 2 12:10-13:10 Spectacol 5 15:00-15:30 Spectacol 7

Deoarece dorim să găsim o rezolvare de tip Greedy a acestei probleme, vom încerca să planificăm spectacolele folosind următoarea strategie: vom încerca, pe rând, să planificăm fiecare spectacol într-una dintre sălile deja utilizate (după ultimul spectacol planificat în sala respectivă), iar dacă acest lucru nu este posibil, vom programa spectacolul respectiv într-o sală nouă (i.e., o sală neutilizată până atunci).

Spectacolele pot fi parcurse în mai multe moduri: în ordinea crescătoare a orelor de terminare, în ordinea crescătoare a duratelor sau în ordinea crescătoare a orelor de început. În continuare, vom analiza planificările pe care le vom obține pentru spectacolele din exemplul dat, utilizând una dintre modalități de parcurgere precizate anterior:

a) *în ordinea crescătoare a orelor de terminare*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S1) se poate planifica tot în sala 1, al treilea spectacol (S2) nu se poate planifica tot în Sala 1, deci va fi planificat într-o sală nouă (Sala 2) ș.a.m.d.

Spectacol	Interval de desfășurare	Sala
S3	08:20 – 09:50	1
S1	10:00 – 11:20	1
S2	09:30 – 12:10	2
S5	12:10 – 13:10	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	4
S7	15:00 – 15:30	1

Evident, planificarea obținută nu este optimă, deoarece folosește 4 săli în loc de 3!

b) *în ordinea crescătoare a duratelor*: vom planifica primul spectacol (S7) în Sala 1, al doilea spectacol (S5) nu se poate planifica tot în Sala 1 (deoarece nu începe după ultimul spectacol planificat în Sala 1!), deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol nu se poate programa niciuna dintre sălile 1 şi 2, deci va fi programat într-o sală nouă (Sala 3) ş.a.m.d.

Spectacol	Interval de desfăşurare	Durată	Sala
S7	15:00 – 15:30	0:30	1
S5	12:10 – 13:10	1:00	2
S1	10:00 – 11:20	1:20	3
S3	08:20 – 09:50	1:30	4
S6	11:15 – 13:15	2:00	4
S4	11:30 – 14:00	2:30	3
S2	09:30 – 12:10	2:40	5

Evident, nici această planificarea nu este optimă, deoarece foloseşte 5 săli în loc de 3!

c) *în ordinea crescătoare a orelor de început*: vom planifica primul spectacol (S3) în Sala 1, al doilea spectacol (S2) nu se poate planifica tot în Sala 1, deci îl vom planifica într-o sală nouă (Sala 2), al treilea spectacol (S1) se poate programa tot în Sala 1 ş.a.m.d.

Spectacol	Interval de desfăşurare	Sala
S3	08:20 – 09:50	1
S2	09:30 – 12:10	2
S1	10:00 – 11:20	1
S6	11:15 – 13:15	3
S4	11:30 – 14:00	1
S5	12:10 – 13:10	2
S7	15:00 – 15:30	1

Evident, această planificarea este optimă, deoarece foloseşte tot 3 săli, chiar dacă spectacolele sunt distribuite în săli altfel decât în exemplul dat!

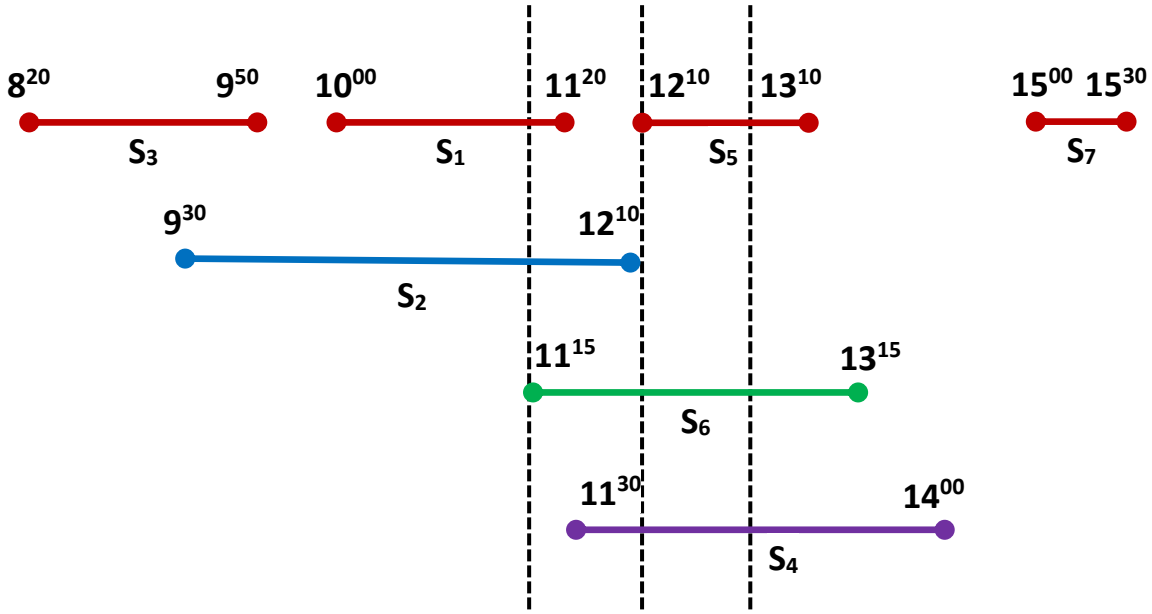
Astfel, analizând exemplele de mai sus, un algoritm Greedy posibil corect, pare a fi următorul:

- sortăm spectacolele în ordinea crescătoare a orelor de început;
- planificăm primul spectacol în prima sală;
- fiecare spectacol rămas îl planificăm fie în prima sală deja utilizată în care acest lucru este posibil (i.e., ora de început a spectacolului curent este mai mare sau egală decât ora de terminare a ultimului spectacol programat în sala respectivă), fie utilizăm o nouă sală pentru el.

În continuare, vom demonstra corectitudinea algoritmului Greedy propus mai sus, folosind următoarele observații:

a) *Algoritmul Greedy nu planifică niciodată două spectacole care se suprapun în aceeași sală*. Argumentați!

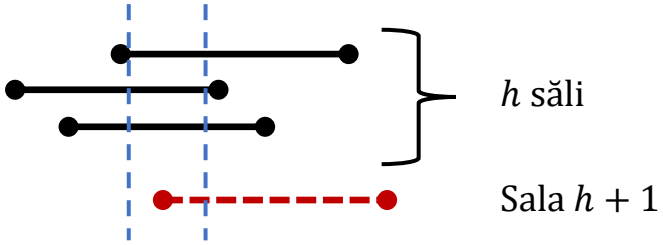
b) Definim *adâncimea h a unui șir de intervale de desfășurare ale unor spectacole* ca fiind numărul maxim de spectacole care se suprapun în orice moment posibil. De exemplu, considerând spectacolele din exemplele de mai sus, putem observa faptul că adâncimea șirului intervalelor lor de desfășurare este $h = 3$:



Se observă foarte ușor faptul că *orice planificare corectă a unor spectacole va utiliza un număr de săli cel puțin egal cu adâncimea șirului intervalelor lor de desfășurare*!

c) Pentru un șir de intervale de desfășurare ale unor spectacole având adâncimea h, planificarea realizată de algoritmul Greedy va folosi cel mult h săli.

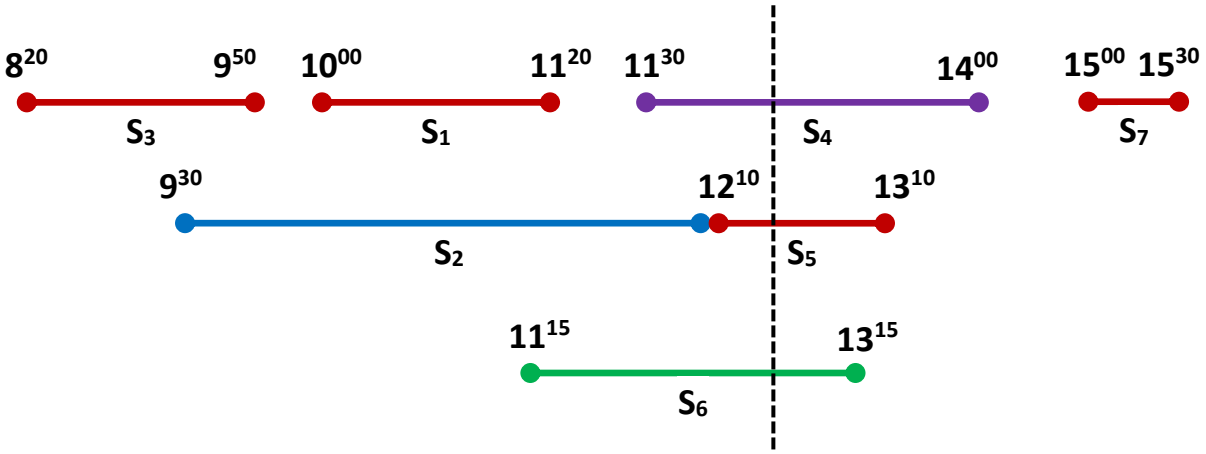
Demonstrație: Presupunem faptul că algoritmul Greedy ar folosi cel puțin $h + 1$ săli pentru a planifica un șir de spectacole având adâncimea intervalelor lor de desfășurare egală cu h. Acest lucru s-ar putea întâmpla doar dacă algoritmul Greedy ar fi utilizat deja h săli pentru a planifica spectacolele anterioare spectacolului curent, iar spectacolul curent nu ar putea fi planificat în niciuna dintre ele:



Acest lucru ar însemna faptul că spectacolul curent începe strict înaintea minimului dintre orele de terminare ale spectacolelor deja planificate, deoarece, în caz contrar, spectacolul curent ar fi fost programat într-una dintre sălile deja utilizate. Totodată, ştim faptul că spectacolul curent începe după maximul dintre orele de început ale spectacolelor deja planificate, deoarece acestea sunt ordonate crescător după ora de început. Aşadar, există un interval de timp (delimitat în figura de mai sus de cele două linii albastre întrerupte) în care spectacolul curent se suprapune cu câte un spectacol din fiecare dintre cele h săli deja utilizate, deci adâncimea şirului de intervale de desfăşurare ale spectacolelor respective ar fi $h + 1$. Acest lucru reprezintă o contradicţie cu faptul că adâncimea şirului intervalelor de desfăşurare ale spectacolelor date este h, deci presupunerea făcută este falsă şi, în consecinţă, algoritmul Greedy va utiliza cel mult h săli pentru a planifica spectacolele respective.

