Arbori de INTERVALE

Dana Lica

În cadrul acestui articol vom prezenta o structură de date cunoscută sub numele de arbore de intervale (engl. segment tree), precum și o serie de aplicații în geometria computațională.

Arbori de intervale

Un arbore de intervale este un arbore binar în care fiecare nod poate avea asociată o structură auxiliară (anumite informații).

Dându-se două numere întregi *st* și *dr*, cu *st* < *dr*, atunci arborele de intervale *T*(*st*, *dr*) se construiește recursiv astfel:

- considerăm rădăcina *nod* având asociat intervalul [*st*, *dr*];
- dacă *st* < *dr* atunci, vom avea asociat subarborele stâng *T*(*st*, *mij*), respectiv subarborele drept *T*(*mij* + 1, *dr*), unde *mij* este mijlocul intervalului [*st*, *dr*].

Intervalul [*st*, *dr*] asociat unui nod se numește ***interval standard***. Frunzele arborelui sunt considerate ***intervale elementar***e, ele având lungimea 1. Un arbore de intervale este ilustrat în figura 1.

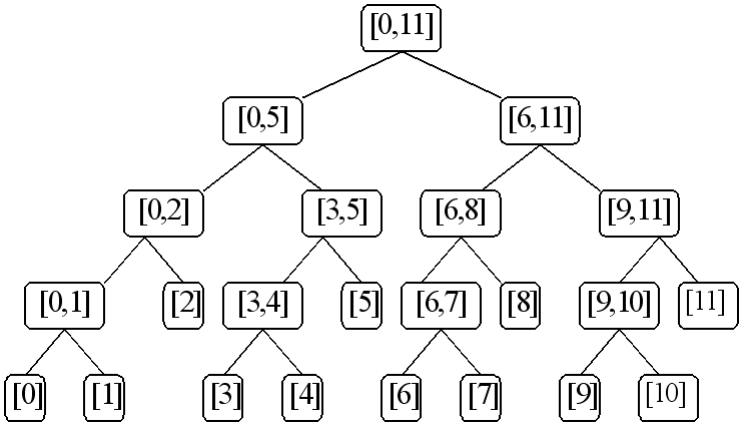


Figura 1: Arbore de intervale

Proprietate

Un arbore de intervale este un arbore binar echilibrat (diferența absolută între adâncimea subarborelui stâng și subarborelui drept este cel mult 1). Astfel adâncimea unui arbore de intervale care conține *N* intervale este $[\log_2 N] + 1$.

Vom prezenta în cele ce urmează operațiile specifice acestei structuri de date.

Actualizarea unui interval dintr-un arbore de intervale

Vom prezenta pseudocodul unei proceduri recursive cu ajutorul căreia se poate insera un interval [*a*, *b*] într-un arbore de intervale *T*(*st*, *dr*) cu rădăcina în nodul *nod*.

Cea mai eficientă metodă de stocare în memorie a unui arbore de intervale este utilizarea unui vector folosind aceeași codificare a nodurilor ca și cea utilizată în cazul *heap*-urilor binare.

```
procedura Actualizare
                (nod, st, dr, a, b)
  dacă a ≤ st și dr ≤ b atunci
    modifică structura auxiliară
                    din nodul nod
  altfel
    mij ← [(st + dr) / 2]
    dacă a ≤ mij atunci
      Actualizare(2 · nod, st,
                      mij, a, b)
    sfârșit dacă
    dacă b > mij atunci
      Actualizare(2 · nod + 1,
              mij + 1, dr, a, b)
    sfârșit dacă
    actualizează structura auxiliară
                    din nodul nod
  sfârșit dacă
sfârșit procedură
```

