Sorting

	На пълното решение	На частичните решения
Тагове	Разделяй и владей Китайска Теорема на Остатъците	Сортировки Двоично търсене

Анализ

Решения със сортировки (пренебрегващи заявката от 2-ри вид)

Най-естествено е в такава задача да се търси решение с някакъв вид сортировка. По-долу съм описал колко точки изкарват сортировките, които съм пробвал:

- `std::sort` – 5.5 точки
- `std::sort` със `std::unordered_map`, който пази питаните до момента въпроси – 5 точки, много TL-та.
- `std::priority_queue` – 10.25 точки
- `std::make_heap` и `std::sort_heap` – 11.5 точки
- Рандомизиран Quick Sort – 5 точки
- Merge Sort – 15 точки

Решения за $\geq O(N^2)$

Идея за 70% от точките.

Всяко от следващите решения ще има нужда да намери на коя позиция се намира 1 или N. Заради това може да използваме $N-1$ *compare* заявки за да я намерим.

Решение за 20(14) точки

Нека да искаме да отгатнем позицията на поредното число x в редицата. За целта ще проверим дали в пермутацията има числата $x-1$ или $x+1$, $x-2$ или $x+2$, ..., $x-(N-1)$ или $x+(N+1)$. За да направим проверка дали $x-p$ или $x+p$ участват в редицата е нужно да се провери дали разликата на x с друго някое число от пермутацията y се дели на p. Ако $|x-y| \equiv 0\ (mod\ p)$ и $x \neq y$, то $|x-y| \geq p$, следователно задължително има някое число в пермутацията z, за което $|x-z| = p$. Нека d е максималното число, за което $x-d$ или $x+d$ присъстват в редицата. Следователно $x-d-1 \leq 0$ и $x+d+1 \geq N+1$. Също, $x-d=1$ или $x+d=N$, следователно $x=d+1$ или $x=N-d$. Така ако знаем на коя позиция се намира 1 в пермутацията, ние лесно може да проверим дали $x=d+1$ и $x=N-d$ – проверяваме дали $x-1 \equiv 0\ (mod\ d)$.

Постигната сложност: $O(N^3)$

Използвани заявки от първи тип: $N-1$ или 1.

Sorting

Имплементация: `sorting_14p.cpp` или `sorting_20p.cpp`

Решение за 30(21) точки

Всъщност, няма нужда да се правят $N-1$ проверки за всяко p, защото ако за някое p проверката важи, то важи и проверката за $p-1$. Ако пък за някое друго p проверката не важи, тогава за $p+1$ също не важи. Заради това може да се направи двоично търсене.

Постигната сложност: $O(N^2\ log_2\ N)$

Използвани заявки от първи тип: $N-1$ или 1.

Имплементация: `sorting_21p.cpp` или `sorting_30p.cpp`

Решение за 40(28) точки

Решението за 40 точки е много по лесно от решенията за 20 и 30 точки. Нека знаем на коя позиция се намира N. Ще намерим числата постепенно, като първо намерим на коя позиция се намира 1, след това на коя позиция се намира 2, …, на коя позиция се намира $N-1$. Нека сме намерили позициите на числата от 1 до $x-1$. Ние искаме да намерим на коя позиция е x. За да направим това, ние ще пуснем линейно обхождане на пермутацията, като за всяка позиция, за която към момента не сме намерили числото на нея, ще проверим дали разликата ѝ с N се дели на $N-x$. Това ще бъде вярно само за позицията на която се намира x.

Постигната сложност: $O(N^2)$

Използвани заявки от първи тип: $N-1$ или 1.

Имплементация: `sorting_28p.cpp` или `sorting_40p.cpp`

Намиране на позицията на 1 за една *divisible* заявка и $O(N^2)$ време.

Тъй като решаваме задачата за $N \leq 10\ 000$, не би било проблем да намерим позицията на 1 за $O(N^2)$ време. Всъщност, разликата на точно една двойка числа ще се дели на $N-1$, а именно на 1 и N. Ние може да обходим всички двойки позиции и да намерим тази, за която разликата се дели на $N-1$. След това може с една *compare* заявка да разберем кое от двете числа е 1 и кое от двете числа е N съответно.

