# АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА СТЕНА

**Наивно решение – подзадача 1**

Наивното решение на задачата е всяка от функциите *change_wall* и *get_wall_h* да работи със сложност *O(N)*. Това може да се реализира с два масива – в единия се съдържат текущите височини на сегментите от стената, а в другия – максималните височини, които сегментите са достигали в своята „история“ до момента. Промяната на височините на сегментите от даден участък се извършва, като се обходят всички сегменти с номера от *L* до *R*, за всеки от тях се преизчисли текущата му височина и се актуализира във втория масив достигната от него максимална височина. Отговорът на въпроса за сегмент от участък *[L, R]*, достигал през годините максимална височина, се извършва по обичайния начин за търсене на максимален елемент в масив.

Такова решение е реализирано във файл **slowAnyInterval.cpp.** То решава подзадача 1 и носи 9 точки.

**Решение на подзадача 2**

Подзадача 2 е характерна с две неща – първо, че всички въпроси са подредени в края и второ, че всички те питат каква е максималната височина, която е достигал през изминалите години някакъв еднометров сегмент от стената.

Нека всяко викане на функция *change_wall* наречем *change_query*, а всяко викане на функция *get_wall_h– get_query*. Всяко *change_query* се идентифицира с пореден номер, започвайки от 1, съответстващ на поредността на викане на функция *change_wall* (да напомним, че всички такива викания се изпълняват преди виканията на *get_wall_h*).

Нека броят на *change_query* е *M* и нека имаме масив *Q* с *M* елемента, номерирани от 1 до *M*. Първоначално всички елементи на този масив имат стойност 0.

Да си представим, че обхождаме стената сегмент по сегмент, започвайки от този с номер 1 и движейки се към сегмент с номер *N*. Попадайки в сегмент с номер *X*, ние преглеждаме всички *change_query* и за ония от тях, чийто интервал на промяна започва в сегмент с номер *X*, записваме *dh* (стойността, с която съответното *change_query* променя височините на сегментите) в елемента на масива *Q*, който е с индекс, равен на номера на query-то. За тези *change_query*, чийто интервал на промяна е завършил в сегмент с номер *X-1*, в елемента на масива *Q*, който е с индекс, равен на номера на query-то, записваме 0.

Тогава намирайки се в сегмент с номер *T*, в масива *Q* ще има стойности *dh* само в елементите, които съответстват на *change_query*, влияещи на височината на сегмент T. При това те ще бъдат подредени в масива в реда, в който са променяли височината му. Тогава максималната височина, която е достигал сегмент с номер *T* през годините ще бъде равна на най-голямата префиксна сума на масива Q, т.е. на най-голямата от сумите *Q[1], Q[1]+Q[2],….., Q[1]+Q[2]+….+Q[M]*.

Има два момента, които трябва да бъдат „изчистени“ преди тази идея да се превърне в бързо решение.

Единият е как, достигайки до сегмент с номер *X*, бързо да откриваме ония *change_query*, чийто обхват на промяна започва в сегмент *X* или е завършил в сегмент *X-1*. Отговорът е – като поддържаме два масива с данни за *change_query*: единия –

идентификатор на *query*-то и номер на сегмент, ляв край на обхвата му, сортиран по този номер на сегмент; втория – идентификатор на *query*-то и номер на сегмент, десен край на обхвата му, сортиран по този номер на сегмент. Това ще ни позволи, намирайки се в сегмент с номер *X*, чрез двоично търсене да откриваме нужните ни *change_query*.

Вторият момент е, че само с един масив Q няма да достигнем необходимата бързина на намиране на максималната префиксна сума. За целта се нуждаем от структура, която да позволява бързото изпълнение на две операции:

- Промяна на стойността на елемент от масива с някаква стойност *dh.*
- Намиране на стойността на най-голямата префиксна сума на масива след поредната промяна на негов елемент.

Това лесно се реализира с индексно дърво. Сложността на този алгоритъм е *O(N*logK).* Той съществено използва факта, че всички *change_query* са в началото на послевателността от query-та и, поради това, не може да се очаква, че ще може да бъде развит за решаването на подзадачи 3 и 4.

