ЗИМНИ МАТЕМАТИЧЕСКИ ПРАЗНИЦИ – РУСЕ' 2006

Задача А1. ПРАЗНИК

Разглеждаме числовата права като път. В някои от целочислените точки по пътя може да има къщи (в една точка може да има много къщи), във всяка от които могат да живеят много хора. Така се е образувал град. Хората могат да се придвижват по този път, но трябва да плащат за това. Цената за да пропътува един човек единица разстояние по пътя е 1 лев. Тъй като това е доста скъпо, хората не пътуват често и надалеч. Градската управа се нуждае непрекъснато от пари, но за да не изглежда прекалено нагло, решила да ги събира като пътни такси. Затова периодически организира градски празници, в които участието е задължително. Всеки празник се прави в определена точка с целочислени координати. Жителите на града сами трябва да си осигурят превоза. Градът, за който става дума, е един динамичен град. В него често се разрушават къщи и хората напускат града или пък нови хора се заселват в новопостроени къщи. Затова преди всеки празник управата внимателно трябва да избере мястото, за да може да си осигури предварително намислената сума. Напишете програма **FEST**, която да получава данни за напусналите и заселилите се граждани и, в зависимост от текущото състояние, да може да отговаря на въпроса: "Къде да се организира празника така, че парите които ще се съберат да са максимално близко до дадено число S?". Ще считаме, че в началото градът е празен.

Вход: Програмата трябва да чете данни от стандартния вход. На Всеки ред ще има или данни за заселване/изселване, или въпрос за търсеното място. Ако данните са за заселване или изселване, тогава редът има вида **1 A B**, където **A** е координатата на къщата, откъдето се изселват –**B** човека (ако **B** < 0) или се заселват **B** човека (ако **B** > 0). Ако редът съдържа въпрос, тогава той има вида **2 S**, където **S** е сумата, която управата иска да събере. Първото число във входа винаги ще е 1. Максималният брой редове на входа е 200 000, от които не повече от 20 000 са въпроси. Координатите **A** ще са в интервала [0,10 000 000], числата **B** – в интервала [-1000,1000], а сумата **S** – в интервала [0,10^15].

Изход: Програмата трябва да изведе резултата на стандартния изход, като за всеки въпрос изведе по един ред. Отговорът се състои от две числа **E P**, където **E** (няма да надвишава по абсолютна стойност 10^8) е координатата на мястото на празника, а **P** е абсолютната стойност на разликата между исканите исканата сума и тази, която ще постъпи в касата от таксите. Ако има повече от един възможен отговор Вашата програма трябва да изведе кой да е от тях.

ПРИМЕР:

Вход:	**Изход:**
1 3 7 1 11 5 2 123 2 36	-4 1 3 4

Решение:

Някои константи които ползвам в решението:

manx = 2^27 – максималната абсолютна стойност на координатата която може да бъде отговор на някой въпрос (това не означава, че непременно има такъв тест)

maxx = 2^24 – максималните координати на които може да има къщи (в тестовете това е по-малко)

mx = 2^17 – броя на листата в дървото ми (трябва да е по голям от максималния брой къщи)

За всеки въпрос ние трябва да намерим оптимална точка от числовата права. Тъй като на всяка точка можем да съпоставим цена (сбора от броя на хората във всяка къща умножен по