Din observaţiile b) şi c) rezultă faptul că algoritmul Greedy este optim, deoarece va utiliza exact h săli pentru a planifica spectacole având adâncimea şirului intervalelor de desfăşurare egală cu h.

Pentru spectacolele din exemplul utilizat, având adâncimea şirului intervalelor de desfăşurare $h = 3$, o planificare optimă (i.e., care foloseşte 3 săli) este următoarea:



În limbajul Python se poate implementa uşor acest algoritm, păstrând sălile într-o listă, iar fiecare sală va fi tot o listă conţinând spectacolele planificate în ea. Totuşi, această variantă de implementare ar avea complexitatea maximă $\mathcal{O}(n^2)$, deoarece ora de început a fiecăruia dintre cele n spectacole date ar trebui să fie comparată cu ora de terminare a ultimului spectacol planificat în fiecare dintre cele maxim n săli.

O implementare cu complexitatea optimă $\mathcal{O}(n\log_2 n)$ necesită utilizarea unei structuri de date numită *coadă cu priorităţi* (*priority queue*). Într-o astfel de structură de date, fiecărei valori îi este asociat un număr întreg reprezentând prioritatea sa, iar operaţiile specifice unei cozi se realizează în funcţie de priorităţile elementelor, ci nu în funcţie de ordinea în care ele au fost inserate. Astfel, operaţia de inserare a unui nou element şi operaţia de extragere a elementului cu prioritate minimă/maximă (depinde de tipul cozii cu priorităţi, respectiv *min-priority queue* sau *max-priority queue*) vor avea, de obicei, complexitatea $\mathcal{O}(\log_2 n)$.

În limbajul Python, clasa `PriorityQueue` implementează o coadă cu priorităţi în care un element trebuie să fie un tuplu de forma $(prioritate, valoare)$. Principalele metode ale acestei clase sunt:

- `get()`: furnizează elementul din coadă care are prioritatea minimă sau, dacă există mai multe elemente cu prioritate minimă, pe primul inserat;
- `put(element)`: inserează în coadă un element de forma indicată mai sus;
- `empty()`: returnează `True` în cazul în care coada nu mai conţine niciun element sau `False` în caz contrar;
- `qsize()`: returnează numărul de elemente din coadă.

Aşa cum deja am menţionat, metodele `get` şi `put` au complexitatea $\mathcal{O}(\log_2 n)$, iar metodele `empty` şi `qsize` au complexitatea $\mathcal{O}(1)$.

În implementarea algoritmului Greedy prezentat, vom folosi o coadă cu priorităţi pentru a memora sălile utilizate, prioritatea unei săli fiind dată de ora de terminare a ultimului spectacol planificat în ea, iar spectacolele planificate într-o sală vor fi păstrate folosind o listă. Astfel, vom extrage sala cu timpul minim de terminare al ultimului spectacol planificat în ea şi vom verifica dacă spectacolul curent poate fi planificat în sala respectivă sau nu (dacă spectacolul curent nu poate fi planificat în această sală, atunci el nu poate fi planificat în nicio altă sală!). În caz afirmativ, vom planifica spectacolul curent în sala respectivă şi îi vom actualiza prioritatea la ora de terminare a spectacolului adăugat, altfel vom insera în coadă o sală nouă în care vom planifica spectacolul curent şi prioritatea sălii va fi egală cu ora de terminare a spectacolului respectiv.

Considerând faptul că fişierele de intrare şi ieşire sunt de forma prezentată în primul exemplu, o implementare în limbajul Python a algoritmului Greedy este următoarea:

```
import queue

# functie folosita pentru sortarea crescătoare a spectacolelor
# în raport de ora de început (cheia)
def cheieOraÎnceput(sp):
    return sp[1]


# citim datele de intrare din fișierul text "spectacole.txt"
fin = open("spectacole.txt")
# lsp = lista spectacolelor, fiecare spectacol fiind memorat
# sub forma unui tuplu (ID, ora de început, ora de sfârșit)
lsp = []
crt = 1
for linie in fin:
    aux = linie.split("-")
    # aux[0] = ora de început a spectacolului curent
    # aux[1] = ora de sfârșit a spectacolului curent
    lsp.append((crt, aux[0].strip(), aux[1].strip()))
    crt = crt + 1
fin.close()
```

```
# sortăm spectacolele crescător după orelor de început
lsp.sort(key=cheieOraÎnceput)

# sălile vor fi stocate într-o coadă cu priorități în care
# prioritatea unei săli este dată de ora de terminare a
# ultimului spectacol planificat în sala respectivă, iar
# spectacolele planificate în ea vor fi păstrate într-o listă
sali = queue.PriorityQueue()

# planificăm primul spectacol în prima sală
sali.put((lsp[0][2], list((lsp[0],))))

# parcurgem restul spectacolelor
for k in range(1, len(lsp)):
    # extragem sala cu ora minimă de terminare a ultimului
    # spectacol planificat în ea
    min_timp_final = sali.get()

    # dacă spectacolul curent lsp[k] poate fi planificat în
    # sala extrasă, atunci îl adăugăm în lista spectacolelor
    # planificate în ea și reintroducem sala în coada cu
    # priorități, dar cu prioritatea actualizată la ora de
    # terminare a spectacolului adăugat
    if lsp[k][1] >= min_timp_final[0]:
        min_timp_final[1].append(lsp[k])
        sali.put((lsp[k][2], min_timp_final[1]))
    # dacă spectacolul curent lsp[k] nu poate fi planificat în
    # sala extrasă, atunci reintroducem sala extrasă în coada
    # cu priorități fără a-i modifica prioritatea și adăugăm
    # o sală nouă în care planificăm spectacolul curent
    else:
        sali.put(min_timp_final)
        sali.put((lsp[k][2], list((lsp[k],))))

# scriem datele de ieșire în fișierul text "programare.txt"
fout = open("programare.txt", "w")
fout.write("Numar minim de sali: " + str(sali.qsize()) + "\n")

scrt = 1
while not sali.empty():
    sala = sali.get()
    fout.write("\nSala " + str(scrt) + ":\n")
    for sp in sala[1]:
        fout.write("\t"+sp[1]+"-"+sp[2]+" Spectacol "+
                   str(sp[0]) + "\n")
    scrt += 1

fout.close()
```

Observați faptul că în fișierul de ieșire sălile vor fi scrise în ordinea priorităților, ci nu în ordinea în care au fost inserate!

5. Problema rucsacului (varianta continuă/fracţionară)

Considerăm un rucsac având capacitatea maximă G şi n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaştem greutăţile lor $g_1, g_2, \dots, g_n$ şi câştigurile $c_1, c_2, \dots, c_n$ obţinute prin încărcarea lor completă în rucsac. Ştiind faptul că orice obiect poate fi încărcat şi fracţionat (doar o parte din el), să se determine o modalitate de încărcare a rucsacului astfel încât câştigul total obţinut să fie maxim. Dacă un obiect este încărcat fracţionat, atunci vom obţine un câştig direct proporţional cu fracţiunea încărcată din el (de exemplu, dacă vom încărca doar o treime dintr-un obiect, atunci vom obţine un câştig egal cu o treime din câştigul integral asociat obiectului respectiv).

În afara variantei continue/fracţionare a problemei rucsacului, mai există şi varianta discretă a sa, în care un obiecte poate fi încărcat doar complet. Varianta respectivă nu se poate rezolva corect utilizând metoda Greedy, ci există alte metode de rezolvare, pe care le vom prezenta în cursul dedicat metodei programării dinamice.

Se observă foarte uşor faptul că varianta fracţionară a problemei rucsacului are întotdeauna soluţie (evident, dacă $G > 0$ şi $n \geq 1$), chiar şi în cazul în care cel mai mic obiect are o greutate strict mai mare decât capacitatea G a rucsacului (deoarece putem să încărcăm şi fracţiuni dintr-un obiect), în timp ce varianta discretă nu ar avea soluţie în acest caz.

Deoarece dorim să găsim o rezolvare de tip Greedy pentru varianta fracţionară a problemei rucsacului, vom încerca să încărcăm obiectele în rucsac folosind unul dintre următoarele criterii:

a) în ordinea descrescătoare a câştigurilor integrale (cele mai valoroase obiecte ar fi primele încărcate);
b) în ordinea crescătoare a greutăţilor (cele mai mici obiecte ar fi primele încărcate, deci am încărca un număr mare de obiecte în rucsac);
c) în ordinea descrescătoare a greutăţilor.

