Metoda greedy

Capitolul

2

- **Prezentarea metodei**
- **Implementări sugerate**
- **Probleme propuse**
- **Soluțiile problemelor**

2.1. Metoda greedy

Metoda rezolvă probleme de optim în care soluția se construieşte pe parcurs. Optimul global se determină prin estimări succesive ale optimului local. Dintr-o mulțime de elemente A trebuie determinată o submulțime B, care verifică anumite ***condiții*** şi care, de obicei, este soluția unei probleme de optimizare. Soluția problemei se construieşte treptat. Inițial B este mulțimea vidă. Se adaugă în B succesiv elemente din A, atingându-se de fiecare dată un optim local. Dar, această construire nu asigură întotdeauna atingerea unui optim global. De aceea metoda *greedy* nu poate fi aplicată decât dacă se demonstrează că modul de construire a mulțimii B duce la obținerea unui optim global.

Datorită modului de construire a soluției, metoda se mai numeşte ***metoda optimului local***.

Exemplu

Dintre cei n elevi ai unei şcoli, trebuie selectat un lot de k elevi care să participe la un concurs de atletism. Bineînțeles că ne vom grăbi să alegem elevii care au mai avut rezultate la alte concursuri sau pe cei cu înclinații în acest sens. Probabil, vom stabili câteva criterii de selectare: rezultate anterioare, timpi înregistrați la ora de sport etc., pe baza cărora vom întocmi un punctaj. Promițător va fi elevul care realizează punctaj maxim. În continuare, dintre cei rămaşi, vom selecta elevul cu următorul punctaj şi tot aşa până la alegerea celor k elevi. Alegerea cât mai bună a criteriilor va duce la alegerea unui lot performant.

Forma generală a metodei *greedy*:

```
Subalgoritm Greedy(A,B):
  B ← ∅
  cât timp nu Soluție(B) și A ≠ ∅ execută:
    Alege(A,b)
    Elimină(A,b)
    dacă Posibil(B,b) atunci
      Adaugă(B,b)
    sfârșit dacă
  sfârșit cât timp
  dacă Soluție(B) atunci
    scrie B
  altfel
    scrie 'Nu s-a gasit solutie.'
  sfârșit dacă
sfârșit subalgoritm
```

În acest algoritm am utilizat următoarele subalgoritmi:

- Subalgoritmul `Soluție(B)` de tip funcție verifică dacă *B* este soluție optimă a problemei;
- `Alege(A,b)` extrage cel mai promițător element *b* din mulțimea *A* pe care îl poate alege la un moment dat;
- Subalgoritmul `Posibil(B,b)` de tip funcție verifică dacă este posibil să se obțină o soluție, nu neapărat optimă, prin adăugarea lui *b* la mulțimea *B*.

Observații

1. La fiecare pas se alege cel mai promițător element la momentul respectiv. Dacă un element se introduce în mulțimea *B*, el nu va fi niciodată eliminat de acolo.
2. Dacă se alege un element din *A* care nu se poate adăuga mulțimii *B*, el se elimină din *A* şi nu se mai testează ulterior.
3. În rezolvarea problemelor, de multe ori este utilă ordonarea mulțimii *A* înainte ca algoritmul propriu-zis să fie aplicat în funcție de cerințele problemei. Datorită ordonării, elementele din *A* vor fi testate pe rând, începând cu primul. Dacă mulțimea *A* se memorează într-un tablou cu *n* elemente a_1, a_2, …, a_n, algoritmul `Greedy` devine:

```
Subalgoritm Greedy(n,a,B):
  Ordonare(n,a)
  B ← ∅
  i ← 1
  cât timp nu Soluție(B) și (i ≤ n) execută:
    dacă Posibil(B,a_i) atunci
      Adaugă(B,a_i)
    sfârșit dacă
```

```
    i ← i + 1
  sfârşit cât timp
  dacă Soluţie(B) atunci
    scrie B
  altfel
    scrie 'Nu s-a gasit solutie.'
  sfârşit dacă
sfârşit subalgoritm
```

Algoritmii de tip *greedy* nu asigură întotdeauna găsirea soluției optime, chiar dacă la fiecare pas se determină soluția optimă pentru pasul respectiv. De aceea, în momentul în care rezolvăm o problemă prin această metodă, va trebui să găsim o demonstrație matematică riguroasă referitoare la optimalitatea globală a soluției. De cele mai multe ori ea se face prin inducție matematică sau prin reducere la absurd.

2.2. Implementări sugerate

Pentru a vă familiariza cu modul în care se abordează problemele în care trebuie aplicată metoda greedy, vă sugerăm să implementați algoritmi pentru:

1. determinarea ordinii în care se vor servi persoanele care stau la o coadă, astfel încât timpul mediu de aşteptare să fie minim;
2. determinarea submulțimii de sumă maximă a unei mulțimi de numere reale date;
3. eliminarea unor puncte date de-a lungul unei linii drepte, astfel încât distanța minimă dintre oricare două puncte succesive să fie cel puțin egală cu o distanță dată;
4. problema benzii magnetice (Să se stabilească ordinea în care se vor înregistra N fişiere pe o bandă magnetică, astfel încât timpul mediu de accesare a lor să fie minimă. Se ştie că pentru accesarea unui fişier, toate cele înregistrate înaintea lui trebuie citite.);
5. problema continuă a rucsacului (Într-un rucsac încap obiecte de greutate totală G. Cunoscând greutățile şi valorile a N obiecte, să se determine care trebuie împachetate în rucsac pentru a obține o valoare totală maximă.);
6. obținerea unei configurații de N numere plasate pe $N + 1$ poziții (o poziție fiind liberă) dintr-o configurație inițială dată în care există, de asemenea, o poziție neocupată de nici un număr; un număr poate fi mutat dintr-o anumită poziție doar în poziția liberă;
7. eliminarea de către doi jucători a unor numere dintr-un şir de numere, ştergând un număr de la una din marginile şirului; se declară câştigător cel care şterge numere având suma maximă;
8. închirierea unei cabane pe anumite intervale de timp solicitate, astfel încât numărul solicitanților serviți să fie maxim;

9. problema spectacolelor (Într-o sală de spectacole trebuie planificate cât mai multe spectacole dintre cele *N* existente în repertoriu, astfel încât fiecare să respecte ora de începere şi de încheiere planificate.)
10. descompunerea unui şir de numere dat în număr minim de subşiruri descrescătoare; ordinea din subşiruri trebuie să fie aceeaşi cu ordinea din şirul dat:
11. coduri *Huffman*;
12. analiza problemei plății sumei cu bancnote de valori date; găsirea unei mulțimi de monede pentru care algoritmul furnizează (respectiv nu furnizează) soluție optimă.