**Решение на подзадачи 2 и 3**

Изложеното решение на подзадача 2 изглежда доста добре, но, за съжаление, то не носи в себе си идеи, които да решат следващите подзадачи.

За да се решат останалите подзадачи се използва техниката **lazy propagation** в интервално дърво. Нормално е тези решения да решават и подзадача 2.

Листата на интервалното дърво съответстват на еднометровите сегменти от стената, а, както се полага в едно интервално дърво, всеки от междинните върхове „отговаря“ за листата от неговото поддърво.

За решаването на подзадачи 2 и 3 във върховете на дървото се пазят следните стойности:

*value* - колко текущият връх допринася към височините на еднометровите стени, които съответстват на листа в поддървото на този връх. Например, височината на еднометров сегмент е равна на сумата от стойностите *value* по пътя от корена на интервалното дърво до листото, съответстващо на този сегмент.

*bestMax* - за листо: максимална стойност на пътя от корена до листото (сума на *value*), която се получила при всичките викания на функцията *change_wall* до текущия момент.

За вътрешен връх *W*: максимална стойност на пътя от корена до *W*, която се е получила при всичките викания на функцията *change_wall* от последния момент когато се е променила стойност в поддървото на *W* до текущия момент.

Това ни позволява да изчислим *bestMax* (за връх *V*) в момента на смяна на стойността на върха *V* (сменяме *value* на *V*) и да актуализираме *bestMax* за преките наследници на *V*. Точно спрямо *bestMax* се прилага техниката lazy propagation - не държим стойностите актуални във всеки един момент, но можем да ги изчислим, когато са необходими.

Тези данни са достатъчни, когато трябва да отговаряме на въпроси за максималната височина, която е достигал даден еднометров сегмент от стената преди последното извикване на функцията *get_wall_h.*

Каква е разликата между подзадача 2 и подзадача 3?

В подзадача 2 първо се изпълняват всички извиквания на функция *change_wall*, т.е. всички изменения на височините на участъци от стената. При първото извикване на

функция *get_wall_h* може да се извърши изчисляване на *bestMax* за всички листа на дървото, т.е. за всички еднометрови сегменти от стената, като стойностите на всички „мързеливи“ променливи се „пробутат“ надолу до всички листа, а на следващите въпроси да се отговаря със сложност *O(1)*. Това решение е реализирано във файл **wall.cpp**. Неговата сложност е *O(K*logN)* и то решава само подзадача 2.

Във файл **wallAnySingle.cpp** е реализирано почти същото решение, но, тъй като фукциите *change_wall* и *get_wall_h* в подзадача 3 се викат в произволен ред, то в *get_wall_h* всеки път се извършва „пробутване“ надолу на стойностите на „мързеливите“ променливи по пътя до конкретното листо. Това решение е също със сложност *O(K*logN)* и решава подзадачи 2 и 3.

**Решение на подзадача 4**

За решаване на подзадача 4 е необходимо да увеличим броя на величините, чийто стойности се пазят във върховете на интервалното дърво (*value* и *bestMax* остават). Новите стойности са:

*maxSumToLeaf* - най-голяма сума на път от вътрешен връх *W* до листо в неговото поддърво, като пътя изключва стойността на *W* в текущия момент. Използва се за изчисляване на максимум за интервал за време от последния момент когато се е променила стойност в поддървото на *W* до сегашния момент. Това е точно bestMax + *maxSumToLeaf*.

*currentBestSol* - максимална стойност на път от корена до някое листо в поддървото на текущия връх. Време - от началото до този момент.
Имплементация - lazy propagation. При актуализация в поддървото на връх *V*, *currentBestSol(V) = max(currentBestSol(V * 2), currentBestSol(V * 2 + 1))*. При актуализация на върха *V*: *currentBestSol(V) = max(currentBestSol(V), bestMax(V) + maxSumToLeaf(V))* - максимум на решението до последната промяна в поддървото на *V*, и след последната промяна в поддървото.

Това решение е реализирано във файл **wallAnyInterval.cpp**, има сложност *O(K*logN)* и решава всички подзадачи.

*Автори: Йордан Чапъров, Руско Шиков*