Пълни решения с „разделяй и владей“ идея

Решение за 70 точки

Какво би било **IATI** без задача с разделяй и владей? Много е естествено да проверим какво може да правим с *divisible* заявката ако използваме $d=2$. Нека си хванем първото число от редицата и проверим с всяко друго число от нея дали разликата им се дели на 2. Така може да разделим числата на четни и нечетни, но няма как да знаем дали първото число е четно или

нечетно. Така при $N = 6$, ще може да разделим позициите на пермутацита на 2 групи, съответно съдържащи $\{1,3,5\}$ и съдържащи $\{2,4,6\}$, но не бихме знаели в какъв ред. Нека също разгледаме какво може да правим с *divisible* заявката с $d = 4$. Забележете, нечетните числа дават остатъци 1 и 3 при деление на 4, а четните – 0 и 2. Така ако приложим същия похват с *divisible* с $d = 4$ за групата на позициите, съдържащи $\{1,3,5\}$ ще може да ги разделим на $\{1,5\}$ и $\{3\}$, а за групата на $\{2,4,6\}$ ще може да я разделим на $\{2,6\}$ и $\{4\}$. Може да приложим и същия алгоритъм за $d = 8$... Виждате ли вече решението?

Нека след такова разбиване сме достигнали до група с равни остатъци при деление на 2^k. Може да разбием групата на 2 групи спрямо остатъците им при деление с 2^{k+1}. Например, нека за $N = 6$ и $p = \{3,2,6,5,4,1\}$ сме получили две групи от позиции $odd = \{1,4,6\}$ и $even = \{2,3,5\}$ и рекурсивно сме ги сортирали да са равни на съответно – $odd = \{6,1,4\}$ и $even = \{2,5,3\}$. Ние няма как да знаем коя група е с четни и нечетни числа от двете, заради това може да проверим дали най-малкото число от едната група е по-малко от най-малкото число от другата група с *compare* заявка (в случая, дали $p_6 < p_2$). Така ще знаем коя група с каква четност е. За да направим владей частта от алгоритъма и да получим сортираните позиции за цялата група, като сме разбрали коя група с каква четност е, то ние просто трябва да редуваме числата от двете групи. В примера, като знаем, че $p_6 < p_2$, най-малкото число в групата ще бъде p_6, второто най-малко число ще бъде p_2, третото най-малко число ще бъде p_1... Така ще получим, че сортираните индекси за цялата група са $\{6,2,1,5,4,3\}$. Като знаем сортираните индекси за цялата пермутация, лесно може да я възстановим.

Постигната сложност: $O(N\ log_2\ N)$

Използвани заявки от първи тип: най-много $N - 1$.

Имплементация: `sorting_70p.cpp`

Решение за 100 точки

Всъщност се оказва, че при владей частта от алгоритъма, не винаги трябва да проверяваме кое е по-малкото от двете най-малки числа от групите. Не е трудно човек да се досети, че двете групи, получени след разбиването на някоя друга група, имат или еднакви дължини (например при $N = 6$ и $d = 2$) или с разлика 1 (например при $N = 3$ и $d = 4$). Забележте, ако групите са с различни дължини, винаги групата с по-голяма дължина ще има по-малък най-малък елемент (Например при $\{1,3,5\}$ – $\{1,5\}$ и $\{3\}$). Ако пък са с равни дължини ≥ 2, числата в групите ще изглеждат в следния вид:

- Първата група: $\{x, 2^{k+1} + x, 2 \times 2^{k+1} + x, \dots, p \times 2^{k+1} + x\}$
- Втората група: $\{2^k + x, 2^{k+1} + 2^k + x, 2 \times 2^{k+1} + 2^k + x, \dots, p \times 2^{k+1} + 2^k + x\}$

Така броят елементи в първоначалната група ще бъде $2p + 2$. Всъщност, нека разгледаме разликата на първия елемент от първата група и последният елемент от втората група, както и разликата на първия елемент от втората група и последният елемент от първата група.