2.3. Probleme propuse

2.3.1. Meniuri

La inaugurarea unui restaurant sunt prezente mai multe persoane. Clienții îşi aleg din meniul pus la dispoziție câte o specialitate. Dar deocamdată restaurantul a angajat un singur bucătar care pregăteşte mâncărurile una după alta, deci clienții nu pot fi serviți decât pe rând. Presupunând că bucătarul se apucă de lucru după ce s-au strâns toate comenzile, stabiliți în ce ordine trebuie să pregătească specialitățile, astfel încât timpul mediu de aşteptare a clienților să fie minim.

Date de intrare
Prima linie a fişierului de intrare **`MENIU.IN`** conține un număr natural *n*, reprezentând numărul clienților. Următoarea linie conține *n* numere întregi, reprezentând timpul necesar pregătirii mâncărurilor comandate, în ordine, pentru cele *n* persoane. Aceste numere vor fi separate prin câte un spațiu.

Date de ieşire
Fişierul de ieşire **`MENIU.OUT`** va conține două linii. Pe prima linie se va scrie un număr real cu două zecimale, reprezentând timpul mediu de aşteptare, iar pe a doua linie se vor afla *n* numere, reprezentând numerele de ordine ale persoanelor din restaurant, în ordinea în care trebuie servite.

Restricții şi precizări
- $1 \le n \le 1000$;
- $1 \le t \le 100$, unde *t* reprezintă timpul necesar preparării unei specialități de mâncare;
- dacă există mai multe soluții, în fişier se va scrie una singură.

Exemplu

`MENIU.IN`	**`MENIU.OUT`**
`5`	`86.00`
`30 40 20 25 60`	`3 4 1 2 5`

2.3.2. Submulţimi

Se consideră o mulţime având *n* elemente numere întregi. Să se determine o submulţime de sumă maximă a acesteia.

Date de intrare

Prima linie a fişierului de intrare **SUBM.IN** conţine numărul *n*, iar pe următoarea linie se află cele *n* numere întregi, separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire **SUBM.OUT** va conţine o singură linie pe care se vor afla numerele din submulţimea de sumă maximă, separate prin câte un spaţiu.

Restricţii şi precizări

- $1 \le n \le 1000$
- Dacă problema are mai multe soluţii, în fişier se va scrie una singură.

Exemplu

SUBM.IN	SUBM.OUT
7 3 6 0 2 7 -3	3 6 2 7

2.3.3. Maximizare

Se dau mulţimile $A = \{a_1, a_2, ..., a_m\}$ şi $B = \{b_1, b_2, \ldots, b_n\}$ având elemente numere întregi nenule. Se ştie că $n \ge m$.

Să se determine o submulţime $B' = \{x_1, x_2, \ldots, x_m\}$ a lui *B*, astfel încât valoarea expresiei $E = a_1 * x_1 + a_2 * x_2 + \ldots + a_m * x_m$ să fie maximă şi să se calculeze expresia *E*.

Date de intrare

Prima linie a fişierului de intrare **MAXIM.IN** conţine numerele *m* şi *n*, despărţite printr-un spaţiu. A doua şi a treia linie conţin elementele numere întregi ale mulţimii *A*, respectiv *B*. Numerele sunt separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire **MAXIM.OUT** va conţine o singură linie pe care se va afla un număr întreg, care reprezintă valoarea expresiei *E*.

Restricţii şi precizări

- $1 \le m, n \le 100$.

Exemplu

MAXIM.IN	**MAXIM.OUT**
3 5 2 4 3 5 -3 8 -1 2	51

2.3.4. Staţii

Patronul unei companii private de transport în comun a primit de la primăria oraşului aprobarea de a putea folosi o parte din staţiile Regiei Locale de Transport în Comun. Staţiile disponibile sunt situate de-a lungul arterei principale a oraşului.

El se hotărăşte să introducă o cursă rapidă care să străbată oraşul, de la un capăt la celălalt, pe artera principală. Pentru început se ocupă de staţiile situate de aceeaşi parte a drumului. Patronul are o dilemă: dacă opririle vor fi prea dese, atunci străbaterea oraşului va dura prea mult şi va plictisi călătorii, iar dacă staţiile sunt prea rare, călătorii vor fi prea puţini. De aceea, criteriile după care patronul stabileşte staţiile în care va opri cursa rapidă sunt:

- Între două staţii alăturate să fie cel puţin x metri.
- Numărul total de staţii să fie maxim.

Ajutaţi patronul să aleagă staţiile!

Date de intrare

Vom considera staţiile situate pe aceeaşi parte a arterei principale numerotate în ordine, dintr-un capăt până în celălalt cu 1, 2, …, n.

Prima linie a fişierului de intrare **STATII.IN** conţine un număr natural n, reprezentând numărul de staţii situate pe artera principală. Următoarea linie conţine $n - 1$ numere întregi a_i, $i = 1, 2, \ldots, n - 1$ cu semnificaţia: a_i este distanţa dintre staţia i şi staţia $i + 1$. Aceste numere vor fi separate prin câte un spaţiu.

Date de ieşire

Fişierul de ieşire **STATII.OUT** va conţine două linii. Pe prima linie se va scrie un număr întreg k care reprezintă numărul maxim de staţii alese de patron, iar pe a doua linie se vor afla k numere, reprezentând numerele de ordine ale acestor staţii. Numerele vor fi scrise în ordine crescătoare.

Restricţii şi precizări

- $1 \le n \le 1000$;
- $1 \le a_i \le 2000$, $i = 1, 2, ..., n - 1$;
- $1 \le k \le 1000$;
- Dacă există mai multe soluţii, în fişier se va scrie una singură.

Exemplu

STATII.IN	**STATII.OUT**
10 60 100 50 25 25 50 10 10 80 20	5 1 2 5 8 9

2.3.5. Rucsac

Un hoţ dă o spargere la un magazin, găsind *n* obiecte. Fiecare obiect are o greutate şi o valoare. Ştiind că în rucsac nu poate pune obiecte a căror greutate totală depăşeşte valoarea dată *G*, să se precizeze ce obiecte trebuie să ia hoţul pentru a avea un câştig maxim. Hoţul a decis să nu ia neapărat obiectele întregi, el poate lua şi bucăţi din acestea, însă în cazul acesta valoarea obiectului scade proporţional cu greutatea.