разстоянието до дадената точка за всички къщи) можем да отговорим на всеки въпрос просто като обходим цялата права и намерим коя точка е оптималната. Но ако разгледаме тази функция то ще видим, че тя е нещо подобно на парабола – т.е. в двата си края функцията приема големи стойности, които намаляват към средата на графиката. Нека първо си отговорим на въпроса "Къде е тази среда на графиката". Нека сега се намираме в произволна точка **X** от числовата права и нека стойността на функцията в тази точка е **C**. Да се опитаме да намерим каква ще е стойността на функцията в точка **X+1**. Всеки човек, който живее в къща с координати **<=X** ще трябва да заплати **1** лев повече, но от друга страна всеки, който живее в къща с координати **>=X+1** ще трябва да заплати **1** лев по-малко. Следователно стойността на функцията в точка **X+1** е равна на **C+(броя на хората живеещи преди или на X) – (броя на хората живеещи след X)**. И това твърдение е вярно за всяка точка. Нека с **Dx** бележим изменението на стойността на функцията между точки **X** и **X+1**. Лесно се вижда, че **Dx > Dy** за **x>y**. Ако за момент си помислим за функцията **Dx** като за производна то лесно че намерим къде е средата (минимума) на функцията. Там където **Dx** е равно на **0**. За съжаление може да няма такава точка (тъй като работим с целочислени стойности), или пък може да има много такива точки. Затова ето къде се намира средата на функцията. Нека къщите са сортирани. Ако съществува къща **K**, такава че **(броя на хората живеещи в къщи 1..K) = (броя на хората живеещи в къщи K+1..N)**, то тогава функцията е минимална в целия затворен интервал **[K,K+1]**, като това са координатите на къщите. Ако пък няма такава къща то тогава съществува къща **K** такава, че **(броя на хората живеещи в къщи 1..K) > (броя на хората живеещи в къщи K+1..N)** и такава къща, че **K** е минимално. Тогава минимума на функцията е в тази точка (координатата на къщата). Нека отсега нататък средата на функцията наричаме **MIDDLE** като това е или произволна точка от този интервал или къща лежаща в този интервал. Трябва да отбележа, че функцията е линейна между всеки две къщи.

Дотук представих две решения:

1) Тривиалното – за всяка точка от числовата права смятаме стойността и сравняваме с исканата стойност.
2) Смятаме последователно за всички точки като по този начин смятането на стойността на функцията за всяка следваща точка е за константно време.

Сега ще представя третото решение, което използва специална структура данни, която позволява всяко заселване/изселване да се извършва със сложност **O(log(mx))** а на всеки въпрос да бъде отговаряно със сложност **O(log(mx)*log(manx))**.

Индексно дърво

В конкретния случай това представлява пълно двоично дърво с определена височина - **h**. Върховете на това двоично дърво са **2^(h+1)-1**. Основното в тази динамична структура е това, че всеки връх пази някаква информация за определен интервал, а децата му пазят същата информация за 2 подинтервала, които не се застъпват и напълно покриват целия бащински интервал. Примерно бащата може да пази информация за интервала (32 99) а децата за интервали (32 39), (33 99). Може и следните интервали (32,66), (66,99). А може и за следните (32 55), (77,99) какъвто е случаят в нашата задача. Макар и на пръв поглед те да не покриват целия интервал те може да покриват само важните точки от него (в нашия случай координатите, в които има къщи). Удобен начин за представяне на тази структура е масив с големина **2^(h+1)**. Това представяне много наподобява представянето на двоична пирамида в масив. Корена на дървото стои в клетка **1**. Децата му стоят в клетки **2** и **3**. Децата на връх **i** стоят в клетки **2i** и **2i+1**. Познаваме, че даден връх е листо ако е по-голям или равен на **2^h=mx**.

При решаване на задачи с такъв вид индексно дърво много често е задължително преди това да прочетем целия вход и да построим дървото вземайки предвид интервалите, за които ще трябва да пазим информация. Ако не можехме да направим това, тогава трябваше да използваме друга структура (балансирано дърво), която е твърде тежка (особено ако трябва да поддържа операциите които се искат в нашата задача) за състезание.

В задачата, във всеки момент има само определен брой къщи и почти никога не присъстват всички. Но нищо не ни пречи да считаме, че винаги имаме всички къщи построени и когато наистина ги няма просто да ги считаме за пусти (т.е. че съдържат 0 жители). По нататък в решението ще използвам означението **K[l]** за координатата на **l**-тата къща.

Сега да построим дървото за нашата задача. Нека всяко листо пази информация за интервала на една единствена къща (т.е. затворен интервал с дължина 0). Тъй като винаги имаме **2^h** листа, а броят на къщите може да не е толкова то можем да сложим фиктивни къщи с по 0 жители, които изобщо не влияят на задачата. Коренът на дървото ще пази информация за интервала **K[0]..K[2^h-1].** Освен координатите на крайните къщи в дървото си пазя и координатата на средната къща или по точно **K[(0+2^h-1)/2]**. Структурата на един възел на дървото е следната:

```
struct node { int l,m,r;  bigtype sum,w; }
```

За яснота в обясненията ще считам, че дървото се пази в масив **h[]**.Текущия връх (или върха, за който става дума в момента) винаги бележа с **where**.