Analizând cele 3 criterii propuse mai sus, putem găsi uşor contraexemple care să dovedească faptul că nu vom obţine o soluţii optime. De exemplu, criteriul c) ar putea fi corect doar presupunând faptul că, întotdeauna, un obiect cu greutate mare are asociat şi un câştig mare, ceea ce, evident, nu este adevărat! În cazul criteriului a), considerând $G =$ 10 kg şi 3 obiecte având câştigurile $(100, 90, 80)$ RON şi greutăţile $(10, 5, 5)$ kg, vom încărca în rucsac primul obiect (deoarece are cel mai mare câştig integral) şi nu vom mai putea încărca niciun alt obiect, deci câştigul obţinut va fi de 100 RON. Totuşi, câştigul maxim de 170 RON se obţine încărcând în rucsac ultimele două obiecte! În mod asemănător (de exemplu, modificând câştigurilor obiectelor anterior menţionate în $(100, 9, 8)$ RON) se poate găsi un contraexemplu care să arate faptul că nici criteriul b) nu permite obţinerea unei soluţii optime în orice caz.

Se poate observa faptul că primele două criterii nu conduc întotdeauna la soluţia optimă deoarece ele iau în considerare fie doar câştigurile obiectelor, fie doar greutăţile

lor, deci criteriul corect de selecţie trebuie să le ia în considerare pe ambele. Intuitiv, pentru a obţine un câştig maxim, trebuie să încărcăm mai întâi în rucsac obiectele care sunt cele mai "eficiente", adică au un câştig mare şi o greutate mică. Această "eficienţă" se poate cuantifica prin intermediul *câştigului unitar* al unui obiect, adică prin raportul $u_i = c_i/g_i$.

Algoritmul Greedy pentru rezolvarea variantei fracţionare a problemei rucsacului este următorul:

- sortăm obiectele în ordinea descrescătoare a câştigurilor unitare;
- pentru fiecare obiect testăm dacă încape integral în spaţiul liber din rucsac, iar în caz afirmativ îl încărcăm complet în rucsac, altfel calculăm fracţiunea din el pe care trebuie să o încărcăm astfel încât să umplem complet rucsacul (după încărcarea oricărui obiect, actualizăm spaţiul liber din rucsac şi câştigul total);
- algoritmul se termină fie când am încărcat toate obiectele în rucsac (în cazul în care $g_1 + g_2 + \cdots + g_n \leq G$), fie când nu mai există spaţiu liber în rucsac.

De exemplu, să considerăm un rucsac în care putem să încărcăm maxim $G = 53$ kg şi $n = 7$ obiecte, având greutăţile $g = (10, 5, 18, 20, 8, 40, 20)$ kg şi câştigurile integrale $c = (30, 40, 36, 10, 16, 30, 20)$ RON. Câştigurile unitare ale celor 7 obiecte sunt $u = \left(\frac{30}{10}, \frac{40}{5}, \frac{36}{18}, \frac{10}{20}, \frac{16}{8}, \frac{30}{40}, \frac{20}{20}\right) = (3, 8, 2, 0.5, 2, 0.75, 1)$ RON/kg, deci sortând descrescător obiectele în funcţie de câştigul unitar vom obţine următoarea ordine a lor: $O_2, O_1, O_3, O_5, O_7, O_6, O_4$. Prin aplicarea algoritmului Greedy prezentat anterior asupra acestor date de intrare, vom obţine următoarele rezultate:

Obiectul curent	Fracţiunea încărcată din obiectul curent	Spaţiul liber în rucsac	Câştigul total
—	—	53	0
O_2: $c_2 = 40, g_2 = 5 \leq 53$	1	53 – 5 = 48	0 + 40 = 40
O_1: $c_1 = 30, g_1 = 10 \leq 48$	1	48 – 10 = 38	40 + 30 = 70
O_3: $c_3 = 36, g_3 = 18 \leq 38$	1	38 – 18 = 20	70 + 36 = 106
O_5: $c_5 = 16, g_5 = 8 \leq 20$	1	20 – 8 = 12	106 + 16 = 122
O_7: $c_7 = 20, g_7 = 20 > 12$	12/20 = 0.6	0	122 + 0.6*20 = 134

În concluzie, pentru a obţine un câştig maxim de 134 RON, trebuie să încărcăm integral în rucsac obiectele O_2, O_1, O_3, O_5 şi o fracţiune de $0.6 = \frac{3}{5}$ din obiectul O_7.

Înainte de a demonstra corectitudinea algoritmului prezentat, vom face următoarele observaţii:

- vom considera obiectele $O_1, O_2, \ldots, O_n$ ca fiind sortate descrescător în funcţie de câştigurile lor unitare, respectiv $\frac{c_1}{g_1} \geq \frac{c_2}{g_2} \geq \cdots \geq \frac{c_n}{g_n}$;

- o soluție a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \dots, x_n)$, unde $x_i \in [0,1]$ reprezintă fracțiunea selectată din obiectul O_i;
- o soluție furnizată de algoritmul Greedy va fi un tuplu de forma $X = (1, \dots, 1, x_j, 0, \dots, 0)$ cu n elemente, unde $x_j \in [0,1)$;
- în toate formulele vom considera implicit indicii ca fiind cuprinși între 1 și n;
- câștigul asociat unei soluții a problemei de forma $X = (x_1, x_2, \dots, x_n)$ îl vom nota cu $C(X) = \sum c_i x_i$;
- dacă $g_1 + g_2 + \cdots + g_n \leq G$, atunci soluția vom obține soluția banală $X = (1, \dots, 1)$, care este evident optimă, deci vom considera faptul că $g_1 + g_2 + \cdots + g_n > G$.

Fie $X = (1, \dots, 1, x_j, 0, \dots, 0)$, unde $x_j \in [0,1)$, soluția furnizată de algoritmul Greedy prezentat, deci rucsacul va fi umplut complet (i.e., $\sum g_i x_i = G$). Presupunem că soluția X nu este optimă, deci există o altă soluție optimă $Y = (y_1, \dots, y_{k-1}, y_k, y_{k+1}, \dots, y_n)$ diferită de soluția X, posibil obținută folosind un alt algoritm. Deoarece Y este o soluție optimă, obținem imediat următoarele două relații: $\sum g_i y_i = G$ și câștigul $C(Y) = \sum c_i y_i$ este maxim.

Deoarece $X \neq Y$, rezultă că există un cel mai mic indice k pentru care $x_k \neq y_k$, având următoarele proprietăți:

- $k \leq j$ (deoarece, în caz contrar, am obține $\sum g_i y_i > G$);
- $y_k < x_k$ (pentru $k < j$ este evident deoarece $x_k = 1$, iar dacă $y_j > x_j$ am obține $\sum g_i y_i > G$).

Considerăm acum soluția $Y' = (y_1, \dots, y_{k-1}, x_k, \alpha y_{k+1}, \dots, \alpha y_n)$, unde α este o constantă reală subunitară aleasă astfel încât $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$. Practic, soluția Y' a fost construită din soluția Y, astfel:

- am păstrat primele $k-1$ componente din soluția Y;
- am înlocuit componenta y_k cu componenta x_k;
- deoarece $x_k > y_k$, am micșorat restul componentelor $y_{k+1}, \dots, y_n$ din soluția Y, înmulțindu-le cu o constantă subunitară α aleasă astfel încât rucsacul să rămână încărcat complet: $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$.

Deoarece Y este soluție a problemei, înseamnă că $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = G$. Dar și $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n = G$, deci $g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = g_1 y_1 + \cdots + g_{k-1} y_{k-1} + g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n$, de unde obținem, după reducerea termenilor egali, relația $g_k y_k + g_{k+1} y_{k+1} + \cdots + g_n y_n = g_k x_k + g_{k+1} \alpha y_{k+1} + \cdots + g_n \alpha y_n$, pe care o putem rescrie astfel:

$$g_k(x_k - y_k) = (1 - \alpha)(g_{k+1} y_{k+1} + \cdots + g_n y_n) \quad (1)$$

Comparăm acum câștigurile asociate soluțiilor Y și Y', calculând diferența dintre ele:

$$\begin{aligned} C(Y') - C(Y) &= c_1y_1 + \cdots + c_{k-1}y_{k-1} + c_kx_k + c_{k+1}\alpha y_{k+1} + \cdots + c_n\alpha y_n \\ &- (c_1y_1 + \cdots + c_{k-1}y_{k-1} + c_ky_k + c_{k+1}y_{k+1} + \cdots + c_ny_n) = \\ &= c_k(x_k - y_k) + (\alpha - 1)(c_{k+1}y_{k+1} + \cdots + c_ny_n) = \\ &= \frac{c_k}{g_k}\left[g_k(x_k - y_k) + (\alpha - 1)\left(\frac{g_k}{c_k}c_{k+1}y_{k+1} + \cdots + \frac{g_k}{c_k}c_ny_n\right)\right] \end{aligned}$$

Rescriind ultima relație, obținem:

$$C(Y') - C(Y) = \frac{c_k}{g_k}\left[g_k(x_k - y_k) + (\alpha - 1)\left(\frac{g_kc_{k+1}}{c_k}y_{k+1} + \cdots + \frac{g_kc_n}{c_k}y_n\right)\right] \quad (2)$$

Dar $\frac{g_k}{c_k} \leq \frac{g_i}{c_i}$ pentru orice $i > k$ (deoarece obiectele sunt sortate descrescător în funcție de câștigurile lor unitare, deci $\frac{c_k}{g_k} \geq \frac{c_i}{g_i}$ pentru orice $i > k$), de unde rezultă că $\frac{g_kc_i}{c_k} \leq g_i$ pentru orice $i > k$, deci obținem relațiile:

$$\frac{g_kc_{k+1}}{c_k} \leq g_{k+1}, \ldots, \frac{g_kc_n}{c_k} \leq g_n \quad (3)$$

Aplicând relațiile (3) în relația (2), obținem:

$$C(Y') - C(Y) \geq \frac{c_k}{g_k}[g_k(x_k - y_k) + (\alpha - 1)(g_{k+1}y_{k+1} + \cdots + g_ny_n)] \quad (4)$$

Din relația (1) obținem că $g_k(x_k - y_k) + (\alpha - 1)(g_{k+1}y_{k+1} + \cdots + g_ny_n) = 0$, deci relația (4) devine $C(Y') - C(Y) \geq 0$, de unde rezultă faptul că $C(Y') \geq C(Y)$. Există acum două posibilități:

a) $C(Y') > C(Y)$, ceea ce contrazice optimalitatea soluției Y, așadar presupunerea că ar exista o soluție optimă Y diferită de soluția X furnizată de algoritmul Greedy este falsă, ceea ce înseamnă că $X = Y$, deci și soluția furnizată de algoritmul Greedy este optimă;
b) $C(Y') = C(Y)$, ceea ce înseamnă că putem să reluăm procedeul prezentat anterior înlocuind Y cu Y' până când, după un număr finit de pași, vom obține o contradicție de tipul celei de la punctul a).