Date de intrare

Considerăm obiectele numerotate cu 1, 2, …, *n*. Prima linie a fişierului **RUCSAC.IN** conţine numărul *n* al obiectelor disponibile şi numărul real *G*, reprezentând capacitatea rucsacului. Linia a doua conţine *n* numere întregi separate prin spaţii, reprezentând greutăţile celor *n* obiecte. Linia a treia conţine *n* numere întregi, care reprezintă valorile celor *n* obiecte.

Date de ieşire

Fişierul de ieşire **RUCSAC.OUT** va conţine două linii. Pe prima linie se va scrie un număr real cu două zecimale exacte, reprezentând valoarea totală a obiectelor selectate, iar pe linia a doua se vor scrie numerele de ordine ale obiectelor alese, separate prin câte un spaţiu.

Restricţii şi precizări

- $1 \le n \le 100$;
- În cazul în care un obiect se taie, în fişierul de ieşire indicele acestuia va fi scris ultimul.
- Dacă problema admite mai multe soluţii, în fişier se va scrie una singură.

Exemplu

RUCSAC.IN	**RUCSAC.OUT**
3 5.5 3 3 4 3 2 8	9.50 3 1

2.3.6. Cutii

Al Bundy este în dificultate! Are *n* + 1 cutii de pantofi şi *n* perechi de pantofi identificate prin valorile 1, 2, …, *n* (*n* perechi sunt aşezate în *n* cutii, iar o cutie este liberă). Dar, din păcate, pantofii nu se află la locurile lor. Până să vină *Gary* (şefa lui), *Bundy* trebuie să potrivească pantofii în cutii. Dar pentru că *Gary* poate să apară în orice clipă şi *Bundy* nu vrea să fie prins că nu îşi face treaba cum trebuie, aranjarea pantofilor tre-

buie făcută rapid şi în aşa fel încât să nu trezească bănuieli; prin urmare, dacă scoate o pereche de pantofi dintr-o cutie, trebuie să o pună imediat în cutia liberă. Ajutați-l pe *Al Bundy* să aranjeze pantofii la locurile lor printr-un număr minim de mutări.

Date de intrare
Prima linie a fişierului de intrare **CUTII.IN** conține numărul *n* de cutii. Linia a doua conține *n* + 1 numere distincte, separate prin câte un spațiu, reprezentând dispunerea inițială a perechilor de pantofi în cutiile numerotate de la 1 la *n* + 1. Printre cele *n* + 1 numere unul singur are valoarea 0 şi corespunde cutiei goale. Linia a treia conține *n* + 1 numere separate prin câte un spațiu, reprezentând configurația finală cerută, în care valoarea 0 se află pe poziția căruia îi corespunde cutia goală.

Date de ieşire
Fişierul de ieşire **CUTII.OUT** va conține pe prima linie un număr întreg *k*, reprezentând numărul minim de mutări.

Restricții şi precizări
- $1 < n < 100$.

Exemplu

CUTII.IN	**CUTII.OUT**	*Explicații*
4	4	Mutarea 1: perechea 1 se mută din cutia 3 în cutia 4
3 4 1 0 2		Mutarea 2: perechea 2 se mută din cutia 5 în cutia 3
4 0 2 1 3		Mutarea 3: perechea 3 se mută din cutia 1 în cutia 5
		Mutarea 4: perechea 4 se mută din cutia 2 în cutia 1

2.3.7. Subşiruri

Se dă un şir de *n* numere întregi. Să se descompună acest şir în număr minim de subşiruri strict crescătoare, astfel încât numerele lor de ordine (din şirul dat) să fie ordonate crescător în subşirurile formate.

Date de intrare
Prima linie a fişierului de intrare **SUBSIR.IN** conține numărul *n*, reprezentând numărul de elemente din şir. Următoarea linie conține cele *n* numere întregi, separate prin câte un spațiu.

Date de ieşire
Fişierul de ieşire **SUBSIR.OUT** va conține pe prima linie un număr întreg *k*, reprezentând numărul minim de subşiruri care se pot forma. Pe următoarele *k* linii vor fi descrise subşirurile crescătoare. Un subşir va fi precizat prin numerele de ordine ale elementelor din şirul inițial, despărțite prin câte un spațiu.

Restricţii şi precizări
- $1 \le n \le 10000$;
- elementele şirului sunt numere cuprinse în intervalul [0, 40000].

Exemplu

SUBSIR.IN	**SUBSIR.OUT**	*Explicaţie*
10	3	Se pot forma 3 subşiruri crescătoare:
2 3 1 6 8 3 7 9 5 7	1 2 4 5 8	2, 3, 6, 8, 9
	3 6 7	1, 3, 7
	9 10	5, 7

2.3.8. Medici

O asociaţie caritabilă asigură consultaţii medicale gratuite pentru cei fără posibilităţi materiale. Există un singur cabinet dotat cu aparatură medicală. Din acest motiv la un moment dat un singur medic poate face consultaţii. Asociaţia apelează la *n* medici de diverse specialităţi, care îşi oferă benevol serviciile. Fiecare prezintă un singur interval $[s_i, f_i]$ de-a lungul aceleiaşi zile, în care este disponibil. Ajutaţi asociaţia să realizeze o planificare a consultaţiilor în cabinet, astfel încât numărul de medici să fie maxim.

Date de intrare
Prima linie a fişierului de intrare **MEDICI.IN** conţine numărul *n* al medicilor. A doua şi a treia linie conţin orele de început şi respectiv orele de sfârşit, corespunzătoare celor *n* medici (numere întregi). Numerele sunt separate prin câte un spaţiu.

Date de ieşire
Fişierul de ieşire **MEDICI.OUT** va conţine pe prima linie un număr *k,* reprezentând numărul maxim de medici care se pot selecta, astfel încât să fie îndeplinită condiţia din enunţ. A doua linie va conţine *k* numere care vor reprezenta indicii medicilor selectaţi.

Restricţii şi precizări
- $1 \le n \le 1000$;
- $1 \le f_i < s_i \le 1000$, pentru $i = 1, 2, ..., n$.

Exemplu

MEDICI.IN	**MEDICI.OUT**
4	3
2 6 20 4	1 4 3
3 8 23 7	

2.3.9. Turişti

O agenție de turism montan are *n* ghizi capabili să însoțească grupurile de turişti străini în drumeții montane. Toate drumețiile care se organizează durează exact *m* zile. Un ghid poate să însoțească într-o drumeție un singur grup, el fiind nedisponibil pentru alte grupuri în acea perioadă. La agenție se adună *p* solicitări înainte de începerea sezonului. Fiecare din cele *p* grupuri solicitante specifică data la care doreşte să înceapă drumeția.

