АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА БИАТЛОН

Формално тази задача може да бъде формулирана така: нека $P = \{p_1, p_2,....,p_N\}$ е пермутация на числата от 1 до N. Таблица на инверсиите за тази пермутация се нарича масив $T=\{t_1, t_2,.....,t_N\}$, в който t_i е равно на броя на елементите на пермутацията P, които се намират наляво от числото i и са по-големи от i. Иска се по зададена таблица на инверсиите да бъде възстановена пермутацията P. Наистина, чрез своите отговори биатлонистите задават таблицата на инверсиите на пермутацията, която определя крайния ред, в който състезателите са пресекли финалната линия (обърнете внимание на това, че, тъй като всеки състезател се движи с постоянна скорост, то, ако биатлонист с номер k изпревари биарлонист с номер j, не е възможно след това биатлонист с номер j да изпревари биатлонист с номер k).

Това е известна задача от комбинаториката. За нея е доказано, че съществува единствено решение. Въпросът е как да бъде решена за големите стойности на N, които фигурират в условието на задачата.

Най-простият начин е да се образуват последователно (например в лексикографичен ред) пермутациите на числата от 1 до N, за всяка пермутация да се образува таблицата на инверсиите и тази таблица да се сравнява със зададената, докато се получи съвпадение на двете таблици. Между другото, задачата за построяване на таблица на инверсиите по зададена пермутация също преставлява интерес при големи N, но тук няма да се спираме на нея (само може да се каже, че за нейното решаване успешно може да се приложи подходът „разделяй и владей“). Няма да се спираме, защото е ясно, че такова решение ще работи за малки N (подзадача 1) не заради построяването на таблицата на инверсиите, а поради факта, че трябва да се генерират всички пермутации, а техният брой нараства ужасяващо бързо. Този подход решава подзадача 1 и е реализиран в **biathlon-nf.cpp.**

Решение със сложност $O(N^2)$

Класическият алгоритъм в комбинаториката за решаване на задачата, който се среща на много места е следния: запълваме масив P (в който ще възстановяваме пермутацията) с нули. В течение на следващите стъпки наличието на нула в елемент на този масив ще означава, че все още не е намерено числото между 1 и N, което трябва да застане на това място в пермутацията. След това започваме да разглеждаме дадената таблица на инверсиите отляво надясно. Първият елемент t_1 в нея показва колко елемента в пермутацията, които са по-големи от 1, се намират наляво от 1. Тъй като 1 е най-малкото число в пермутацията, това означава, че 1 се намира на позиция t_1+1 в нея. Нататък разсъждаваме по аналогичен начин: нека сме стигнали до t_k. Тъй като се движим отляво надясно, то t_k е броят на елементите, които още не сме разгледали, но за които трябва да осигурим „празни“ места, т.е. елементи равни на 0 в масива P преди в него да запишем числото k. Това дава идеята за алгоритъма: броим t_k нули от началото на масива P (прескачайки елементите, в които вече сме записали числа от пермутацията) и на мястото на следващата нула записваме k. Този алгоритъм е със сложност $O(N^2)$ и решава подзадача 2. Той е реализиран в **biathlon-n2.cpp.**

Решение със сложност $O(N*(logN)^2)$

Следвайки идеята от предното решение, нека се опитаме да направим търсенето на нулевия елемент на масива P, в който да запишем числото k, по-бързо. За целта ще използваме още един масив B с N елемента, в който първоначално ще запишем нули, а

след това, попълвайки елемент с номер j в масива P, в $B[j]$ ще записваме 1. Тогава, ако разглеждаме елемент с индекс m в масива P, броят на нулите до него (включвайки и него самия) ще бъде $m-\sum B[j]$, където j се променя от 1 до m. Сега да се върнем към таблицата с инверсиите. Разглеждайки елемент t_k, ние търсим индекса на елемент в масива P, който е равен на 0 и броят на нулите, до който (включително и нулата в него) е равен на t_k+1. Това може да стане чрез двоично търсене като използваме бърз метод за пресмятане на сумата $\sum B[j]$. За целта може да се използва индексно дърво, евентуално под формата на дърво на Фенуик. Този метод пресмята $\sum B[j]$, където j се променя от 1 до M със сложност $logM$. Самото двоично търсене е със сложност $logN$ и при всяко разделяне на масива на две части трябва да се пресмята сума, така че, за намирането на мястото на едно число, сложността става $O(logN)^2$. Сложността на целия алгоритъм е $O(N*(logN)^2)$ и той е реализиран в **biathlon-nlogn2.cpp**. Този алгоритъм решава и подзадача 3.

Решение със сложност O(N*logN)

Запазвайки идеите за двоично търсене и индексно дърво, можем да направим по-добър алгоритъм. Да организираме двоично индексно дърво с 2^m листа, където $m=logN$, ако $N= 2^m$ или $m=[logN]+1$, ако $2^{m-1}<N<2^m$. Когато N не е точна степен на 2 ще имаме излишък от листа, но това не пречи. Всяко листо ще съответства на елемент от масива P (излишните листа не оказват влияние на алгоритъма). Във всеки връх на дървото ще пазим броя на нулите в отрязъка от масива P, който съответства на множеството от листа, чийто предшественик се явява върхът. В самите листа първоначално ще бъдат записани единици (има по една нула във всеки елемент на масива P). Когато попълним някой елемент на масива P, в съответното листо на индексното дърво ще заменяме единицата с нула с последващо изменение на бройките нули нагоре по дървото (операция със сложност $logN$). Когато търсим индекса на елемент в масива P, такъв че броят на нулите до него да е равен на t_k+1, можем да започнем от корена на индексното дърво и да се придвижваме наляво или надясно, така че накрая да стигнем до листото, в което се получава точно търсеният брой. По този начин за всяко k ще намираме мястото, на което трябва да се запише в масива P за време $O(logN)$. Сложността на този алгоритъм е $O(N*logN)$, той решава и подзадача 4 и е реализиран в **biathlon-nlogn.**

Автор: Руско Шиков