În concluzie, după un număr finit de pași, vom transforma soluția optimă Y în soluția X furnizată de algoritmul Greedy, ceea ce înseamnă că și soluția furnizată de algoritmul Greedy este, de asemenea, optimă.

În continuare, vom prezenta implementarea în limbajul Python a algoritmului Greedy pentru rezolvarea variantei fracționare a problemei rucsacului:

```
# functie folosita pentru sortarea descrescătoare a obiectelor
# în raport de câștigul unitar (cheia)
def cheieCâștigUnitar(ob):
    return ob[2] / ob[1]
```

```
# citim datele de intrare din fișierul text "rucsac.in"
fin = open("rucsac.in")
# de pe prima linie citim capacitatea G a rucsacului
G = float(fin.readline())
# fiecare dintre următoarele linii conține
# greutatea și câștigul unui obiect
obiecte = []
crt = 1
for linie in fin:
    aux = linie.split()
    # un obiect este un tuplu (ID, greutate, câștig)
    obiecte.append((crt, float(aux[0]), float(aux[1])))
    crt += 1
fin.close()

# sortăm obiectele descrescător în funcție de câștigul unitar
obiecte.sort(key=cheieCâștigUnitar, reverse=True)
# n reprezintă numărul de obiecte
n = len(obiecte)
# solutie este o listă care va conține fracțiunile încărcate
# din fiecare obiect
soluție = [0] * n
# inițial, spațiul liber din rucsac este chiar G
spațiu_liber_rucsac = G
# considerăm, pe rând, fiecare obiect
for i in range(n):
    # dacă obiectul curent încape complet în spațiul liber
    # din rucsac, atunci îl încărcăm complet
    if obiecte[i][1] <= spațiu_liber_rucsac:
        spațiu_liber_rucsac -= obiecte[i][1]
        soluție[i] = 1
    else:
        # dacă obiectul curent nu încape complet în spațiul liber
        # din rucsac, atunci calculăm fracțiunea din el necesară
        # pentru a încărca complet rucsacul și algoritmul se termină
        soluție[i] = spațiu_liber_rucsac / obiecte[i][1]
        break

# calculăm câștigul maxim
câștig = sum([soluție[i] * obiecte[i][2] for i in range(n)])

# scriem datele de ieșire în fișierul text "rucsac.out"
fout = open("rucsac.out", "w")
fout.write("Castig maxim: " + str(câștig) + "\n")
fout.write("\nObiectele incarcate:\n")
i = 0
while i < n and soluție[i] != 0:
    # trunchiem procentul încărcat din obiectul curent
    # la două zecimale
```

```
    procent = format(soluție[i]*100, '.2f')
    fout.write("Obiect "+str(obiecte[i][0])+": "+procent+"%\n")
    i = i + 1
fout.close()
```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

rucsac.in	**rucsac.out**
53 10 30 5 40 18 36 20 10 8 16 40 30 20 20	Castig maxim: 134.0 Obiectele incarcate: Obiect 2: 100.00% Obiect 1: 100.00% Obiect 3: 100.00% Obiect 5: 100.00% Obiect 7: 60.00%

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, selectarea și încărcarea obiectelor în rucsac are cel mult complexitatea $\mathcal{O}(n)$, iar afișarea câștigului maxim obținut $\mathcal{O}(1)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.

TEHNICA DE PROGRAMARE "DIVIDE ET IMPERA"

1. Prezentare generală

Tehnica de programare *Divide et Impera* constă în descompunerea repetată a unei probleme în două sau mai multe subprobleme de acelaşi tip până când se obţin probleme direct rezolvabile (etapa *Divide*), după care, în sens invers, soluţia fiecărei probleme se obţine combinând soluţiile subproblemelor în care a fost descompusă (etapa *Impera*).

Se poate observa cu uşurinţă faptul că tehnica *Divide et Impera* are în mod nativ un caracter recursiv, însă există şi cazuri în care ea este implementată iterativ.

Evident, pentru ca o problemă să poată fi rezolvată folosind tehnica Divide et Impera, ea trebuie să îndeplinească următoarele două condiţii:

1. condiţia *Divide*: problema poate fi descompusă în două (sau mai multe) subprobleme de acelaşi tip;
2. condiţia *Impera*: soluţia unei probleme se poate obţine combinând soluţiile subproblemelor în care ea a fost descompusă.

De obicei, subproblemele în care se descompune o problemă au dimensiunile datelor de intrare aproximativ egale sau, altfel spus, aproximativ egale cu jumătate din dimensiunea datelor de intrare ale problemei respective.

De exemplu, folosind tehnica Divide et Impera, putem calcula suma elementelor unei liste t formată din n numere întregi, astfel:

1. împărţim lista t, în mod repetat, în două jumătăţi până când obţinem liste cu un singur element (caz în care suma se poate calcula direct, fiind chiar elementul respectiv);
2. în sens invers, calculăm suma elementelor dintr-o listă adunând sumele elementelor celor două sub-liste în care ea a fost descompusă.

Se observă imediat faptul că problema verifică ambele condiţii menţionate mai sus!

Pentru a putea manipula uşor cele două sub-liste care se obţin în momentul împărţirii listei t în două jumătăţi, vom considera faptul că lista curentă este secvenţa cuprinsă între doi indici st şi dr, unde $st \leq dr$. Astfel, indicele mij al mijlocului listei curente este aproximativ egal cu $[(st + dr)/2]$, iar cele două sub-liste în care va fi descompus lista curentă sunt secvenţele cuprinse între indicii st şi mij, respectiv $mij + 1$ şi dr.

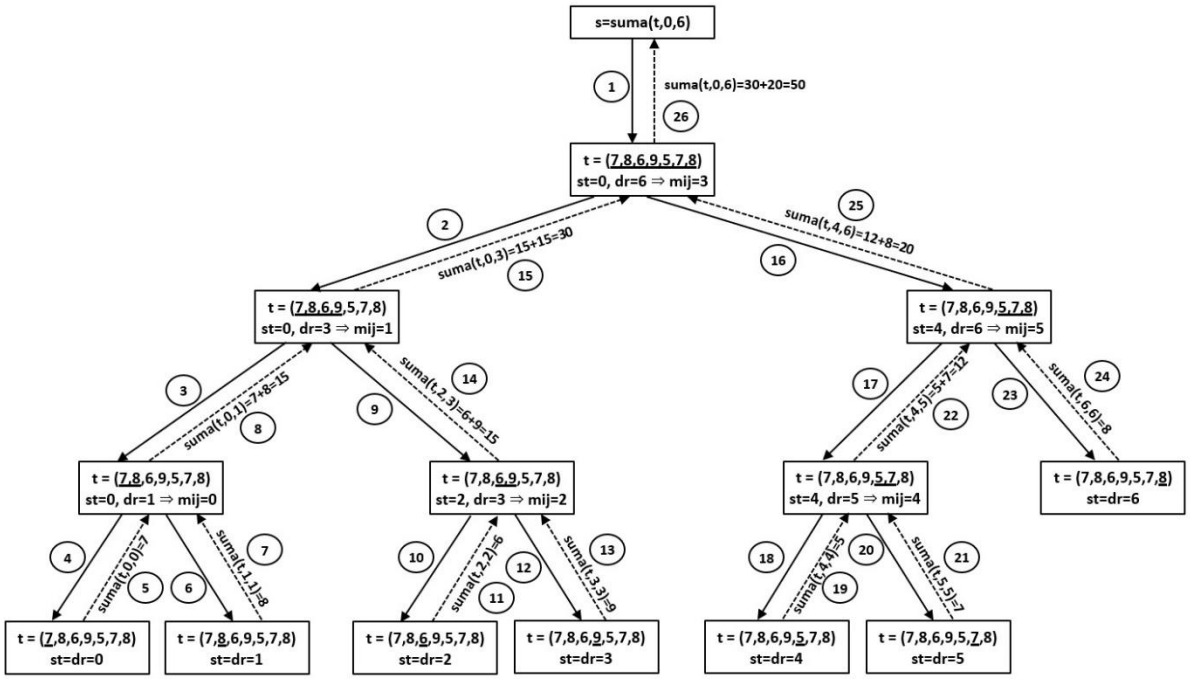
Considerând $suma(t, st, dr)$ o funcţie care calculează suma secvenţei $t[st], t[st + 1], \dots, t[dr]$, putem să o definim în manieră Divide et Impera astfel:

$$suma(t, st, dr) = \begin{cases} t[st], & \text{dacă } st = dr \quad (1) \\ suma(t, st, mij) + suma(t, mij + 1, dr), & \text{dacă } st < dr \quad (2) \end{cases}$$

unde $mij = [(st + dr)/2]$.

Pentru o listă t cu n elemente, suma s a tuturor elementelor sale se va obţine în urma apelului $s = suma(t, 0, n - 1)$.