Ajutați directorul agenției să aleagă numărul maxim de grupuri care vor putea fi însoțite de ghizi pe parcursul unui an întreg. În plus, el a hotărât că dacă există solicitări care au data de început *d* în timpul sezonului şi s-ar putea desfăşura în totalitate doar dacă se depăşeşte ultima zi a sezonului, se prelungeşte sezonul cu zilele necesare.

Date de intrare

Prima linie a fişierului de intrare **`TURIST.IN`** conține numerele *k*, *n*, *m* şi *p*, despărțite prin câte un spațiu (*k* reprezintă numărul zilelor din sezon, *n* este numărul ghizilor, *m* reprezintă durata drumețiilor, iar *p* reprezintă numărul solicitărilor.) Pe următoarea linie sunt scrise datele d_i ($i = 1, 2, \ldots, p$) pe care grupurile le-au scris pe solicitări.

Date de ieşire

Fişierul de ieşire **`TURIST.OUT`** conține un număr întreg care reprezintă numărul maxim de solicitări rezolvate.

Restricții şi precizări

- $1 \le n \le 100$;
- $1 \le m \le 14$;
- $1 \le p \le 1000$;
- $1 \le k \le 365$;
- $1 \le d_i \le k$ ($i = 1, 2, ..., p$).

Exemplu

`TURIST.IN`
```
20 2 5 8
8 2 10 10 4 9 6 19
```

`TURIST.OUT`
```
5
```

Explicații

Exemplul poate corespunde următoarei situații:

- primul ghid rezolvă solicitările care încep în zilele 2, 8 şi 19
- al doilea ghid rezolvă solicitările care încep în zilele 4 şi 9

2.4. Soluțiile problemelor propuse

2.4.1. Meniuri

În această problemă se cere stabilirea unui minim. Se ştie că problemele de optim se pot rezolva aplicând metoda *greedy*, dacă se poate arăta că, pe baza alegerii optimului local, metoda generează soluții optime. Deci, mai întâi trebuie să demonstrăm că acest lucru este posibil.

În concluzie, ştiind că $t_{k_1} \le t_{k_2} \le ..., \le t_{k_n}$ sunt timpii necesari preparării specialităților, presupunem că dacă acestea se prepară în ordinea $k_1, k_2, ..., k_n$, vom avea un timp total de aşteptare minim. Evident $\{k_1, k_2, ..., k_n\} = \{1, 2, ..., n\}$.

Demonstrația o facem prin *reducere la absurd*. Presupunem că ordinea $k_1, k_2, ..., k_n$ nu asigură timpul total minim de aşteptare. Din această presupunere temporară rezultă că există o altă ordine de servire a clienților (diferită de ordinea $k_1, k_2, ..., k_n$), care conduce la un timp total de aşteptare mai mic. Presupunem că o astfel de ordine diferă de ordinea $k_1, k_2, ..., k_n$ în pozițiile i şi j.

Conform ordinii $k_1, ..., k_i, ..., k_j, ..., k_n$ avem timpul total de aşteptare:

$$timptotal(k_1, ..., k_i, ..., k_j, ..., k_n) =$$
$$= nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_i} + ... + (n-j+1)t_{k_j} + ... + t_{k_n}.$$

Conform presupunerii de mai înainte, pentru ordinea $k_1, ..., k_j, ..., k_i, ..., k_n$ avem timpul total de aşteptare:

$$timp_{total}(k_1, ..., k_j, ..., k_i, ..., k_n) =$$
$$= nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_j} + ... + (n-j+1)t_{k_i} + ... + t_{k_n}$$

care este mai mic decât

$$nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_i} + ... + (n-j+1)t_{k_j} + ... + t_{k_n} =$$
$$= timp_{total}(k_1, ..., k_i, ..., k_j, ..., k_n).$$

Eliminând termenii asemenea din partea stângă, respectiv dreaptă a operatorului relațional avem:

$$nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_j} + ... + (n-j+1)t_{k_i} + ... + t_{k_n} <$$
$$< nt_{k_1} + (n-1)t_{k_2} + ... + (n-i+1)t_{k_i} + ... + (n-j+1)t_{k_j} + ... + t_{k_n}$$
$$(n-i+1)t_{k_j} + (n-j+1)t_{k_i} < (n-i+1)t_{k_i} + (n-j+1)t_{k_j}$$
$$(n-i+1-n+j-1)t_{k_j} < (n-i+1-n+j-1)t_{k_i}$$
$$(j-i)t_{k_j} < (j-i)t_{k_i}$$

Deci, împărțind această relație cu $j - i$ (valoare pozitivă, deoarece am pornit cu presupunerea că $i < j$), avem $t_{k_j} < t_{k_i}$, ceea ce înseamnă că avem un timp de aşteptare mai mic, după unul mai mare în şirul ordonat crescător a timpilor de aşteptare, constatare care evident este o contradicție cu ipotezele fixate.

Pentru a minimiza timpul mediu de aşteptare ($timp_{mediu}$) va trebui să minimizăm timpul total de aşteptare ($timp_{total}$) al persoanelor pentru a fi servite, deoarece $timp_{mediu} = timp_{total}/n$. O persoană va trebui să aştepte prepararea meniului ei şi a tuturor persoanelor care vor fi servite în faţa lui. Intuitiv, dacă o persoană care doreşte un meniu sofisticat este servită înaintea uneia al cărei meniu se prepară mai repede, timpul total de aşteptare al celor doi este mai mare decât dacă servirea s-ar fi făcut invers. De exemplu, pentru timpii 30 şi 20, în prima situaţie $timp_{total} = 30 + (30 + 20)$, iar în a doua situaţie $timp_{total} = 20 + (20 + 30)$.

În concluzie, vom alege persoanele în ordinea crescătoare a timpilor de preparare a meniurilor, ceea ce necesită ca ele să fie ordonate crescător în funcţie de acest criteriu şi vom calcula timpul total de aşteptare ca fiind suma timpilor de aşteptare a fiecărei persoane.

În situaţia în care există mai multe durate egale de preparare, nu contează ordinea alegerii acestora.

2.4.2. Submulţimi

În cazul în care mulţimea dată conţine doar elemente strict pozitive, submulţimea va fi formată din toate acestea şi evident, în acest caz se obţine o sumă maximă. Dacă în mulţime avem şi numere negative, atunci suma maximă se obţine din mulţimea formată din elementele strict pozitive, deoarece dacă am mai adăuga un element y, valoarea sumei s-ar micşora (y este negativ) sau ar rămâne egală cu cea calculată din elementele pozitive (dacă $y = 0$).