Sorting

Първата разлика:

$$(p \times 2^{k+1} + 2^k + x) - x = p \times 2^{k+1} + 2^k = 2p \times 2^k + 2^k = (2p + 1) \times 2^k$$

Втората разлика:

$$(p \times 2^{k+1} + x) - (2^k + x) = p \times 2^{k+1} - 2^k = 2p \times 2^k - 2^k = (2p - 1) \times 2^k$$

Винаги първата разлика ще се дели на $2p + 1$ (равно на големината на първоначалната група без 1), а втората никога няма да се дели на $2p + 1$. Това е защото $2p + 1$ и $2p - 1$ са нечетни и $p \geq 1$. Така с една *divisible* заявка за $2p + 1$ може да се разбере коя група е с по-малко най-малко число. При $p = 0$ нямаме избор и трябва да направим *compare* заявка.

Постигната сложност: $O(N\ log_2\ N)$

Използвани заявки от първи тип: най-много $\frac{N}{2}$.

Имплементация: `sorting_100p.cpp`

Пълни решения, използващи Китайска Теорема на Остатъците

Когато измислях задачата, първоначално я реших с гореописаната идея, но в последствие се досетих и за решение с китайската теорема на остатъците. За да я решим по този начин, ние ще трябва да намерим на коя позиция се намира 1-цата, заради това ще почнем с решение за 70 точки.

Решение за 70 точки

Нека сме намерили на коя позиция се намира N в пермутацията. Тогава, нека с идеята от решението на решението за 40 точки намерим на кои позиции се намират числата от 1 до 17. Нека искаме да разберем стойността на някое число от пермутацията, което е > 17. Ако разберем остатъците му при деление на съответно 2, 3, 5, 7, 11, 13 и 17, ние чрез китайската теорема ще може да намерим остатъка му при деление на $2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 = 510\,510$, което би било равно на самото число. Въпросът е как намираме остатъка му при деление на тези числа. Нека искаме да намерим остатъка на числото на позиция pos при деление на x. Нека позицията, на което се намира числото i е $numb_i$ за всяко $1 \leq i \leq 17$. Тогава може да питаме x *divisible* заявки, съответно $divisible(pos, numb_i, x)$ за $1 \leq i \leq x$. За точно една от тази заявка ще се върне *true*. Ако се върне *true* за заявката с $numb_x$, то остатъка на числото е 0 при деление на x. В противен случай, ако се върне true за заявка с $numb_i$ за $1 \leq i \leq x - 1$, остатъка на числото ще бъде равен на i.

Постигната сложност: $O(N \times 58)$, където $58 = 2 + 3 + 5 + 7 + 11 + 13 + 17$.

Използвани заявки от първи тип: $N - 1$.

Имплементация: `sortingCRT_70p.cpp`

Решение за 100 точки

Sorting

Единственото нещо, което трябва да се направи, е да се намери позицията на N за по-малко заявки. Всъщност, нека вземем двоичното търсене от решението за 30 точки. Ако си изберем някоя позиция от пермутацията и намерим максималното d за което проверката е важи и намерим с кое число разликата им се дели на d, то това число ще бъде или 1, или N. Като намерим едното от двете числа, ние трябва да намерим другото, като направим обхождане на редицата с *divisible* заявки с $N-1$ и направим една *compare* заявка.

Постигната сложност: $O(N \times (log_2 N + 58))$, където $58 = 2+3+5+7+11+13+17$.

Използвани заявки от първи тип: 1.

Имплементация: `sortingCRT_100p.cpp`

П.П: Не е възможно да се реши задачата без заявки от първия вид. Ако използваме заявките от втория вид, ние не бихме могли да различим $p_1, \dots, p_N$ от $N+1-p_1, \dots, N+1-p_N$. Това е защото разликите остават същите: $|(N+1-p_i)-(N+1-p_j)| = |N+1-p_i-N-1+p_j| = |-p_i+p_j| = |p_i-p_j|$, съответно и отговорите на всяка една заявка *divisible* за всяка тройка числа (i,j,d).

П.П №2: Причината да мисля за решения с китайска теорема на остатъците са решенията на Александър Гатев `gatevCringe_70p.cpp` и `gatevCringe_100p.cpp`. Колегата също даде идеите на решенията за 20 и 30 точки, както и как да намеря минимума в редицата за 1 *compare* заявка. Може да погледнете `gatevCringe` решенията, те пак ползват китайска теорема на остатъците, но са рандомизирани. Има и решение за 60 точки, което се базира пак на разделяй и владей идея и малко по-малко отгатва каноничното разлагане на всички числа.

Автор: Борис Михов

За да продължа да пиша анализи, може да се присъедините към моя Patreon със следния линк.