De exemplu, considerând lista $t = (7, 8, 6, 9, 5, 7, 8)$ cu $n = 7$ elemente, pentru a calcula suma elementelor sale, se vor efectua următoarele apeluri recursive:



În figura de mai sus, săgeţile "pline" reprezintă apelurile recursive, efectuate folosind relaţia (2) de mai sus, în timp ce săgeţile "întrerupte" reprezintă revenirile din apelurile recursive, care au loc în momentul în care se ajunge la o subproblemă direct rezolvabilă folosind relaţia (1) de mai sus. Ordinea în care se execută apelurile şi revenirile este indicată prin numerele scrise în cercuri. Numerele subliniate din lista *t* reprezintă secvenţa curentă (i.e., care se prelucrează în apelul respectiv).

2. Forma generală a unui algoritm de tip Divide et Impera

Plecând chiar de la exemplul se mai sus, se poate deduce foarte uşor forma generală a unui algoritm de tip Divide et Impera aplicat asupra unei liste *t*:

```
# functie care furnizeaza solutia unei probleme combinand solutiile
# subproblemelor in care ea a fost descompusa
def combinare(sol_st, sol_dr):
    pass

def divimp(t, st, dr):
   # daca subproblema curentă este direct rezolvabilă
   if dr-st <= k:          # k este, de obicei, 0 sau 1
      return solutie_problema_directa

   # etapa Divide
   mij = (st + dr) // 2
   sol_st = divimp(t, st, mij)
   sol_dr = divimp(t, mij+1, dr)

   # etapa Impera
   return combinare(sol_st, sol_dr)
```

Vizavi de algoritmul general prezentat mai sus trebuie făcute câteva precizări:

- există algoritmi Divide et Impera în care nu sunt utilizate liste (de exemplu, calculul lui a^n - https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/), dar aceştia sunt mult mai rari decât cei în care sunt utilizate liste;
- de obicei, o subproblemă este direct rezolvabilă dacă secvenţa curentă din lista `t` este vidă (i.e., `st>dr`) sau are un singur element (i.e., `st==dr`);
- variabila `mij` va conţine indicele mijlocului secvenţei `t[st], t[st+1], …, t[dr]`;
- variabilele `sol_st` şi `sol_dr` vor conţine soluţiile celor două subprobleme în care se descompune problema curentă, iar `combinare(sol_st, sol_dr)` este o funcţie care determină soluţia problemei curente combinând soluţiile subproblemelor în care aceasta a fost descompusă (în unele cazuri, nu este necesară o funcţie, ci se poate folosi o simplă expresie);
- dacă funcţia `divimp` nu furnizează nicio valoare, atunci vor lipsi variabilele `sol_st` şi `sol_dr`, precum şi cele două instrucţiuni `return`.

Aplicând algoritmul general pentru a calcula suma elementelor dintr-o listă de numere întregi, vom obţine următoarea implementare în limbajul Python:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Aşa cum am menţionat în observaţiile anterioare, nu a mai fost necesară implementarea unei funcţii `combinare`, ci a fost suficientă utilizarea unei simple expresii.

3. Determinarea complexităţii unui algoritm de tip Divide et Impera

Deoarece algoritmii de tip Divide et Impera sunt implementaţi, de obicei, folosind funcţii recursive, determinarea complexităţii computaţionale a unui astfel de algoritm este mai complicată decât în cazul algoritmilor iterativi.

Primul pas în determinarea complexităţii unei algoritm Divide et Impera îl constituie determinarea unei relaţii de recurenţă care să exprime complexitatea $T(n)$ a rezolvării unei probleme având dimensiunea datelor de intrare egală cu n în raport de timpul necesar rezolvării subproblemelor în care aceasta este descompusă şi de complexitatea operaţiei de combinare a soluţiilor lor pentru a obţine soluţia problemei iniţiale. Presupunând faptul că orice problemă se descompune în a subprobleme, fiecare având dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{b}$, iar împărţirea problemei curente în subprobleme şi combinarea soluţiilor subproblemelor pentru a obţine soluţia sa se

realizează folosind un algoritm cu complexitatea $f(n)$, se obține foarte ușor forma generală a relației de recurență căutate:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{3}$$

unde $a \geq 1$, $b > 1$ și $f(n)$ este o funcție asimptotic pozitivă (i.e., există $n_0 \in \mathbb{N}$ astfel încât pentru orice $n \geq n_0$ avem $f(n) \geq 0$). De asemenea, vom presupune faptul că $T(1) \in \mathcal{O}(1)$.

Reluăm algoritmul Divide et Impera prezentat mai sus pentru calculul sumei elementelor unei liste, cu scopul de a-i determina complexitatea:

```
def suma(t, st, dr):
    # daca subproblema curentă este direct rezolvabilă,
    # respectiv lista curenta are un singur element
    if dr == st:
        return t[st]

    # etapa Divide
    mij = (st + dr) // 2
    sol_st = suma(t, st, mij)
    sol_dr = suma(t, mij + 1, dr)

    # etapa Impera
    return sol_st + sol_dr
```

Analizând fiecare etapa a algoritmului, obținem următoarea relație de recurență pentru $T(n)$:

$$T(n) = \begin{cases} 1 + T\left(\left[\frac{n}{2}\right]\right) + T\left(\left[\frac{n}{2}\right]\right) + 1, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases} = \begin{cases} 2T\left(\left[\frac{n}{2}\right]\right) + 2, & \text{dacă } n \geq 2 \\ 1, & \text{dacă } n = 1 \end{cases}$$

În concluzie, o problemă având dimensiunea datelor de intrare egală cu n se descompune în $a = 2$ subprobleme, fiecare având dimensiunea datelor de intrare aproximativ egală cu $\frac{n}{2}$ (deci $b = 2$), iar împărțirea problemei curente în subprobleme și combinarea soluțiilor subproblemelor pentru a obține soluția sa se realizează folosind un algoritm cu complexitatea $\mathcal{O}(1)$, deci relația de recurență este următoarea:

$$T(n) = 2T\left(\frac{n}{2}\right) + 1 \tag{4}$$

Al doilea pas în determinarea complexității unei algoritm Divide et Impera îl constituie rezolvarea relației de recurență de mai sus, utilizând diverse metode matematice, pentru a determina expresia analitică a lui $T(n)$. În continuare, vom prezenta două dintre cele mai utilizate metode, simplificate, respectiv *iterarea directă a relației de recurență* și *teorema master*.

În cazul primei metode, bazată pe *iterarea directă a relației de recurență*, vom presupune faptul că n este o putere a lui b, după care vom itera relația de recurență până

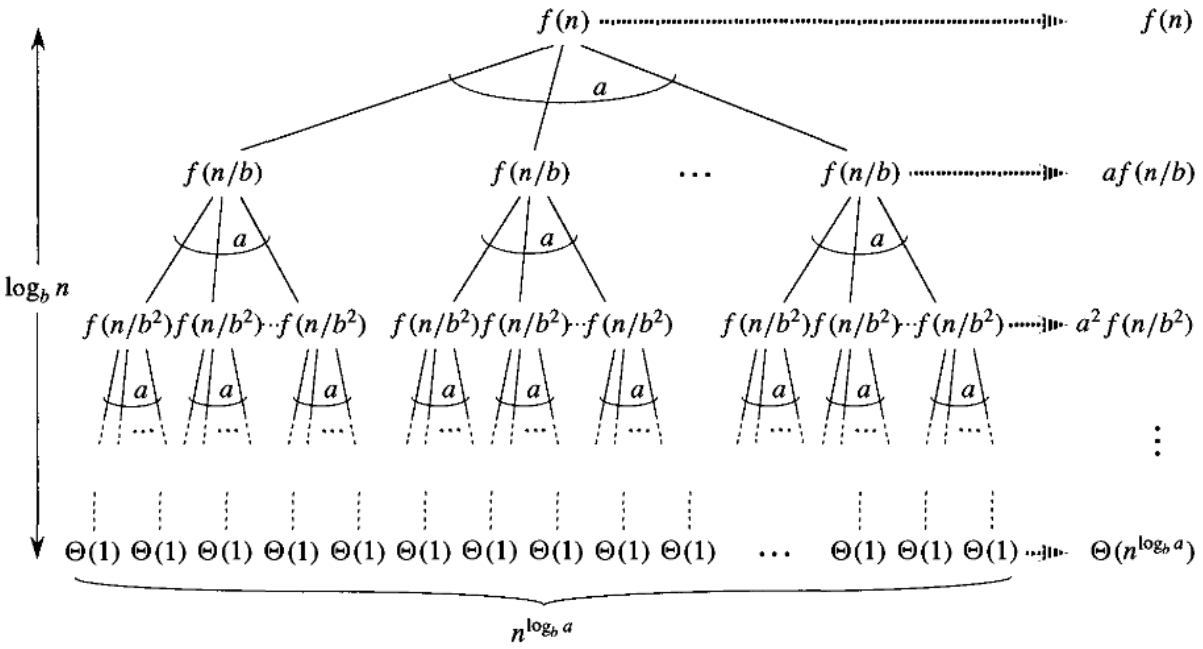
când vom ajunge la $T(1)$ sau $T(0)$, care sunt ambele egale cu 1 fiind complexităţile unor probleme direct rezolvabile.

