

KthElement (Решение)

Това беше една от хитрите задачи в темата. Нейното решение е лесно за имплементиране и сложно за измисляне.

Главният проблем беше огромният вход. Така де. Кайнд ъф. Макар и само 6 числа, можеше да генерира до 20 милиона такива. И макар на spoj0 да няма memory limit (недостатък на системата), то има time limit, който реших да експлоатирам за компенсация (нях нях) :)

Първо да кажем, че съществува особено лесна имплементация, ако познавате стандартната библиотека. Функцията `nth_element()` върши точно това, което се изисква в задачата. При това го прави бързо (ползва идеята на Merge Sort, но не сортира всички числа). Повече информация как да я ползвате можете да научите тук: http://www.cplusplus.com/reference/algorithm/nth_element/

Ако не сте знаели това, обаче, съществуваше и друг вариант, по който можеше да я решите.

Най-най-очевидното решение е да генерираме числата, да ги сортираме и да върнем K-тото по големина. Всъщност, кажи-речи това се искаше в условието. Това, обаче, изисква $O(N \cdot \log(N))$ време за сортирането и мнооого памет (която, обаче, имаме). Все пак, $O(N \cdot \log(N))$ на тест води до приблизително $20,000,000 * 25 = 500,000,000$ операции на тест. Което е много. Реално зад сортиращия алгоритъм има и константа, която, макар и малка, също влияе забележимо тук. Значи това не става.

Принципно би било окей ако можехме да приложим $O(N)$ сортиращ алгоритъм (който не е базиран на сравнения), като например Counting Sort, но пък входните числа можеха да са до около 1,000,000,000, което правеше това също невъзможно.

Втора идея: Двоично Търсене! Няма проблем да правим повече от един пас над числата. Ако имаме хипотеза какъв е отговорът, то можем да преброим колко числа са стрикто по-малки от него и колко равни на него и така да знаем дали то е K-тото по големина. При това - с константна памет! Найс. Обаче отново биваме спънати от $O(N \cdot \log(M))$, (където M е максималното число, което може да присъства във входа). Този път имаме $20,000,000 * 30 = 600,000,000$. Даже по-зле, макар и тук константата да е по-малка.

Хмммм... Then what?

Ще приложим хитро решение, което комбинира (донякъде) идеята от първото и второто. Вече казахме, че не можем да ползваме Counting Sort, който е един от най-разпространените сортиращи алгоритъма, които не са базирани на сравнения (и, съответно, не е обвързан с долната граница от $O(N \log N)$). Защо, обаче, не ползваме друг подобен алгоритъм? В случая ще приложим Bucket Sort отново за $O(N)$.

Под "бъкет" ще разбираме клетка от масив, която брой колко числа в даден интервал има. Например, ако входните ни числа бяха в $[0, 100)$, можеше да направим 10 бъкета, отговарящи за, съответно, $[0, 9]$, $[10, 19]$, $[20, 29]$, ..., $[90, 99]$. Така както 42, така и 47 биха попаднали в един бъкет (и биха увеличи една и съща стойност - стойността на 4-тата (индексирано от нула) клетка в масива). 9 и 13, от друга страна, биха попаднали в различни бъкети (увеличавайки нулевата и първата клетка, съответно).

Първо обхождане

Разделяме възможните числа на $\sqrt{M}$ бъкета, всеки с големина $\sqrt{M} + 1$. Така, очевидно, можем да разпределим числа с големина до M в бъкетите. За всеки бъкет единствено броим колко числа са попаднали в него, без да пазим кои са те.

След завършването на този първи пас, вече знаем по колко числа попадат във всеки бъкет. Нещо повече, така можем да определим в кой бъкет е търсеното K -то по големина число! Също така знаем относително колко голямо е то, тъй като за да е попаднало в бъкета, неговият размер трябва да е било между неговата долна и горна граница. Сложността тук е $O(N)$.

Второ обхождане

При второто обхождане ни интересуват само тези от числата, които са попаднали в бъкета, където се намира и отговорът. Вътрешно в този бъкет правим Counting Sort на числата, попадащи в него. Сложността тук отново е $O(N)$, защото не сме запазили кои числа са попаднали в този бъкет - само колко на брой са те. Но дори да ги пазехме по някакъв начин, то е възможно всички входни числа да попаднат в един единствен бъкет, така че няма как да избегнем повторното им обхождане в най-лошия случай.

Вече, обаче, не търсим K -тото число в бъкета, ами K' -тото, където $K' = K$ минус броя на числата, в бъкети по-малки от този на K -тото. Намирането на K' -тото по големина число във въпросния бъкет може да стане с $O(\sqrt{M})$. И сме готови!

Общата сложност на алгоритъма е $O(N + \sqrt{M})$. Сложността по памет пък е едва $O(\sqrt{M})$.

Забележете, че генерирането на числата е относително бавно, тъй като трябва да ползваме `long long` и имаме `mod` на всяка стъпка, което е сравнително бавна операция. Дори при само две генерирания на числата е по-добре при първото да ги запишем в масив, като изхабим известно количество памет. Но пък така няма да се налага да ги генерираме втори път и ще си спестим над 30% време! Все пак авторското решение не ползва тази оптимизация за да позволи да минат и "оптимални" по памет и време решения.