Dacă în mulţime nu există nici un număr strict pozitiv, vom alege cel mai mare număr negativ şi acesta va fi singurul element din submulţimea cerută.

În cazul în care mulţimea îl conţine pe 0 şi are cel puţin două elemente, există două soluţii: una în care apare 0 şi una în care 0 nu apare în submulţime.

2.4.3. Maximizare

În maximizarea lui E vom aplica următoarele proprietăţi evidente:

- Dacă a, x, y sunt trei numere întregi, unde $a > 0$, relaţia $a \cdot x > a \cdot y$ este adevărată în cazul în care $x > y$.
- Dacă a, x, y sunt trei numere întregi, unde $a < 0$, expresia $a \cdot x > a \cdot y$ este adevărată în cazul în care $x < y$.

Vom sorta mai întâi crescător elementele şirurilor care memorează mulţimile A şi B. Vom obţine $a_1 < a_2 <\ldots< a_m$ şi $b_1 < b_2 <\ldots< b_n$.

În ceea ce priveşte valorile elementelor din cele două mulţimi, putem avea trei situaţii:

a) Elementele din A sunt pozitive. În acest caz valoarea lui E este maximă, dacă înmulţim cel mai mare element din A (a_m) cu cel mai mare element din B (b_n), apoi a_{m-1} cu b_{n-1}, a_{m-2} cu b_{n-2} etc.

b) Elementele din A sunt negative. În acest caz valoarea lui E este maximă, dacă înmulțim cel mai mic element din A (a_1) cu cel mai mic element din B (b_1), apoi a_2 cu b_2, a_3 cu b_3, ..., a_m cu b_m.
c) A are primele m_1 elemente numere negative şi următoarele ($m - m_1$) elemente numere pozitive. În acest caz valoarea lui E este maximă dacă înmulțim primele m_1 elemente din A cu primele m_1 elemente din B, iar restul elementelor din A, în ordine, cu ultimele $m - m_1$ elemente din mulțimea B.

În scrierea programului, vom aplica direct punctul c). Dacă A are m_1 elemente negative, vom selecta elementele corespunzătoare lor din B, începând cu b_1. Vom alege în continuare pentru elementele pozitive din A ultimele $m - m_1$ elemente din B.

2.4.4. Staţii

Fie mulțimea ordonată a staţiilor disponibile $A = \{1, 2, \ldots n\}$. Corespunzător acesteia se cunosc cele $n - 1$ distanțe dintre oricare două staţii alăturate: $a_1, a_2, \ldots, a_{n-1}$.

Problema revine la a determina o submulțime $B \subseteq A$ cu număr maxim de elemente (staţii), $B = \{i_1, i_2, \ldots, i_k\}$ în care să se păstreze ordinea din A şi să avem îndeplinită proprietatea $a_{i_{j+1}} - a_{i_j} \le x$, pentru orice $j = 1, 2, \ldots, k - 1$.

```
Subalgoritm Selecţie_Greedy(A,B):
  B ← {1}
  dist_ultim ← 0
  pentru j=2,n execută:
    dacă aj-1 + dist_ultim ≥ x atunci
      B ← B ∪ {j}
      dist_ultim ← 0
    altfel
      dist_ultim ← dist_ultim + aj-1
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Algoritmul începe cu selectarea staţiei 1. În continuare se parcurge şirul staţiilor, căutând prima staţie suficient de îndepărtată de 1. Aceasta se adaugă la soluţie. Algoritmul se repetă în acest fel până când s-au verificat toate staţiile. În scrierea algoritmului s-a folosit variabila `dist_ultim` pentru calcularea distanţei dintre două staţii.

Demonstraţie

Mai întâi vom demonstra proprietatea: *Există întotdeauna o soluţie optimă a problemei care conţine staţia având numărul* 1.

Vom nota cu *dist*(*i*, *j*) distanța dintre stațiile *i* şi *j*. Fie $B \subseteq A$ o soluție optimă a problemei, unde $B = \{j_1, j_2, \ldots, j_k\}$ nu conține elementul 1, corespunzător primei stații. Atunci $dist(j_1, j_2) \geq x$. Dar cum stația 1 precede stația j_1, avem relația: $dist(1, j_2) = dist(1, j_1) + dist(j_1, j_2)$. Deci $dist(1, j_2) \geq x$. Prin urmare şi mulțimea $B' = \{1, j_2, \ldots, j_k\}$ verifică condițiile problemei şi are numărul de elemente egal cu cel al mulțimii *B*. În aceste condiții şi B' este o soluție optimă.

Am demonstrat că există o soluție optimă a problemei care conține stația cu numărul 1. În continuare vor putea fi selectate în construirea soluției optime doar acele stații care, față de stația 1, se situează la o distanță cel puțin egală cu *x*.

Dacă *B* este o soluție optimă pentru mulțimea *A*, vom arăta că $B'' = B - \{1\}$ construită cu subalgoritmul `Selecție_Greedy` este o soluție optimă pentru $A' = A - \{j \mid dist(1, j) < x\}$. Presupunem că B'' nu este soluție optimă. Deci, există B''', astfel încât $|B'''| > |B''|$. Dar atunci mulțimea $\{1\} \cup B'''$ are cardinalul $|\{1\} \cup B'''| > |B|$, ceea ce reprezintă o contradicție cu presupunerea că *B* este soluție optimă. Aşadar mulțimea B'', construită cu subalgoritmul `Selecție_Greedy` are număr maxim de elemente. Prin urmare şi mulțimea $B = \{1\} \cup B''$ va avea număr maxim de elemente şi deci va fi o soluție a problemei date.

2.4.5. Rucsac

Problema este cunoscută în literatura de specialitate sub denumirea de *problema fracționară a rucsacului*.

Să luăm pentru început un exemplu concret. În magazin există 4 obiecte şi capacitatea rucsacului este $G = 10$.

Număr obiect	1	2	3	4
Greutate	5	4	5	2
Valoare	10	20	1	3

Vom calcula, pentru fiecare obiect, valoarea pe unitatea de greutate.

Valoare/greutate	2	5	0.2	1.5

Vom alege pe rând obiectele în ordine descrescătoare a acestei valori până la umplerea rucsacului.

Pasul 1. Se alege obiectul 2 (are raportul valoare/greutate maxim). Se calculează *G_parțial* = 4 şi *Câştig_parțial* = 20.

Pasul 2. Se alege obiectul 1 (are raportul valoare/greutate maxim între obiectele nealese). Se calculează *G_parțial* = 4 + 5 şi *Câştig_parțial* = 20 + 5.

Pasul 3. Se alege obiectul 4, care se „taie”. Este necesar doar în cantitatea 1, deci câştigul în urma alegerii lui va fi de 1.5, *G_parțial* = 10 şi *Câştig_parțial* = 26,5.

Câştigul final este 26.5.

Deoarece se pot lua obiecte tăiate, se obține o încărcare optimă a rucsacului, la fiecare pas luându-se obiectul cel mai valoros.

În algoritm se calculează pentru fiecare obiect raportul valoare/greutate şi se ordonează descrescător elementele şirului în funcție de acesta. Se extrag obiecte din şirul ordonat până când acestea vor umple rucsacul sau până se vor epuiza. În situația în care s-a ajuns la un obiect cu o greutate prea mare, acesta se taie.

În algoritmul de mai jos `nr_alese` contorizează numărul obiectelor alese de hoț, `y` este un şir de indici, reprezentând numerele de ordine ale obiectelor alese, iar `Câştig` este variabila care reține rezultatul final, reprezentând câştigul total realizat de hoț. În momentul apelării subalgoritmului `Alege`, şirul *x* este deja ordonat descrescător în funcție de raportul valoare/greutate.