Interogarea unui interval dintr-un arbore de intervale

Vom prezenta în continuare pseudocodul unei funcții recursive care returnează informațiile asociate unui interval [*a*, *b*] dintr-un arbore de intervale *T*(*st*, *dr*) cu rădăcina în nodul *nod*.

```
funcție Interogare
                (nod, st, dr, a, b)
  dacă a ≤ st și dr ≤ b atunci
    returnează structura
            auxiliară din nodul nod
  altfel
    mij ← [(st + dr) / 2]
```

```
    dacă a ≤ mij atunci
      Interogare(2 · nod, st,
                 mij, a, b)
    sfârșit dacă
    dacă b > mij atunci
      Interogare(2 · nod + 1,
            mij + 1, dr, a, b)
    sfârșit dacă
    returnează structura auxiliară
      din fiul stâng și din fiul drept
  sfârșit dacă
sfârșit funcție
```

Analiza complexității

Vom demonstra în continuare că operațiile prezentate mai sus au ordinul de complexitate $O(\log N)$ pentru un arbore de N intervale. Este posibil ca într-un nod să aibă loc apel atât pentru fiul stâng cât și pentru cel drept. Acest fapt determină un cost adițional doar prima dată când are loc.

După prima "rupere în două", oricare astfel de "rupere" nu va aduce cost adițional, deoarece unul din fii va fi mereu inclus complet în intervalul $[a, b]$. Datorită faptului că înălțimea unui arbore de N intervale este întotdeauna $[\log_2 N] + 1$, ordinul de complexitate al operațiilor prezentate va fi $O(\log N)$.

Necesarul de memorie

Pentru a reține în memorie un arbore de intervale pentru N valori, avea nevoie de $N + N / 2 + N / 4 + N / 8 + \ldots = 2 \cdot N - 1$ locații de memorie (sunt $2 \cdot N - 1$ noduri).

Deoarece arborele nu este complet, trebuie verificat de fiecare dată dacă fiii unui nod există în arbore (această verificare a fost omisă în pseudocodul prezentat anterior), altfel s-ar încerca accesarea de valori din vector care nu există.

Dacă memoria disponibilă este suficientă, se poate declara vectorul care reține arborele de intervale de lungime 2^K astfel încât $2^K \geq 2 \cdot N - 1$, simulând astfel un arbore complet și nefiind necesare verificările menționate mai sus.

Aplicații

În cele ce urmează vom prezenta enunțurile câtorva probleme pentru a căror rezolvare arborii de intervale sunt utili.

Problema #1

Se consideră $N \leq 50.000$ segmente în plan dispuse paralel cu axele Ox și Oy. Să se determine care este numărul total de intersecții între segmente.

În fișierul **segment.in** se află pe prima linie numărul N al segmentelor, iar pe fiecare dintre următoarele N linii se află câte patru numere naturale mai mici decât 50.000, reprezentând coordonatele carteziene ale extremităților fiecărui segment.

Rezultatul se va scrie în fișierul **segment.out**.

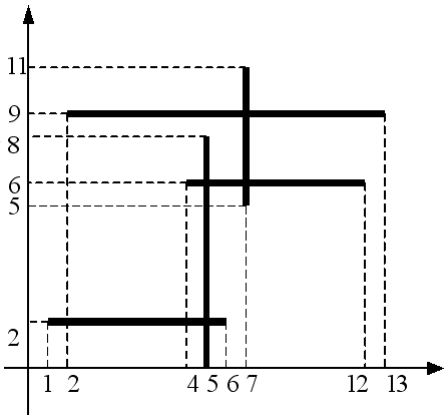
Exemplu

segment.in

```
5
2 9 13 9
4 6 12 6
1 2 6 2
5 0 5 8
7 5 7 11
```

segment.out

```
4
```



Timp de execuție: 1 secundă/test

Algoritmi de "baleiere" (line sweeping)

Folosind cunoștințe generale de geometrie analitică se poate obține un algoritm cu ordinul de complexitate $O(N^2)$, dar acesta nu se va încadra în limita de timp.

Pentru rezolvarea acestei probleme vom folosi o tehnică ce este cunoscută sub numele de "baleiere" (engl. *sweeping*) care este comună multor algoritmi de geometrie computațională.

Pentru operația de baleiere, o *dreaptă de baleiere* verticală, imaginară, traversează mulțimea obiectelor geometrice, de obicei de la stânga la dreapta.

