АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА STORIES

Задачата **Stories** е не много известна задача от интервюта. Както повечето такива задачи, и тя има много на брой различни възможни решения.

O(N * K)
Най-тривиалното решение е за всеки от N-те дни да намерим максималното число в предните K елемента с вложен цикъл. Това решение е банално за имплементация, но би хванало едва около 50 точки, тъй като съществуват значително по-ефективни такива.

O(N*logN)
Съществуват няколко на брой решения със сложност O(N * logK), тук ще покажем три от тях.

Първото се базира на приоритетна опашка. За всеки индекс вкарваме забавността на историята в този ден, заедно със самия ден (като pair) в една приоритетна опашка (вградената priority_queue би свършило страхотна работа). След това, гледаме върха на пирамидата и проверяваме дали неговата позиция не е станала твърде „далеч" от текущата. Ако не е, директно го печатаме. Ако е, го махаме от приоритетната опашка и проверяваме следващия най-голям елемент.

Друга алтернатива е да пазим в мултисет последните K числа. Тъй като стандартните set/multiset в C++ са имплементирани чрез балансирани двоични дървета, имаме лесен начин да питаме за най-големия елемент.

Трето възможно решение се базира на RMQ. Реално тук ни интересува интервал винаги с една и съща големина, затова можем да направим RMQ в сегментно дърво (примерно) с K елемента, като всеки ден обновяваме въпросния индекс по модул K. Ако в началото инициализираме всички индекси с 0, то даже няма да ни се налага да се справяме изрично с първите елементи. Максималното число можем лесно да намерим, като видим максималния елемент в последните K – което е просто елементът на индекс 1 (тъй като пазим само последните K елемента).

И трите решения са със сложност O(N*logK), но първото и второто ползват вградена в езика структура. Това, от една страна е хубаво за нас, тъй като се налага да пишем по-малко код. От друга, те са и по-бавни от третото предложение, тъй като имат скрита относително голяма константа за алокиране на памет и пренареждане на дървото. Тези решения биха хванали около 80-85 точки.

O(N)
Най-ефективните решения са със с линейна сложност (няма как да постигнем нещо по-добро, тъй като трябва да обходим поне по веднъж всички елементи).

Първото възможно решение е подобно на това с приоритетната опашка. Вместо приоритетна опашка, обаче, ще ползваме обикновена такава, или по-скоро двустранна такава (дек). В нея ще държим сортирано подмножество на последните K елемента. Трябва да направим наблюдението, че ако в някои от предходните K позиции сме имали число X, а на текущата имаме числото Y, като Y >= X, то няма

никакъв смисъл повече да пазим X, защото Y хем е по-голямо, хем по-близо до бъдещите позиции, които ще разглеждаме.

Ползвайки това наблюдение, можем да пъхаме елементите в опашката по следния начин (отново като pair с индекса, където се срещат). Нека имаме опашка Q, и текущият елемент, който разглеждаме, има стойност Y. Докато Q не е празно и Y >= Q.front(), ще правим Q.pop_front() (така премахваме елементите, които със сигурност не биха ни помогнали нататък). След това добавяме Y в началото на опашката (Q.push_front(Y)). Така елементите в Q са винаги сортирани, като най-големите са в края на опашката. По някое време, обаче, те биха могли да станат твърде далеч (на разстояние > K) от текущата позиция, в който случай не можем да ги ползваме. Тогава ги pop-ваме от опашката докато стигнем елемент, който е достатъчно близо. Накрая просто го печатаме. Тъй като всеки елемент го вкарваме и изкарваме от опашката точно по веднъж, а всички операции са константни, то цялото решение е O(N).

Друго възможно решение, отново за O(N), би могло да бъде направено чрез бъкети. Нека разделим входните числа на бъкети с големина K (последният бъкет с потенциално по-малка). Нека вътрешно във всеки бъкет запишем за всяко число:

1. Какво е най-голямото число в бъкета, наляво от текущото (в масив fwd())
2. Какво е най-голямото число в бъкета, надясно от текущото (в масив bck())

Така, когато трябва да намерим най-голямото число сред последните K в индекс idx, можем бързо (за O(1)) да намерим какво е най-голямото число в началото на текущия бъкет, до idx (ползвайки fwd()), както и кое е най-голямото число от края на предходния бъкет, вземайки (K – idx % K) елемента (ползвайки bck()). Тоест, ако K = 5, и idx е третият елемент в текущия бъкет, трябва да намерим най-голямото число сред тези три (ползвайки fwd()) и най-голямото число от последните 2 на предходния бъкет (ползвайки bck()).
Тук имаме O(N) за да построим fwd(), още O(N) за да построим bck(), и трети път O(N) за да намерим максималните елементи във всяка позиция. Както и предходното решение, това е O(N), но на практика се оказва, че е по-бързо от това с опашката, тъй като отново няма скритата константа от заделянето на паметта.

Автор: Александър Георгиев