Анализ на задача permutation

Тагове: сегментно дърво, наблюдения

Първа подзадача

Първата подзадача беше за комуникация със системата.

Втора подзадача

Ограниченията на подзадачата позволяват успешно решение чрез метода на "грубата сила".

Четвърта подзадача

Умело написани решения за първата подзадача успяваха да минат и тази. Все пак целта на подзадачата беше да се подчертае комутативността на операцията bitwise-xor. Независимо колко заявки от тип "Update" извършим, накрая пермутацията ни ще бъде една от следните 4 възможни - началната с всеки елемент "xor"-нат с 0, 1, 2 или 3. Заявки от тип "Query" могат отново да бъдат отговорени чрез идеята от първата подзадача.

Трета подзадача

Тук единствените съществени числа са 1 и 2. Благодарение на миналата подзадача знаем, че вместо да променяме след всеки "Update" редицата, можем да пазим насъбралото се число, по което ще "xor"-ваме. Да го означим с k. Сега за всяко "Query" търсим разстоянието между числата $1 \oplus k$ и $2 \oplus k$.

Тази подзадача и специалната дефиниция на "Query" силно подсказват, че ни интересува не кое число се намира на i-тата позиция, а на коя позиция се намира p_i-тото число. В момента, ако сме си запазили пермутацията в масив $p[i]$, имаме бърз достъп до i-тата позиция(лошо). Заради това ще "обърнем" пермутацията и ще запишем в $inv_p[p[i]] = i$.

Пета подзадача

Тук комбинираме идеите на трета и четвърта подзадача и за да постигнем по-бърза сложност на заявките "Query" ще се възползваме от следното наблюдение: отговорът на заявка $[l, r]$ е разликата между максималната и минималната позиция на число в този интервал. Това ефективно превръща задачата ни в Range Minimum/Maximum Query върху масива inv_p, който дефинирахме в третата подзадача.

Пълно решение

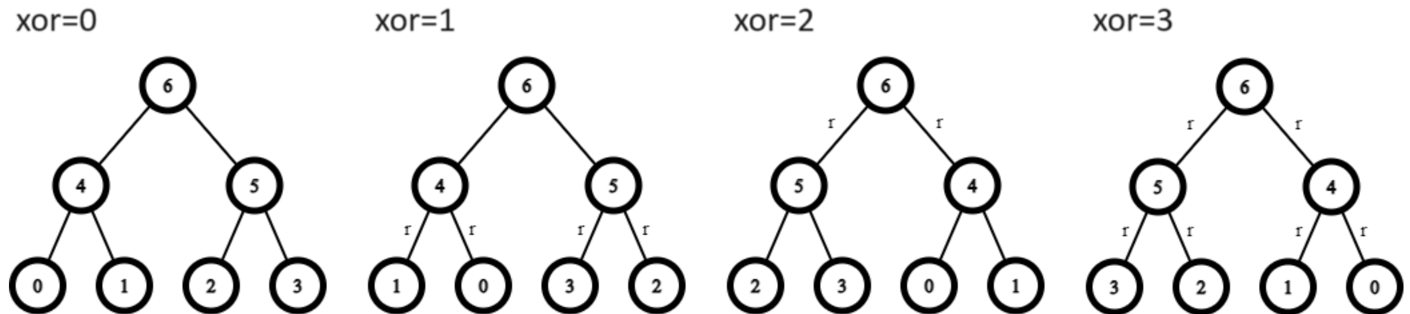
Първо да отбележим, че извършването на "Update x" за $x = 2^{b_1} + 2^{b_2} + ...2^{b_k}$ е идентично като извършването на всички заявки "Update 2^{b_i}", ако b_i са битовете на x. Това е очаквано от побитовите операции, при които случващото се с един бит няма ефект върху другите битове в числото.

Нека забележим какво се случва след заявка от тип "Update 1". Тогава числата 1 и 2, 3 и 4, ... v и $v+1$ и т.н. разменят местата си. Подобен ефект се наблюдава при "Update 2^p" - разменят се позициите на две числа, когато те са напълно идентични в двоичното си представяне с **изключението** на p-тия си бит.

Това на пръв поглед изглежда като непосилна операция за обработване, но тъй като я извършваме на всички числа, можем да се възползваме от структурата на сегментното дърво, построено върху $2^t (= N)$ числа - листата в лявото поддърво на корена отговарят за позициите с 0 на p-тия бит, а в дясното дърво - 1 на p-тия бит. Като продължим надолу по дървото, логиката се задържа - ако върха ни е на дълбочина d в лявото си поддърво той ще има числа с бит 0 на d-тата позиция, а в дясното - бит 1.

Така се оказва, че един “Update 2^p“ всъщност разменя ребрата на всички върхове на някаква константна дълбочина в сегментното дърво. Това все още звучи бавно, докато не се усетим, че може да запазваме и обработваме тези размени “lazily“ - в авторовата имплементация това се случва с един масив `bool lazy[30];`.

На долната рисунка съм означил промените, които причиняват заявките “Update x“ за $x = 0, 1, 2, 3$. С две ребра се разменят, ако имат общ родител и двете са означени с r.



Релевантни материали:

- блог в codeforces.com
- задача B от тук се ползва от подобна идея
- отново подобна задача

Автор: Иван Лупов