Baleierea oferă o metodă pentru ordonarea obiectelor geometrice, de obicei, plasându-le într-o structură de date, pentru obținerea relațiilor dintre ele.

De obicei, algoritmii de baleiere gestionează două mulțimi de date:

- ***starea liniei de baleiere*** indică relația dintre obiectele intersectate de linia de baleiere;
- ***lista punct-eveniment*** este o secvență de coordonate x, ordonate de la stânga la dreapta de obicei, care definesc pozițiile de oprire ale dreptei de baleiere; fiecare astfel de poziție de oprire se numește ***punct eveniment***; numai în punctele eveniment se întâlnesc modificări ale stării liniei de baleiere.

Pentru unii algoritmi lista punct-eveniment este determinată dinamic în timpul execuției algoritmului.

Soluție

Vom deplasa o dreaptă de baleiere verticală, imaginară, de la stânga la dreapta. Lista punct-eveniment va conține capetele segmentelor orizontale și tipul acestora (extremitate stângă sau extremitate dreaptă) și segmentele verticale.

Pe măsură ce ne deplasam de la stânga la dreapta, când întâlnim o extremitate stângă o inserăm în stările dreptei de baleiere, iar când întâlnim o extremitate dreaptă ștergem extremitatea stângă corespunzătoare din stările dreptei de baleiere.

Când întâlnim un segment vertical, numărul de intersecții ale acestui segment cu alte segmente orizontale va fi dat de numărul capetelor de intervale care se află în stările dreptei de baleiere cuprinse între coordonatele verticale ale extremităților segmentului vertical.

Astfel, stările dreptei de baleiere trebuie să fie reprezentate de o struc-

tură de date pentru care avem nevoie de următoarele operații:
- *Inserează*(y): inserează extremitatea y;
- *Șterge*(y): șterge extremitatea y;
- *Interogare*(y_1, y_2) : determină numărul de extremități cuprinse în intervalul $[y_1, y_2]$

Fie *maxc* valoarea maximă a coordonatelor extremităților de segmente. Folosind un vector pentru a implementa aceste operații vom obține un ordin de complexitate $O(1)$ pentru primele două operații și $O(maxc)$ pentru cea de-a treia.

Astfel, ordinul de complexitate al algoritmului de rezolvare a problemei va fi $O(N \cdot maxc)$ în cazul cel mai defavorabil.

Putem comprima spațiul [0… *maxc*] observând că doar maxim N valori din acest interval contează, și anume extremitățile segmentelor orizontale, astfel reducându-se ordinul de complexitate a celei de-a treia operații la $O(N)$, dar algoritmul va avea ordinul de complexitate $O(N^2)$, ceea ce nu aduce nici o îmbunătățire față de algoritmul trivial.

Această situație ne determină să căutăm o structură de date mai eficientă. O primă variantă ar fi împărțirea vectorului în bucăți de $\sqrt{N}$ elemente, reducând astfel ordinul de complexitate la $O(N \cdot \sqrt{N})$.

În continuare vom prezenta soluția bazată pe arborii de intervale.

Vom folosi un arbore de intervale pentru a simula operațiile, care erau efectuate folosind un vector obișnuit, în timp logaritmic.

Astfel, în fiecare nod al arborelui din intervale vom reține câte extremități există în acel interval.

Primele două operații vor fi implementate folosind procedura `Actualizare` de mai sus pentru intervalul $[y, y]$ în arborele $T(0, maxc)$ și adunând +1, respectiv -1 la fiecare nod actualizat.

Cea de-a treia operație poate fi realizată folosind funcția `Interogare` pentru intervalul $[y_1, y_2]$. Astfel ordinul de complexitate al algoritmului se reduce la $O(N \cdot \log maxc)$.

Folosind aceeași tehnică de "comprimare" a coordonatelor se poate obține ordinul de complexitate $O(N \cdot \log N)$.