De exemplu, pentru a rezolva relaţia de recurenţă (4), vom presupune că $n = 2^k$ şi apoi o vom itera, astfel:

$$\begin{aligned} T(n) = T(2^k) &= 2T(2^{k-1}) + 1 = 2[2T(2^{k-2}) + 1] + 1 = 2^2T(2^{k-2}) + 2 + 1 = \\ &= 2^2[2T(2^{k-3}) + 1] + 2 + 1 = 2^3T(2^{k-3}) + 2^2 + 2 + 1 = \cdots = \\ &= 2^kT(2^0) + 2^{k-1} + \cdots + 2 + 1 = 2^k + 2^{k-1} + \cdots + 2 + 1 = 2^{k+1} - 1 = \\ &= 2 \cdot 2^k - 1 = 2n - 1 \end{aligned}$$

În concluzie, am obţinut faptul că $T(n) = 2n - 1$, deci complexitatea algoritmului Divide et Impera pentru calculul sumei elementelor unei liste formate din n numere întregi este $\mathcal{O}(2n - 1) \approx \mathcal{O}(n)$.

A doua metodă constă în utilizarea *teoremei master* pentru a afla direct soluţia analitică a unei relaţii de recurenţă de tipul (3): $T(n) = aT\left(\frac{n}{b}\right) + f(n)$. Arborele de recursie asociat unei relaţii de recurenţă de acest tip este următorul (sursa imaginii: https://en.wikipedia.org/wiki/Introduction_to_Algorithms):



Se observă faptul că arborele este unul perfect (i.e., orice nod interior are exact 2 fii şi frunzele se află toate pe acelaşi nivel) cu înălţimea $h = \log_b n$, deci pe fiecare nivel, mai puţin pe ultimul, se găsesc $a^i f\left(\frac{n}{b^i}\right)$ noduri interne, iar pe ultimul nivel se găsesc $a^h = a^{\log_b n} = n^{\log_b a}$ frunze. Evident, complexitatea totală $T(n)$ se obţine însumând complexităţile asociate tuturor nodurilor:

$$T(n) = \underbrace{\sum_{i=0}^{\log_b n - 1} \left[a^i \cdot f\left(\frac{n}{b^i}\right)\right]}_{\substack{\text{timpul necesar pentru} \\ \text{divizarea problemei} \\ \text{şi reconstituirea soluţiei}}} + \underbrace{\mathcal{O}\left(n^{\log_b a}\right)}_{\substack{\text{timpul necesar pentru} \\ \text{rezolvarea subproblemelor} \\ \text{directe}}}$$

Pentru anumite forme particulare ale funcției f, se poate calcula suma din expresia lui $T(n)$ și, implicit, se poate determina forma sa analitică.

Teorema master: Fie o relație de recurență de forma (3) și presupunem faptul că $f \in \mathcal{O}(n^p)$. Atunci:

a) dacă $p < \log_b a$, atunci $T(n) \in \mathcal{O}(n^{\log_b a})$;
b) dacă $p = \log_b a$, atunci $T(n) \in \mathcal{O}(n^p \log_2 n)$;
c) dacă $p > \log_b a$ și $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare, atunci $T(n) \in \mathcal{O}(f(n))$.

Exemple:

1. Pentru a rezolva relația de recurență (4), observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(1) = \mathcal{O}(n^0) \Rightarrow p = 0 < \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } a)} T(n) \in \mathcal{O}(n)$.

2. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n) \Rightarrow p = 1 = \log_b a = \log_2 2 = 1 \xrightarrow{\text{Cazul } b)} T(n) \in \mathcal{O}(n \log_2 n)$.

3. Pentru a rezolva relația de recurență $T(n) = 2T\left(\frac{n}{2}\right) + n^2$, observăm faptul că $a = b = 2$ și $f \in \mathcal{O}(n^2) \Rightarrow p = 2 > \log_b a = \log_2 2 = 1$. În acest caz, trebuie să verificăm și faptul că $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare $\Leftrightarrow \exists c < 1$ astfel încât $2\frac{n^2}{4} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $\frac{n^2}{2} \leq cn^2 \Leftrightarrow \exists c < 1$ astfel încât $n^2 \leq 2cn^2 \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat, de exemplu pentru $c = \frac{1}{2}$ sau, în general, pentru orice $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}(n^2)$.

Varianta prezentată mai sus a teoremei master este una simplificată, dar care acoperă majoritatea cazurilor întâlnite în practică. O variantă extinsă a acestei teoreme este prezentată aici: https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms). Totuși, nici varianta extinsă nu acoperă toate cazurile posibile. Mai mult, teorema de master nu poate fi utilizată dacă subproblemele nu au dimensiuni aproximativ egale, fiind necesară utilizarea teoremei Akra-Bazzi (https://en.wikipedia.org/wiki/Akra-Bazzi_method).

Observație importantă: În general, algoritmii de tip Divide et Impera au complexități mici, de tipul $\mathcal{O}(\log_2 n), \mathcal{O}(n)$ sau $\mathcal{O}(n \log_2 n)$, care se obțin datorită faptului că o problemă este împărțită în două subprobleme de același tip cu dimensiunea datelor de intrare înjumătățită față de problema inițială și, mai mult, subproblemele nu se suprapun! Dacă aceste condiții nu sunt îndeplinite simultan, atunci complexitatea algoritmul poate să devină foarte mare, de ordin exponențial! De exemplu, o implementare recursivă, de tip Divide et Impera, care să calculeze termenul de rang n al șirului lui Fibonacci ($F_n = F_{n-1} + F_{n-2}$ și $F_0 = 0, F_1 = 1$) nu respectă condițiile precizate anterior (dimensiunile subproblemelor nu sunt aproximativ jumătate din dimensiunea unei probleme și subproblemele se suprapun, respectiv mulți termeni vor fi calculați de mai multe ori), ceea ce va conduce la o complexitate exponențială! Astfel, relația de recurență pentru

complexitatea algoritmului este $T(n) = 1 + T(n-1) + T(n-2)$. Cum $T(n-1) > T(n-2)$, obţinem că $2T(n-2) < T(n) < 2T(n-1)$. Iterând dubla inegalitate, obţinem $2^{\frac{n}{2}} < T(n) < 2^n$, ceea ce dovedeşte faptul că implementarea recursivă are o complexitate exponenţială. Totuşi, există mai multe metode iterative cu complexitate liniară pentru rezolvarea acestei probleme: https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/.

În continuare, vom prezenta câteva probleme clasice care se pot rezolva utilizând tehnica de programare Divide et Impera.

4. Problema căutării binare

Fie t o listă formată din n numere întregi sortate crescător şi x un număr întreg. Să se verifice dacă valoarea x apare în lista t.

Evident, problema ar putea fi rezolvată printr-o simplă parcurgere a listei t (*căutare liniară*), obţinând un algoritm având complexitatea $\mathcal{O}(n)$, dar nu am utiliza deloc faptul că elementele listei sunt în ordine crescătoare. Pentru a efectua o căutare binară într-o secvenţă $t[st], t[st+1], \dots, t[dr]$ a listei t în care $st \leq dr$ vom folosi această ipoteză, comparând valoarea căutată x cu valoarea $t[mij]$ aflată în mijlocul secvenţei. Astfel, vom obţine următoarele 3 cazuri:

a) $x < t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[st], \dots, t[mij-1]$;
b) $x > t[mij] \Longrightarrow$ vom căuta valoarea x doar în secvenţa $t[mij+1], \dots, t[dr]$;
c) $x = t[mij] \Longrightarrow$ am găsit valoarea x, deci operaţia de căutare se încheie cu succes.

Dacă la un moment dat $st > dr$, înseamnă că nu mai există nicio secvenţă $t[st],, \dots, t[dr]$ în care să aibă sens să căutăm valoarea x, deci operaţia de căutare eşuează.

O implementare a căutării binare în limbajul Python, sub forma unei funcţii care furnizează o poziţie pe care apare valoarea x în lista t sau valoarea -1 dacă x nu apare deloc în listă, este următoarea:

```
def cautare_binara(t, x, st, dr):
    if st > dr:
        return -1

    mij = (st+dr) // 2
    if x == t[mij]:
        return mij
    elif x < t[mij]:
        return cautare_binara(t, x, st, mij-1)
    else:
        return cautare_binara(t, x, mij+1, dr)
```

Se observă faptul că acest algoritm de tip Divide et Impera constă doar din etapa Divide, nemaifiind combinate soluţiilor subproblemelor (etapa Impera). Practic, la fiecare pas, problema curentă se restrânge la una dintre cele două subprobleme, ci nu se rezolvă ambele subprobleme! Astfel, ţinând cont de faptul că etapa Divide are complexitatea

$\mathcal{O}(1)$, relaţia de recurenţă asociată complexităţii algoritmului de căutare binară este următoarea:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Folosind atât iterarea directă a relaţiei de recurenţă, cât şi teorema master, demonstraţi faptul că $T(n) \in \mathcal{O}(\log_2 n)$!

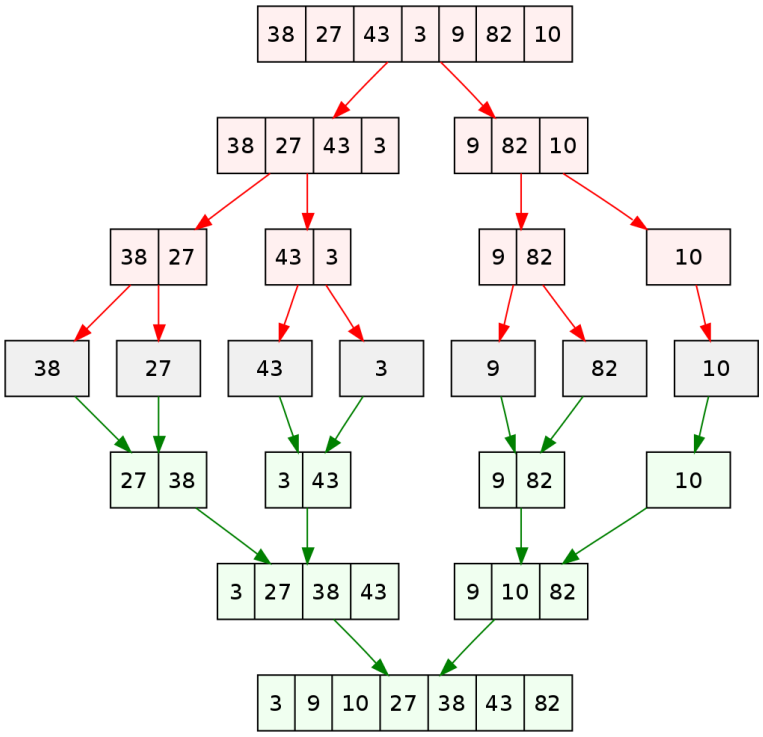
Observaţie importantă: Complexitatea $\mathcal{O}(\log_2 n)$ reprezintă strict complexitatea algoritmului de căutare binară, deci complexitatea unui program, chiar foarte simplu, în care se va utiliza acest algoritm va fi mai mare! De exemplu, un simplu program de test pentru funcţia de mai sus necesită citirea celor n elemente ale listei t şi afişarea valorii furnizate de funcţie, deci complexitatea sa va fi $\mathcal{O}(n) + \mathcal{O}(\log_2 n) + \mathcal{O}(1) \approx \mathcal{O}(n)$!