```
Subalgoritm Alege(nr_alese,y,Câştig):
  i ← 1                                              { primul obiect }
  G_parțial ← 0                  { deocamdată greutatea împachetată este 0 }
  sfârşit ← fals                                  { încă nu s-a împachetat rucsacul }
  cât timp (G_parțial < G) şi (i ≤ n) şi nu sfârşit execută:
    nr_alese ← nr_alese + 1                        { se selectează un obiect }
    y[nr_alese] ← x[i].nr
                  { în şirul soluției se păstrează numărul de ordine al obiectului }
    dacă x[i].gr ≤ G - G_parțial atunci       { poate mai încape ceva... }
      G_parțial ← G_parțial + x[i].gr
      Câştig_parțial ← Câştig_parțial + x[i].v
    altfel                                  { ultimul obiect se taie dacă este cazul }
      Câştig ← Câştig_parțial + (G-G_parțial) * x[i].gv
                          { nu mai încape nimic, intrerupem reluarea while-ului }
      sfârşit ← adevărat
    sfârşit dacă
    i ← i + 1
  sfârşit cât timp
sfârşit subalgoritm
```

Observații

- Pentru anumite date de intrare problema admite mai multe soluții. Pot exista obiecte care conduc la acelaşi câştig. În acest caz ele se selectează în ordinea în care apar în şirul ordonat. Afişarea tuturor variantelor de selectare a obiectelor se poate face folosind metoda backtracking.
- Pot exista situații în care se aleg toate obiectele, dar nu se completează capacitatea rucsacului. La scrierea algoritmului trebuie să ținem cont şi de acest caz.

2.4.6. Cutii

Fie mai întâi exemplul din enunț. Avem 4 perechi de pantofi şi 5 cutii în care acestea sunt aranjate în felul următor:

```
3 4 1 0 2
```

Se cunoaşte şi aranjarea dorită:

```
4 0 2 1 3
```

Adică, inițial, în prima cutie se găseşte perechea 3, dar acolo trebuie să fie perechea 4, în a doua cutie se găseşte perechea 4, dar a doua cutie trebuie să fie goală etc.

Un şir posibil de mutări (dar care nu conduce la soluția optimă) ar fi:

Mutarea 1: perechea 3 se mută din cutia 1 în cutia 4;
Mutarea 2: perechea 4 se mută din cutia 2 în cutia 1;
Mutarea 3: perechea 1 se mută din cutia 3 în cutia 2;
Mutarea 4: perechea 2 se mută din cutia 5 în cutia 3;
Mutarea 5: perechea 3 se mută din cutia 4 în cutia 5;
Mutarea 6: perechea 1 se mută din cutia 2 în cutia 4.

Am aranjat pantofii în **6** mutări.

Dar putem proceda în felul următor:

Mutarea 1: perechea 1 se mută din cutia 3 în cutia 4 (a ajuns unde trebuie, acum cutia 3 este goală);
Mutarea 2: perechea 2 se mută din cutia 5 în cutia 3 (a ajuns unde trebuie, acum cutia 5 este goală);
Mutarea 3: perechea 3 se mută din cutia 1 în cutia 5 (a ajuns unde trebuie, acum cutia 1 este goală);
Mutarea 4: perechea 4 se mută din cutia 2 în cutia 1 (a ajuns unde trebuie, acum cutia 2 este goală, aşa cum cere configurația dorită).

Astfel am aranjat pantofii în **4** mutări.

Analizând exemplul, ajungem la concluzia că pentru a aranja pantofii în cutii, trebuie să mutăm fiecare pereche *p* de pantofi la locul ei (dacă nu este acolo). Există două cazuri posibile:

A. Cutia în care trebuie pusă perechea *p* este goală;
B. Cutia perechii *p* este ocupată de altă pereche *q*.

În cazul **A** vom muta perechea *p* la locul ei, în cutia goală, fără a mai face alte mutări. În cazul **B** trebuie eliberată mai întâi cutia ocupată de *q* şi doar pe urmă vom putea muta perechea *p* în cutia ei, acum eliberată.

Situația **A** este de preferat, deoarece în acest caz o pereche ajunge la locul ei printr-o singură mutare. În situația **B** sunt necesare două mutări pentru a duce o pereche de pantofi unde trebuie. *Din acest motiv vom identifica în rezolvarea problemei cazul* **A** *cât timp este posibil* şi o vom rezolva. În celelalte cazuri (similare situației **B**) vom aranja câte o pereche în cutia potrivită prin două mutări, conform explicațiilor date.

În şirurile `ci` şi `cf` se memorează configuraţiile iniţială şi finală a perechilor de pantofi în cutii. În variabilele `cigol` şi `cfgol` se memorează indicele cutiei goale în configuraţia iniţială şi respectiv finală.