În figura 2 este prezentată structura arborelui de intervale, după actualizarea pentru intervalele [2, 2], [6, 6] și [9, 9]. Sunt marcate intervalele care conduc la obținerea numărului de segmente intersectate de primul segment vertical, obținute în urma interogării pentru intervalul [0, 8].

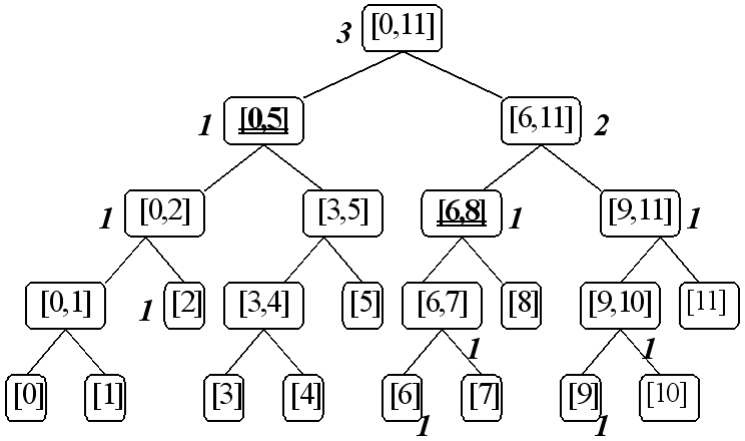


Figura 2: Arbore de intervale pentru problema #1

Problema #2

(preluată de la IOI '98 - Portugalia)

Se considera $N \leq 50.000$ dreptunghiuri în plan, fiecare având laturile paralele cu axele Ox și Oy.

Lungimea marginilor (contururilor) reuniunii tuturor dreptunghiurilor se va numi perimetru.

Să se calculeze perimetrul celor N dreptunghiuri.

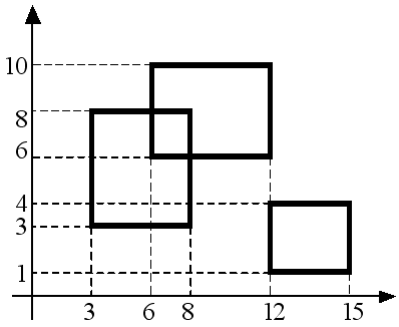
În fișierul **`drept.in`** se află pe prima linie numărul N al dreptunghiurilor, iar pe fiecare dintre următoarele N linii se află câte patru numere naturale, mai mici decât 50.000, reprezentând coordonatele carteziene ale colțurior stânga-sus, respectiv dreapta-jos, ale fiecărui dreptunghi.

Rezultatul va fi scris în fișierul **`drept.out`**.

Exemplu

`drept.in`
```
3
3 8 8 3
6 10 12 6
12 4 15 1
```

`drept.out`
```
44
```



Timp de execuție: 1 secundă/test

Soluție

Folosind un raționament asemănător celui utilizat în cazul primei probleme, constatăm necesitatea unui algoritm de baleiere.

Vom descompune problema în două subprobleme: prima dată calculăm perimetrul folosind doar laturile stânga și dreapta ale dreptunghiurilor pentru a calcula perimetrul pe axa Oy; iar apoi putem să rotim dreptunghiurile și să folosim aceeași procedură pentru a calcula perimetrul pe axa Ox, folosind doar laturile sus și jos ale dreptunghiurilor.

Fiecare latură (stângă sau dreaptă) va fi un punct eveniment. Sortăm laturile crescător în funcție de coordonata orizontală și începem parcurgerea de la stânga la dreapta.

În momentul în care întâlnim o latură stângă marcăm în stările dreptei de baleiere, intervalul de unități ocupat de latură pe axa Oy, iar când întâlnim o latură dreaptă demarcăm

în stările dreptei de baleiere intervalul de unități ocupat de latură.

Între oricare două puncte eveniment consecutive are loc o modificare a perimetrului total pe axa *Oy*.

Astfel, după fiecare actualizare a stărilor dreptei de baleiere se va reține numărul de unități marcate până în prezent (nu se ține cont dacă o unitate este marcată de mai multe ori).