5. Sortarea prin interclasare (Mergesort)

Sortarea prin interclasare utilizează tehnica de programare Divide et Impera pentru a sorta crescător o listă t, astfel:

- se împarte secvenţa curentă $t[st], \dots, t[dr]$, în mod repetat, în două secvenţe $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$ până când se ajunge la secvenţe implicit sortate, adică secvenţe de lungime 1;
- în sens invers, se sortează secvenţa $t[st], \dots, t[dr]$ interclasând cele două secvenţe în care a fost descompusă, respectiv $t[st], \dots, t[mij]$ şi $t[mij + 1], \dots, t[dr]$, şi care au fost deja sortate la un pas anterior.

O reprezentară grafică a modului în care rulează această metodă de sortare se poate observa în următoarea imagine (sursa: https://en.wikipedia.org/wiki/Merge_algorithm):



Începem prezentarea detaliată a acestei metodei de sortare reamintind faptul că interclasarea este un algoritm care permite obținerea unei liste sortate crescător din două liste care sunt, de asemenea, sortate crescător. Considerând dimensiunile listelor care vor fi interclasate ca fiind egale cu m și n, complexitatea algoritmului de interclasare este $\mathcal{O}(m+n)$.

În cazul sortării prin interclasare, se vor interclasa secvențele $t[st], \dots, t[mij]$ și $t[mij+1], \dots, t[dr]$ într-o listă aux de lungime $dr-st+1$, iar la sfârșit elementele acesteia se vor copia în secvența $t[st], \dots, t[dr]$:

```
def interclasare(t, st, mij, dr):
    i = st
    j = mij+1
    aux = []
    while i <= mij and j <= dr:
        if t[i] <= t[j]:
            aux.append(t[i])
            i += 1
        else:
            aux.append(t[j])
            j += 1

    aux.extend(t[i:mij+1])
    aux.extend(t[j:dr+1])

    t[st:dr+1] = aux[:]
```

Se observă ușor faptul că funcția `interclasare` are o complexitate egală cu $\mathcal{O}(dr-st+1) \leq \mathcal{O}(n)$.

Sortarea listei t se va efectua, folosind tehnica Divide et Impera, în cadrul următoarei funcții:

```
def mergesort(t, st, dr):
    if st < dr:
        mij = (dr+st) // 2
        mergesort(t, st, mij)
        mergesort(t, mij+1, dr)
        interclasare(t, st, mij, dr)
```

Observați faptul că, în acest caz, funcția `interclasare` are rolul funcției `combinare` din algoritmul generic Divide et Impera!

Complexitatea funcției `mergesort` și, implicit, complexitatea sortării prin interclasare, se obține rezolvând următoarea relație de recurență:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

În exemplul 2 de la teorema master am demonstrat faptul că $T(n) \in \mathcal{O}(n \log_2 n)$, deci sortarea prin interclasare are complexitatea $\mathcal{O}(n \log_2 n)$.

6. Selecția celui de-al k-lea minim (Quickselect)

Considerăm o listă L de lungime n și dorim să determinăm al k-lea minim al său (evident, $1 \leq k \leq n$), respectiv valoarea care s-ar afla pe poziția $k-1$ după sortarea crescătoare a sa. De exemplu, pentru $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$ și $k = 5$ vom obține valoarea 7, deoarece lista A sortată crescător este $[3, 4, 4, 7, 7, 9, 10, 12, 25]$. Evident, problema poate fi rezolvată sortând lista și apoi accesând elementul aflat pe poziția $k-1$, complexitatea acestei soluții fiind $\mathcal{O}(n \log_2 n)$.

În continuare, vom prezenta un algoritm de tip Divide et Impera pentru rezolvarea acestei probleme:

- alegem aleatoriu un element din listă ca pivot;
- partiționăm lista A în 3 liste în raport de pivotul respectiv:
  - o listă L formată din elementele strict mai mici decât pivotul;
  - o listă E formată din elementele egale cu pivotul;
  - o listă G formată din elementele strict mai mari decât pivotul;
- comparăm valoarea k cu lungimile celor 3 liste de partiție și fie furnizăm valoarea căutată (dacă a k-a valoare se găsește în lista E), fie reluăm procedeul pentru una dintre listele L sau G.

De exemplu, considerând $A = [10, 7, 25, 4, 3, 4, 9, 12, 7]$, $k = 5$ și pivotul aleatoriu 9, vom obține următoarele mulțimi de partiție: $L = [7, 4, 3, 4, 7]$, $E = [9]$ și $G = [10, 25, 12]$. Deoarece $k = 5 \leq \text{len}(L)$, vom relua procedeul pentru lista L și aceeași valoare $k = 5$. Dacă am fi considerat $k = 8$, atunci am fi reluat procedeul pentru lista G și $k = 8 - len(L) - len(E) = 8 - 5 - 1 = 2$!

Implementarea în limbajul Python a acestui algoritm de tip Divide et Impera este următoarea:

```
import random

def quickselect(A, k, f_pivot=random.choice):
    pivot = f_pivot(A)

    L = [x for x in A if x < pivot]
    E = [x for x in A if x == pivot]
    G = [x for x in A if x > pivot]

    if k < len(L):
        return quickselect(L, k, f_pivot)
    elif k < len(L) + len(E):
        return E[0]
    else:
        return quickselect(G, k - len(L) - len(E), f_pivot)
```

Deoarece pivotul poate fi selectat și în alte moduri (ci nu numai aleatoriu), am folosit mecanismul de call-back în cadrul funcției `quickselect` astfel încât ea să poată utiliza o anumită funcție pentru selectarea pivotului, primită prin intermediul parametrului `f_pivot`. Implicit, acest parametru este inițializat cu funcția `choice` din pachetul `random`, care furnizează în mod aleatoriu o valoare dintr-o colecție dată ca parametru. Atenție, funcția trebuie apelată prin `quickselect(A, k-1)`!

Deoarece pivotul este ales în mod aleatoriu, estimarea complexității algoritmului `quickselect` prezentat anterior este destul de complicată. Totuși, se poate observa ușor faptul că un pivot "bun" ar trebui să genereze două liste de partiție L și G cu dimensiuni aproximativ egale cu jumătate din numărul de elemente ale listei inițiale A. În acest caz, complexitatea algoritmului `quickselect` ar fi dată de următoarea relație de recurență:

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\substack{\text{se alege una dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ și } G}}$$

Pentru a rezolva această relație de recurență folosind teorema master, observăm faptul că $a = 1$ și $b = 2$, iar $f(n) = n \in \mathcal{O}(n^1) \Rightarrow p = 1 > \log_b a = \log_2 1 = 0$. Deoarece suntem în cazul c) al teoremei master, trebuie să verificăm și faptul că $\exists c < 1$ astfel încât $af\left(\frac{n}{b}\right) \leq cf(n)$ pentru orice n suficient de mare $\Leftrightarrow \exists c < 1$ astfel încât $\frac{n}{2} \leq cn \Leftrightarrow \exists c < 1$ astfel încât $n \leq 2cn \Leftrightarrow \exists c < 1$ astfel încât $1 \leq 2c$, ceea ce este adevărat pentru orice valoare $c \in \left[\frac{1}{2}, 1\right)$. Astfel, obținem că $T(n) \in \mathcal{O}\big(f(n)\big) = \mathcal{O}(n)$. Pe de altă parte, selectarea repetată ca pivot a valorii minime/maxime din listă va conduce la complexitatea $\mathcal{O}(n^2)$!

Practic, un pivot "bun" ar trebui să fie *mediana listei* respective, adică valoarea aflată la mijlocul listei sortate crescător, sau, cel puțin, o valoare apropiată de aceasta. În continuare, vom prezenta algoritmul *mediana medianelor*, tot de tip Divide et Impera, care permite determinarea unui pivot "bun" pentru o listă. Algoritmul a fost creat în anul 1973 de către informaticienii M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest și R. Tarjan, din acest motiv fiind cunoscut și sub numele de *algoritmul BFPRT*. Pașii algoritmului, pentru o listă A formată din n elemente, sunt următorii:

- se împarte lista A în $\left[\frac{n}{5}\right]$ liste de lungime 5 (dacă ultima listă nu are exact 5 elemente, atunci ea poate fi eliminată);
- pentru fiecare dintre cele $\left[\frac{n}{5}\right]$ liste de lungime 5 se determină direct mediana sa (e.g., se sortează lista și se returnează elementul aflat în mijlocul său);
- se reia procedeul pentru lista formată din medianele celor $\left[\frac{n}{5}\right]$ liste de lungime 5 până când se obține mediana medianelor (i.e., lista medianelor are cel mult 5 elemente).