И за корена е изпълнено **h[1].r=K[0], h[1].l=K[2^h-1], h[1].m=K[2^(h-1)-1]**.

Всяко ляво дете има за ляв край левия край на бащата, а за десен средата на бащата. Всяко дясно дете има за десен край десния край на бащата а за ляв край къщата, която се намира непосредствено след средната къща на бащата. Т.е. ако интервала на бащата са координатите на къщите **(l,r)** то на лявото дете интервалът са координатите **(l,(l+r)/2)** а на дясното – **((l+r)/2+1,r)**. За всяко листо имаме **h[where].l=h[where].m=h[where].r**.

Дотук обясних структурните особености на дървото. Сега е ред на информацията, която се пази във всеки връх:

h[where].sum е броя на жителите живеещи в къщите намиращи се в интервала на върха. Този брой е динамичен – той се променя със времето, когато се заселват или изселват хора (във фиктивните къщи **h[where].sum** винаги ще е 0. По време на изпълнение на програмата винаги следните равенства ще са в сила (за да не е противоречиво дървото). **h[where].sum=h[where*2].sum+h[where*2+1].sum** за всяко **where<mx** (т.е за всеки връх - не лист).

h[where].w е цената, която трябва да заплатят жителите в интервала на върха ако трябва да се съберат в точка **h[where].l**. По време изпълнение на програмата следното равенство винаги трябва да е изпълнено за всеки лист: **h[where].w=0** тъй като никой не трябва да пътува, а за всекли не-лист следното:

```
h[where].w = h[where*2].w + (h[where*2+1].l-h[where].l) *
h[where*2+1].sum + h[where*2+1].w
```

Това е така тъй като **h[where*2].w** е цената за жителите в интервала на лявото дете да пропътуват до **h[where].l**, а **h[where*2+1].w** цената за хората от дясното дете да

пропътуват до **h[where*2+1].l**. Но тези които живеят в дясното дете **h[where*2+1].sum** трябва да пропътуват и разстоянието от **h[where*2+1].l** до **h[where].l** и ето откъде идва третото събираемо.

Сега нека обясня как да променяме структурата при всяко заселване/изселване. Целта ни е да намерим листа, който съдържа къщата, в която се заселват хората. Можем да направим рекурсивна функция, която получава като параметър номер на връх, координата на къща и стойност, която указва с колко човека ще се промени населението на дадената къща. Функцията преценява в кое от двете му деца се намира къщата като сравнява координатата й с **h[where].m** и влиза рекурсивно в съответното дете. При това за всеки посетен връх обновява стойността на **h[where].sum**. Когато се връщаме от рекурсията е добре също така да обновим стойността **h[where].w**. Можем да я обновим като използваме горепосочената формула, но можем и директно на пресметнем каква цена биха допринесли новодошлите хора. За всеки посетен връх пресмятаме новите стойности на информацията за интервала му. И тъй като няма връх, който сме обходили а не сме обходили баща му то новополученото дърво не е противоречиво.

Сега вече можем да обновяваме дървото, но да видим как това може да ни помогне да решим задачата. В нея се иска за дадена цена да намерим такава координата **X** така че стойността на функцията е максимално близка до дадено число.

Можем да вземем интервала **–manx..MIDDLE**. И нека вземем този **MIDDLE**, който ще ни направи функцията в този интервал строго намаляваща. Тъй като е строго намаляваща функция можем да приложим метода на двоичното търсене за да намерим такава точка **X**, която е максимално надясно, но стойността на функцията е **>=** от търсената. Тъй като може да нямаме съвпадение то трябва да пресметнем каква е стойността на функцията в точка **X–1** и от тези двете да изберем по доброто. Сега остава въпроса да намерим за дадено **X** стойността на функцията в тази точка без да обхождаме всички къщи. Точно тук е мястото и на индексното дърво.