De fiecare dată vom aduna la perimetrul total pe axa *Oy* diferența absolută între numărul de unități marcate până în prezent și valoarea imediat anterioară (înaintea actualizării) a numărului de unități marcate.

Așadar, este necesară o structură de date care se efectueze următoarele operații în timp eficient:

- *Marchează*(*a*, *b*): marchează intervalul [*a*, *b*];
- *Demarchează*(*a*, *b*): demarchează intervalul [*a*, *b*];
- *Interogare*: returnează numărul total de coordonate marcate până în prezent

Putem utiliza un arbore de intervale pentru a obține un ordin de complexitate $O(\log maxc)$ sau $O(\log N)$ pentru primele două operații și $O(1)$ pentru ce-a de treia.

În fiecare nod al arborelui reținem de câte ori a fost marcat intervalul respectiv și câte unități din intervalul respectiv au fost marcate.

Primele două operații pot fi implementate folosind procedura `Actualizare`, iar a treia operație va returna valoarea numărului de unități care au fost marcate din rădăcina arborelui de intervale.

În figura 3 este descrisă structura arborelui de intervale, după marcarea intervalelor [3...8] și [6...10] (un interval $[y_1 \ldots y_2]$ reprezintă intervalul de unități $[y_1, y_2 - 1]$ din arbore).

Problema #3

Se dau $N \leq 100.000$ puncte în plan ale căror coordonate sunt numere naturale mai mici ca 2.000.000.000.

Să se răspundă la $M \leq 1.000.000$ întrebări de forma "câte puncte dintre cele N există în dreptunghiul cu colțul stânga-sus în punctul de coordonate (x_1, y_1) și colțul dreapta-jos în punctul de coordonate (x_2, y_2)?"

În fișierul **`puncte.in`** se vor afla pe prima linie numerele N și M. Pe următoarele N linii se află coordonatele punctelor. Pe următoarele M linii se vor afla câte patru numere naturale reprezentând coordonatele colțurilor dreptunghiurilor.

În fișierul **`puncte.out`** se vor afla M numere naturale reprezentând răspunsurile la întrebări.

Exemplu

`puncte.in`

```
6 1
2 8
5 3
6 5
8 7
8 1
10 10
4 8 9 4
```

`puncte.out`

```
2
```

Timp de execuție: 1 secundă/test

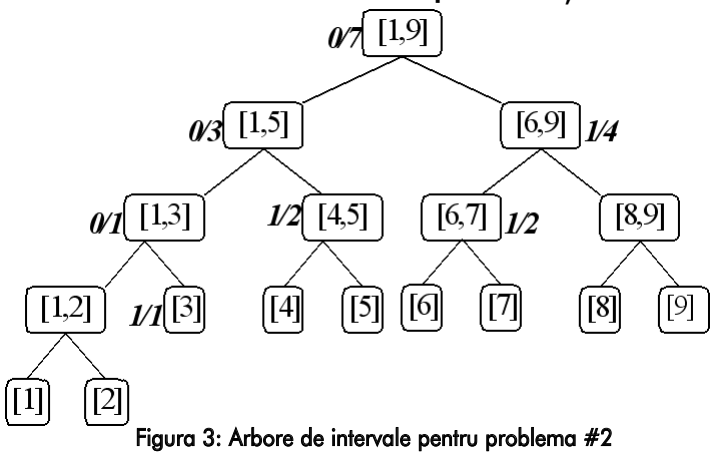
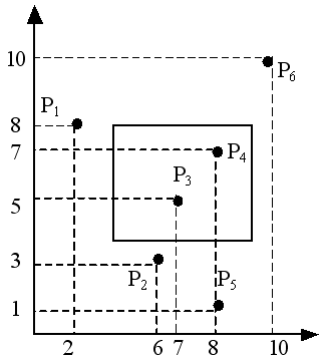


Figura 3: Arbore de intervale pentru problema #2



Soluție

Problema determinării numărului de puncte din interiorul unui dreptunghi este o particularizare 2D pentru problema numită în literatura de specialitate *Orthogonal Range Search*.