Exemplu: Considerăm lista $A = [3, 14, 10, 2, 15, 10, 5, 51, 15, 20, 40, 4, 18, 13, 8, 40, 21, 61, 19, 50, 12, 35, 8, 7, 22, \sout{100, 17}]$ cu $n = 27$ elemente. După sortarea fiecărei liste de lungime 5 (am eliminat lista [100,17], deoarece este formată doar din două elemente) și determinarea medianelor lor, reluăm procedeul pentru lista medianelor, respectiv [10, 15, 13, 40, 12], și obținem pivotul 13:

	2	5	4	19	7
	3	10	8	21	8
Mediane	10	15	13	40	12
	14	20	18	50	22
	15	51	40	61	35

Implementarea acestui algoritm în limbajul Python este foarte simplă:

```
def BFPRT(A):
    if len(A) <= 5:
        return sorted(A)[len(A) // 2]

    grupuri = [sorted(A[i:i + 5]) for i in range(0, len(A), 5)]
    mediane = [grup[len(grup) // 2] for grup in grupuri]

    return BFPRT(mediane)
```

Rearanjând listele de lungime 5 în ordinea crescătoare a medianelor lor, obţinem:

Mediane

2	7	4	5	19
3	8	8	10	21
10	12	13	15	40
14	22	18	20	50
15	35	40	51	61

Se observă faptul că elementele marcate cu galben sunt mai mici decât pivotul 13, iar elementele marcate cu verde sunt mai mari decât pivotul. De asemenea, se observă faptul că aproximativ jumătate dintre medianele grupurilor sunt mai mici decât pivotul şi aproximativ jumătate sunt mai mari decât el. Astfel, rezultă faptul că fiecare dintre cele două grupuri va avea cel puţin $\frac{3n}{10}$ elemente, deoarece în fiecare grup intră aproximativ jumătate din numărul de $\left\lceil \frac{n}{5} \right\rceil$ liste de lungime 5, deci $\left\lceil \frac{n}{10} \right\rceil$ liste, iar în fiecare listă sunt 3 elemente. În acelaşi timp, fiecare dintre cele două grupuri nu poate avea mai mult de $n - \frac{3n}{10} = \frac{7n}{10}$ elemente, deci oricare dintre cele două grupuri va avea un număr de elemente cuprins între $\frac{3n}{10}$ şi $\frac{7n}{10}$.

Practic, elementele marcate cu galben sunt cele din lista L definită în algoritmul `quickselect`, iar cele marcate cu verde sunt cele din lista G, deci utilizând algoritmul `BFPRT` pentru selectarea pivotului, prin apelul `quickselect(A, k-1, BFPRT)`, vom obţine următoarea complexitate a sa:

$$T(n) \leq \underbrace{T\left(\frac{n}{5}\right)}_{\substack{\text{determinarea pivotului} \\ \text{folosind mediana medianelor}}} + \underbrace{T\left(\frac{7n}{10}\right)}_{\substack{\text{selectarea uneia dintre} \\ \text{listele } L \text{ sau } G}} + \underbrace{n}_{\substack{\text{crearea listelor} \\ L,E \text{ şi } G}}$$

Pentru rezolvarea acestei relaţii de recurenţă nu putem utiliza teorema master, deoarece dimensiunile subproblemelor nu sunt egale, dar se poate intui faptul că $T(n) \leq cn$ şi apoi demonstra acest lucru folosind inducţia matematică (https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec01.pdf), deci $T(n) \in \mathcal{O}(n)$. Cu toate acestea, deoarece valoarea constantei c pentru care se obţine o complexitate liniară este mare, respectiv $c = 140$, în practică se preferă utilizarea variantei cu pivot ales aleatoriu deoarece are o complexitate medie mai bună!

TEHNICA PROGRAMĂRII DINAMICE

1. Prezentare generală

Programarea dinamică este o tehnică de programare utilizată, de obicei, tot pentru rezolvarea problemelor de optimizare. Programarea dinamică a fost inventată de către matematicianul american Richard Bellman în anii '50 în scopul optimizării unor probleme de planificare, deci cuvântul *programare* din denumirea acestei tehnici are, de fapt, semnificaţia de *planificare*, ci nu semnificaţia din informatică! Practic, tehnica programării dinamice poate fi utilizată pentru planificarea optimă a unor activităţi, iar decizia de a planifica sau nu o anumită activitate se va lua *dinamic*, ţinând cont de activităţile planificate până în momentul respectiv. Astfel, se observă faptul că tehnica programării dinamice diferă de tehnica Greedy (care este utilizată tot pentru rezolvarea problemelor de optim), în care decizia de a planifica sau nu o anumită activitate se ia într-un mod static, fără a ţine cont de activităţile planificate anterior, ci doar verificând dacă activitatea curentă îndeplineşte un anumit criteriu predefinit. Totuşi, cele două tehnici de programare se aseamănă prin faptul că ambele determină o singură soluţie a problemei, chiar dacă există mai multe.

În general, tehnica programării dinamice se poate utiliza pentru rezolvarea problemelor de optim care îndeplinesc următoarele două condiţii:

1. *condiţia de substructură optimală*: problema dată se poate descompune în subprobleme de acelaşi tip, iar soluţia sa optimă (optimul global) se obţine combinând soluţiile optime ale subproblemelor în care a fost descompusă (optime locale);
2. *condiţia de superpozabilitate*: subproblemele în care se descompune problema dată se suprapun.

Se poate observa faptul că prima condiţie este o combinaţie dintre o condiţie specifică tehnicii de programare *Divide et Impera* (problema dată se descompune în subprobleme de acelaşi tip) şi o condiţie specifică tehnicii *Greedy* (optimul global se obţine din optimele locale). Totuşi, o rezolvare a acestui tip de problemă folosind tehnica *Divide et Impera* ar fi ineficientă, deoarece subproblemele se suprapun (a doua condiţie), deci aceeaşi subproblemă ar fi rezolvată de mai multe ori, ceea ce ar conduce la un timp de executare foarte mare (de obicei, chiar exponenţial)!

Pentru a evita rezolvarea repetată a unei subprobleme, se va utiliza *tehnica memoizării*: fiecare subproblemă va fi rezolvată o singură dată, iar soluţia sa va fi păstrată într-o structură de date corespunzătoare, de obicei, unidimensională sau bidimensională.

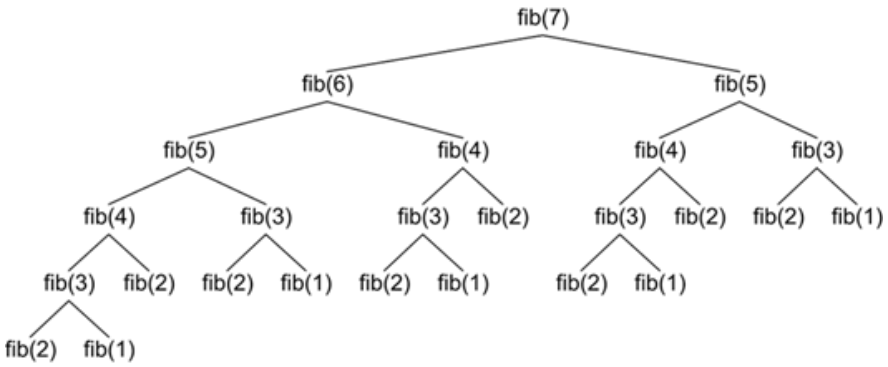
De exemplu, să considerăm şirul lui Fibonacci, definit recurent astfel:

$$f_n = \begin{cases} 0, & \text{dacă } n = 1 \\ 1, & \text{dacă } n = 2 \\ f_{n-2} + f_{n-1}, & \text{dacă } n \geq 3 \end{cases}$$

O implementare directă a relației de recurență de mai sus pentru a calcula termenul f_n se poate realiza utilizând o funcție recursivă:

```
def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n-2) + fib(n-1)
```

Pentru a calcula termenul f_7 vom apela funcția prin `fib(7)` și vom obține următorul arbore de apeluri recursive (sursa imaginii: https://medium.com/@shmuel.lotman/the-2-00-am-javascript-blog-about-memoization-41347e8fa603):



Se observă faptul că anumiți termeni ai șirului se calculează în mod repetat (de exemplu, termenul $f_3 =$ `fib(3)` se va calcula de 5 ori), ceea ce va conduce la o complexitate exponențială (am demonstrat acest fapt în capitolul dedicat tehnicii Divide et Impera).

Pentru a evita calcularea repetată a unor termeni ai șirului, vom folosi tehnica memoizării: vom utiliza o listă `f` pentru a memora termenii șirului, iar fiecare termen nou `f[i]` va fi calculat ca sumă a celor doi termeni precedenți, respectiv `f[i-2]` și `f[i-1]`:

```
def fib(n):
    f = [-1, 0, 1]
    for i in range(3, n+1):
        f.append(f[i-2] + f[i-1])
    return f[n]
```

Se observă faptul că lista `f` este completată într-o maniera ascendentă (*bottom-up*), respectiv se începe cu subproblemele direct rezolvabile (cazurile $n = 0$ și $n = 1$) și apoi se calculează restul termenilor șirului, până la valoarea dorită `f[n]`. Practic, putem afirma faptul că se efectuează doar etapa Impera din rezolvarea de tip Divide et Impera!

În concluzie, rezolvarea unei probleme utilizând tehnica programării dinamice necesită parcurgerea următoarelor etape:

1) se identifică subproblemele problemei date și se determină o relație de recurență care să furnizeze soluția optimă a problemei în funcție de soluțiile optime ale subproblemelor sale (se utilizează substructura optimală a problemei);
2) se identifică o structură de date capabilă să rețină soluțiile subproblemelor;