Само че за целта ще са ни нужни 2. Като строяхме предното дърво ние за всеки интервал смятахме колко пари ще струва придвижването на всички хора от интервала на върха до левия му край. Но тъй като при нас за дадена точка ще трябва да се придвижват хора и отляво надясно лъм дадена точка, то трябва да направим дърво в което всеки връх пази за интервла си колко пари ще трябват на всички хора живеещи в него за да пропътуват до десния му край. За да не пишем нови функции за работа с това дърво можем да направим симетрично дърво. Т.е. за всяка къща **p** в симетричното дърво слагаме къща с координати **maxx–K[p]**. И обновявамв и двете дървета съответно за къщи с различни координати. И стойността **h[where].w** за 2-рото поддърво е все едно каква е цената за всички хора живеещи в интервала му да се съберат в десния му край на обърнатия интервал.

Сега да се върнем на търсенето на стойността на функцията в дадена точка. Разбивам тази процедура на две. В първата част търся цената за всички хора живеещи вляво от дадената точка (за целта ползвам второто индексно дърво). Във втората част търся цената за всички хора живеещи вдясно от дадената точка (за целта ползвам първото индексно дърво). Тъй като функциите за работа с двете дървета са еднакви, различават се само по точа че тези във второто дърво ги извикваме с обърнати координати, то можем да не мислим че имаме 2 дървета ами да се съсредоточим върху функциите за работа със дървото описано горе.

За да пресметна стойността всички хора живеещи вдясно от дадена точка да се съберат в нея използвам следната рекурсивна функция

find (where, left, right)

Това означава, че искам да намеря колко че струва за всички хора от интервала **[left,right]** да се съберат в точка **left**, като знам, че няма къща в този интервал която не се съдържа в интервала на **where**.

Ако **left>h[where].r** или пък **right<h[where].l** , отговорът е 0 тъй като в дадения интервал не живеят хора.

Ако **left<=h[where].l** и **right>=h[where].r** (интервала **[left,right]** покрива целия интервал на върха **where**) отговорът е **h[where].w+h[where].sum*(h[where].l-left)**. **h[where].w** е цената да се съберат всички хора от интервала **[left,right]** (които всъщност живеят в интервала на **h[where]**) в точка **h[where].l**, но те **h[where].sum** ще трябва да пропътуват още **(h[where].l-left)** докато стигнат до исканата точка.

Ако е възможно за функцията да съдържа къща от интервалите на двете му деца (но не и да покрива целите – този случай беше току що разгледан), **(right>=h[where*2+1].l & left<h[where*2+1].l)** отговорът е:

```
find(where*2,left,h[where*2+1].l-1)  +
find(where*2+1,h[where*2+1].l,right)  +
(h[where*2+1].l-left)*h[where*2+1].sum
```

find(where*2,l,h[where*2+1].l-1) е цената за хората живеещи в сечението на интервалите на лявото поддърво **where*2** и на дадения интервал да се придвижат до точка лефт.

find(where*2+1,h[where*2+1].l,right) е цената на хората в интервала на дясното дете да се придвижат то лявата граница **h[where*2+1].l** на интервала на дясното дете.

(h[where*2+1].l-left)*h[where*2+1].sum – това е цената за същите тези хора **h[where*2+1].sum** трябва да се придвижат от **h[where*2+1].**l до **left**.

Ако пък къщите в дадения интервал лежат изцяло в интервала на дясното поддърво (**left>=h[where*2+1].l**) или пък изцяло в интервала на лявото поддърво (**right<h[where*2+1].l**) тогава мога спокойно да извикам функцията за съответния подинтервал - **find(where*2+1,l,r)** и **find(where*2,l,r)** съответно.

Със извикване на функцията **find(1,X,maxx)** за оригиналното дърво и на функцията **find(1,maxx-X,maxx)** за второто дърво можем да намерим каква цена трябва да платят всички хора заедно за да се придвижат до точка **X**. Този алгоритъм ще работи дори и за **X<0** или пък **X>maxx**. Сложността на функцията **find()** е **O(log(mx))**. Доказателството на този факт оставям за читателя.

И заедно с двоичното търсене сложността на отговарянето на въпрос става **O(log(mx)*log(manx))**. Отбелязвам, че двоичното търсене трябва да се пусне и за другата половина от графиката. Разсъжденията там са аналогични.