Varianta aleasă pentru acest caz, constă într-un arbore de intervale, care permite determinarea numărului de puncte din interiorul unui dreptunghi folosind un algoritm cu ordinul de complexitate $O(\log^2 N)$.

Astfel, se sortează coordonatele x ale punctelor și se construiește un arbore de intervale. Primul nivel al arborelui va conține toate coordonatele x.

Cei doi fii ai rădăcinii vor conține prima jumătate, respectiv a doua jumătate a punctelor (sortate în funcție de coordonata x) ș.a.m.d. Pentru fiecare interval de coordonate x, se mențin sortate coordonatele y corespunzătoare punctelor care au coordonatele x în intervalul respectiv. Astfel, memoria folosită are ordinul $O(N \cdot \log N)$.

Pentru a construi efectiv se folosește o abordare asemănătoare algoritmului de sortare prin interclasare: în fiecare nod, pentru a obține lista de coordonate y ordonate, se interclasează listele celor doi fii (deja ordonate). Când se ajunge într-o frunză, lista nodului este formată dintr-un singur punct.

Pentru fiecare dreptunghi, se efectuează căutarea în arborele de intervale pentru segmentul $[x_1, x_2]$ (deoarece se folosesc doar cele N coordonate x ale punctelor, se vor potrivi capetele acestui interval folosind căutarea binară la cea mai apropiată coordonată x de fiecare extremitate).

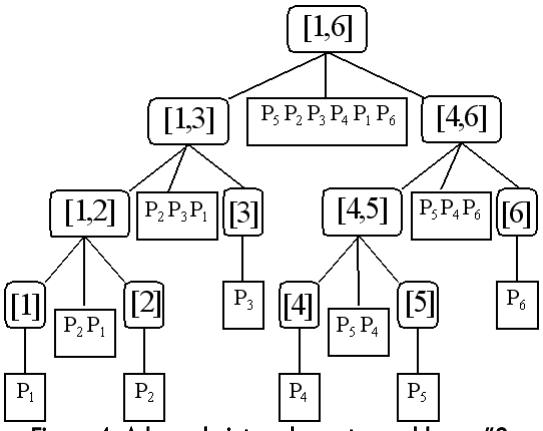


Figura 4: Arbore de intervale pentru problema #3

Pentru fiecare interval din arbore atins, se execută două căutări binare printre coordonatele *y* corespunzătoare coordonatelor *x* din acel interval, pentru a determina câte puncte din intervalul respectiv se află în intervalul $[y_1, y_2]$ (adică în interiorul dreptunghiului).

Structura arborelui de intervale utilizat este prezentată în figura 4.

Ordinul de complexitate $O(\log^2 N)$ poate fi redus, folosind o metodă descoperită independent de *Willard* și *Lueker* în anul 1978. Se observă că pentru fiecare coordonată *y* dintr-un nod, dacă presupunem că se efectuează o căutare binară, atunci putem determina în timpul interclasării poziția din fiecare fiu pe care o va returna căutarea binară pentru aceeași valoare.

Este evident că pentru o valoare *y* dintr-un nod, care a provenit dintr-un fiu în timpul interclasării, există o unică valoare maximă *y'* din celălalt fiu astfel încât $y' \le y$. Așadar, păstrăm pentru fiecare valoare *y* dintr-un nod doi indici care identifică

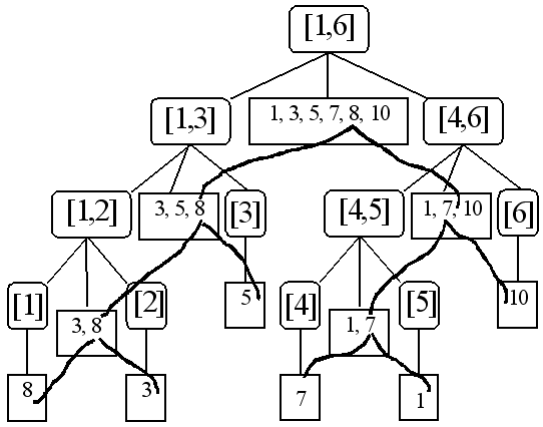


Figura 5: Reducerea complexității pentru problema #3

pozițiile corespunzătoare efectuării unei căutări binare în fii pentru valoarea *y*.