```
Subalgoritm Număr_mutări(n,ci,cf):
  repetă
    schimb ← fals                                  { încă nu am schimbat nimic }
    cât timp cigol ≠ cfgol execută:                { rezolvăm situaţia A }
      ci[cigol] ← cf[cigol]      { mutăm în cutia goală perechea potrivită }
      cigol ← Caută(cf[cigol])
      ci[cigol] ← 0                              { se eliberează o nouă cutie }
      mutări ← mutări + 1                          { am făcut doar o mutare }
      schimb ← adevărat                            { am efectuat o schimbare }
    sfârşit cât timp
                                                 { se caută o cutie nearanjată }
    cât timp (ci[i] = cf[i]) şi (i ≤ n+1) execută:
      i ← i + 1
    sfârşit cât timp
    dacă i ≤ n atunci                              { rezolvăm situaţia B }
                          { golim cutia nearanjată şi aducem perechea potrivită }
      ci[cigol] ← ci[i]
      ci[i] ← cf[i]
      cigol ← Caută(ci[i])
      i ← i + 1
      mutări ← mutări + 2                          { am făcut două mutări }
      schimb ← adevărat                            { am efectuat o schimbare }
    sfârşit cât timp
  până când nu schimb    { până când nu a mai fost necesară nici o schimbare }
sfârşit subalgoritm
```

Observaţii

1. În algoritm s-a folosit funcţia `Caută(p)` care returnează numărul cutiei care conţine perechea *p*.
2. Situaţia **A** apare dacă indicele cutiei goale în configuraţia iniţială este diferit de indicele cutiei goale în configuraţia finală.

2.4.7. Subşiruri

Subşirurile se construiesc simultan, printr-o singură parcurgere a şirului dat. Pentru fiecare element x_i din şir se caută un subşir existent la care acesta se poate adăuga la sfârşit. Dacă nu se găseşte un astfel de subşir, vom crea unul nou în care x_i se va introduce ca prim element.

Dacă vom găsi mai multe subşiruri în care s-ar putea adăuga x_i, îl vom alege pe cel pentru care valoarea ultimului element este maximă.

Exemplu

Fie $n = 8$ şi şirul de numere 9, 3, 10, 1, 11, 5, 12, 8

Pasul	**Observaţii**	**Şir 1**	**Şir 2**	**Şir 3**
1.	9 va fi primul element din primul şir	**9**	-	-
2.	3 va fi primul element din al doilea şir	**9**	**3**	-
3. (*)	10 se adaugă la sfârşitul şirului 1	9, **10**	**3**	-
4.	1 va fi primul element din şirul 3	9, **10**	**3**	**1**
5. (*)	11 se adaugă la sfârşitul şirului 1	9, 10, **11**	**3**	**1**
6. (*)	5 se adaugă la sfârşitul şirului 2	9, 10, **11**	3, **5**	**1**
7. (*)	12 se adaugă la sfârşitul şirului 1	9,10,11, **12**	3, **5**	**1**
8. (*)	8 se adaugă la sfârşitul şirului 2	9,10,11, **12**	3, 5, **8**	**1**

Observaţii

1. La paşii marcaţi cu (*) au existat mai multe posibilităţi de alegere a şirului la care să se adauge elementul curent.
2. Se observă că prin acest mod de construire a subşirurilor, elementele maxime din fiecare subşir (cele care s-au adăugat ultima oară) formează la rândul lor un şir descrescător (la pasul 4: 10, 3, 1, dar şi la pasul 8: 12, 8, 1). Dacă nu ar fi aşa, ar însemna că la un moment dat nu am ales în mod corect subşirul în care se adaugă un element. De exemplu, în tabelul de mai sus, elementul 10 nu poate fi pus după 3, deoarece el trebuie „înghesuit" (conform cerinţelor problemei) lângă 9. Din acest motiv, vom căuta cel mai potrivit subşir pentru un element cu algoritmul de căutare binară în şirul „vârfurilor" fiecărui subşir construit până la momentul respectiv. Astfel vom găsi mai rapid subşirul potrivit pentru elementul curent.

În următorul algoritm folosim notaţiile:

`n`: numărul elementelor în şirul dat;
`x`: şirul dat;
`nrs`: numărul subşirurilor crescătoare;
`s`: `s[i]` =numărul de ordine al subşirului din care face parte `x[i]`;
`nrs`: numărul subşirurilor crescătoare;
`vf`: valorile elementelor maxime din fiecare subşir.

```
Subalgoritm ConstruieşteSubşiruri(n,nrs,x,s,vf):
  nrs ← 1
  s[nrs] ← 1                                { x[1] face parte din subşirul 1 }
```

```
  vf[nrs] ← x[1]
  pentru j=2,n execută:                { se caută subşirul potrivit pentru x[j] }
    k ← Caută(vf,1,nrs,x[j])    { se apelează algoritmul de căutare binară }
   { care va returna numărul de ordine al subşirului unde ar trebui să se afle x[j] }
                                    { sau 0 dacă un astfel de subşir nu s-a găsit }
    dacă k = 0 atunci                  { dacă nu am găsit un subşir potrivit }
      nrs ← nrs + 1                        { x[j] va forma un nou subşir }
      s[j] ← nrs                         { numărul de ordine al subşirului }
      vf[nrs] ← x[j]          { valoarea elementului maxim din subşirul nrs }
    altfel                             { x[j] se va adăuga unui subşir existent }
      s[j] ← k                           { numărul de ordine al subşirului }
      vf[k] ← x[j]              { valoarea elementului maxim din subşirul k }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Chiar dacă algoritmul căutării binare se cunoaşte, prezentăm modul în care se aplică în rezolvarea acestei probleme, deoarece stabilirea numărului de ordine al subşirului în care se va adăuga elementul curent din şirul dat se realizează în cadrul acestui subalgoritm. Presupunem că în parametrul `ce` s-a transmis valoarea elementului `x[j]` curent. Variabila `găsit` o folosim în scopul de a reține faptul că s-a găsit sau nu `ce` în şirul vârfurilor.

```
Subalgoritm Caută(vf,s,d,ce):
  găsit ← fals
  cât timp (s ≤ d) şi nu găsit execută:
    m ← (s+d) div 2
    dacă vf[m] = ce atunci
      găsit ← adevărat
    altfel
      dacă vf[m] > ce atunci
        s ← m + 1
      altfel
        d ← m - 1
      sfârşit dacă
    sfârşit dacă
  sfârşit cât timp
  dacă găsit atunci          { dacă l-am găsit pe x[j] în capătul unui subşir }
                                                  { avansăm în şirul vârfurilor }
    cât timp (ce = vf[m]) şi (m ≤ nrs) execută:
      m ← m + 1                        { cât timp se succede aceeaşi valoare }
    sfârşit cât timp
```

```
    dacă m ≤ nrs atunci        { dacă ne-am oprit după vârful unui şir existent }
      Caută ← m                                  { aici se va adăuga x[j] }
    altfel
        { dacă şi ultimul şir îl are în vârf pe x[j], trebuie să creăm un subşir nou }
      Caută ← 0
    sfârşit dacă
  sfârşit dacă
  dacă nu găsit atunci                        { dacă nu l-am găsit pe x[j] }
    dacă (s ≤ nrs) şi (vf[s] < ce) atunci
                                    { acolo unde a eşuat căutarea, şirul vf }
      Caută ← s          { conţine o valoare mai mică, am găsit locul lui x[j] }
    altfel
      Caută ← 0               { în caz contrar trebuie să creăm un subşir nou }
    sfârşit dacă
  sfârşit dacă
sfârşit subalgoritm
```

2.4.8. Medici

Fie mulțimea medicilor $M = \{1,2,\ldots, n\}$. Corespunzător acesteia se cunosc cele n intervale care memorează orarele medicilor: $[s_1, f_1]$, $[s_2, f_2]$, …, $[s_n, f_n]$.