Astfel, în timpul căutării, în loc să efectuăm căutări binare, se pot parcurge acești indici care oferă în timp constant poziția în lista căutată, reducând ordinul de complexitate la $O(\log N)$. Procedeul este ilustrat în figura 5.

Probleme propuse

Prezentăm în cele ce urmează enunțurile a patru probleme care pot fi rezolvate folosind arborii de intervale.

Problema #4

Olimpiada Baltică de Informatică, 2001

Se consideră $N \le 50.000$ dreptunghiuri în plan, fiecare având laturile paralele cu axele *Ox* și *Oy*. Să se determine aria ocupată de reuniunea celor *N* dreptunghiuri.

În fișierul **`drept2.in`** se află pe prima linie numărul *N* al dreptunghiurilor, iar pe fiecare dintre următoarele *N* linii se află câte patru numere naturale mai mici decât 50.000, reprezentând coordonatele carteziene ale colțului stânga-sus, respectiv dreapta-jos ale fiecărui dreptunghi.

Rezultatul se va scrie în fișierul **`drept2.out`**.

Problema #5

Olimpiada Națională de Informatică, Polonia, 2001

Se consideră $N \le 50.000$ puncte care au coordonate numere naturale mai mici decât 50.000. Să se determine poziția în care se poate așeza un dreptunghi de lungime *DX* și lățime *DY*, astfel încât numărul de puncte incluse în dreptunghi să fie maxim.

În fișierul **`puncte2.in`** se află pe prima linie numerele *N*, *DX* și *DY*. Pe următoarele *N* linii se află coordonatele punctelor.

În fișierul **`puncte2.out`** se vor găsi cinci numere naturale reprezentând coordonatele colțurilor stânga-sus și dreapta-jos ale așezării dreptunghiului și numărul maxim de puncte din dreptunghi.

Problema #6

Baraj pentru lotul național, 2003

Se consideră $N \le 100.000$ puncte care au coordonate numere naturale mai mici ca 100.000. Să se determine unde se pot așeza două dreptunghiuri de lungime *DX* și lățime *DY*, fără să se intersecteze, astfel încât numărul de puncte incluse în cele două dreptunghiuri să fie maxim.

În fișierul **`puncte3.in`** se află pe prima linie numerele *N*, *DX* și *DY*. Pe următoarele *N* linii se află coordonatele punctelor.

În fișierul **`puncte3.out`** se vor afla nouă numere naturale reprezentând coordonatele colțurilor din stânga-sus și dreapta-jos ale celor două dreptunghiuri, precum și numărul maxim de puncte incluse în acestea.

Problema #7

USACO, ianuarie 2004

Se consideră $N \le 250.000$ dreptunghiuri în plan, fiecare având laturile paralele cu axele *Ox* și *Oy*, care nu se intersectează și nu se ating, dar pot fi incluse unul în altul. Se numește ***închisoare*** un dreptunghi înconjurat de alte dreptunghiuri. Să se determine numărul maxim de dreptunghiuri de care poate fi înconjurată o închisoare și câte astfel de închisori maxime există.

În fișierul **`inchis.in`** se află pe prima linie numărul *N* al dreptunghiurilor, iar pe fiecare din următoarele *N* linii se află câte patru numere naturale mai mici decât 1.000.000.000, reprezentând coordonatele carteziene ale colțului stânga-sus, respectiv dreapta-jos ale fiecărui dreptunghi.

În fișierul **`inchis.out`** se vor afla două numere naturale: numărul maxim de dreptunghiuri de care poate fi înconjurată o închisoare și numărul acestor "închisori" maxime.