Dacă un medic i este selectat, cabinetul va fi ocupat în intervalul de timp $[s_i, f_i]$.

Vom spune că doi medici i şi j au orarele compatibile dacă $[s_i, f_i] \cap [s_j, f_j] = \varnothing$. Problema revine la a determina o submulțime maximă $S \subseteq M$ de medici cu orare compatibile două câte două.

Schimbăm ordinea medicilor, obținând $M = \{i_1, i_2, \ldots, i_n\}$, astfel încât $f_{i_1} \le f_{i_2} \le \ldots \le f_{i_n}$ (medicii apar în M ordonați după capătul din dreapta al intervalului de timp în care ei pot consulta). Evident $\{i_1, i_2, \ldots, i_n\}$ este o permutare a mulțimii $\{1, 2, \ldots, n\}$.

Selectarea o vom realiza cu algoritmul `Selecţie-Greedy`:

```
Subalgoritm Selecţie-Greedy(Sel,s,f):
  Sel ← {i1}                                  { selectarea primului medic (i1) }
  ultim ← i1                          { acesta este şi ultimul printre cei selectaţi }
  pentru k=2,n execută:
            { căutarea următorului medic din M care are orarul compatibil cu el }
    dacă sik > fultim atunci
      Sel ← Sel ∪ {ik}                        { cel compatibil se adaugă soluţiei }
      ultim ← ik                                { acum cel adăugat este ultimul }
    sfârşit dacă
  sfârşit pentru
sfârşit subalgoritm
```

Demonstrație

Considerăm că am ordonat mulțimea medicilor după timpul final al fiecărui medic: $M = \{i_1, i_2, \ldots, i_n\}$ şi prin urmare $f_{i_1} \le f_{i_2} \le f_{i_3} \le \ldots \le f_{i_n}$. (1)

Deci primul medic din M (cel cu indice i_1) va avea valoarea f_{i_1} minimă dintre toate valorile f_{i_k}, $k = 1, 2, \ldots, n$. Vom demonstra mai întâi următoarea propoziție:

Există întotdeauna o soluție optimă a problemei care îl conține pe i_1.

Fie $A \subseteq M$ o soluție optimă a problemei, unde $A = \{j_1, j_2, \ldots, j_k\}$. Presupunem că această mulțime nu îl conține pe i_1. Atunci are loc relația: $f_{j_1} < s_{i_2}$. (2)

Fie $A' = \{i_1, j_2, \ldots, j_k\}$. Din (1) şi (2) rezultă că $f_{i_1} < s_{i_2}$, deci medicul i_1 va avea orarul compatibil cu medicii $j_2, j_3, \ldots, j_k$. Deci toți medicii din A' au orare compatibile reciproc. Cum $|A'| = |A|$, rezultă că şi A' este o soluție optimă a problemei.

Am demonstrat deci că există o soluție optimă a problemei care conține medicul i_1. În continuare vor putea fi selectați în soluția optimă doar acei medici care au orarele compatibile cu i_1.

Dacă A este o soluție optimă pentru M, vom arăta că $A'' = A - \{i_1\}$ construită cu algoritmul `Selecție-Greedy` este o soluție optimă pentru $M' = M - \{j \mid s_j \le f_{i_1}\}$. Presupunem că A'' nu e soluție optimă. Deci există o soluție B, astfel încât $|B| > |A''|$. Dar atunci $|B \cup \{i_1\}| > |A|$. Am ajuns la contradicție cu presupunerea inițială că A este soluție optimă.

Programul presupune citirea datelor din fişier, sortarea acestora şi parcurgerea şirului pentru construirea mulțimii A, deci complexitatea este $O(n) + O(n \lg n) = O(n \lg n)$.

2.4.9. Turişti

Dacă agenția dispune de un singur ghid, trebuie să îi repartizăm cât mai multe solicitări, fără suprapunerea perioadelor. Există deci p intervale de forma $[s_i, s_i + m]$, s_i fiind data începerii drumeției scrisă pe solicitarea i. Dintre acestea trebuie ales un număr maxim de intervale care să nu se suprapună. Regăsim în acest caz datele problemei anterioare, uşor simplificată. Cum toate drumețiile se execută în acelaşi număr de zile, vom putea alege solicitările în funcție de data inițială s_i, respectând algoritmul problemei anterioare.

Pe caz general vom memora în tabloul x_i numărul de cereri existente cu ziua de început i. Numărul de elemente ale tabloului este egal cu numărul de zile ale sezonului. În subalgoritmul `Rezolvă_ziua` încercăm să găsim ghizi pentru toate drumețiile care încep în ziua i. Pentru a găsi un ghid disponibil, vom traversa şirul `zi_liberă` în care componenta a i-a conține prima zi în care ghidul i este disponibil.

```
Subalgoritm Rezolvă_ziua(i):
  cât timp x[i] > 0 execută
    primul_ghid_disponibil ← Caută_ghid(i)
```

```
    dacă primul_ghid_disponibil > n atunci
      ieşire forţată din cât timp                    { nu se poate rezolva cererea }
    altfel
      zi_liberă[primul_ghid_disponibil] ← i + m
                { prima zi liberă a ghidului găsit disponibil va fi cu m mai mare }
      rezolvate ← rezolvate + 1                      { încă un grup rezolvat }
      x[i] ← x[i] - 1       { scade numărul solicitărilor care încep în ziua i }
    sfârşit dacă
  sfârşit cât timp
sfârşit subalgoritm
```

Căutarea ghidului disponibil se realizează cu subalgoritmul `Caută_ghid`:

```
Subalgoritm Caută_ghid(d):
{ caută primul ghid disponibil începând cu data d şi dacă nu găseşte, întoarce n + 1}
  i ← 1
  cât timp (i ≤ n) şi (zi_liberă[i] > d) execută:
    i ← i + 1
  sfârşit câttimp
  Caută_ghid ← i
sfârşit subalgoritm
```