Анализ на задача „Битмакс“

Автор: Емил Инджев
Анализ: Енчо Мишинев

В тази задача работим с 63-битови числа и за удобство ще означаваме броя на битовете с $B = 63$.

Подзадача 1 ($N, Q \leq 10^4$)

Прилагаме всяка операция върху всяко число и намираме максимума. Сложността е $O(NQ)$.

Подзадача 2 (само `AND` и `OR`)

Чест подход при побитови операции е да разглеждаме всяка позиция в двоичния запис отделно от останалите. Нека проследим една позиция и нека x е битът на X на тази позиция.

- `AND`: при $x = 1$ нищо не се променя, а при $x = 0$ битът на всяко число става 0.
- `OR`: при $x = 0$ нищо не се променя, а при $x = 1$ битът на всяко число става 1.

Тоест една операция или не променя битовете на позицията, или ги прави еднакви във всички числа. Съответно във всеки момент една позиция е в едно от две състояния:

- *свободна* – никоя операция още не я е променила и битът на всяко число е началният му бит;
- *фиксирана* – битът е един и същ във всички числа и вече не зависи от началните им стойности.

Ключовото наблюдение е, че позиция, която веднъж е станала фиксирана, остава фиксирана завинаги: следващите операции могат само да сменят общата ѝ стойност. Позициите са B, така че в рамките на един тест нова позиция се фиксира най-много B пъти.

Съответно е достатъчно да пазим две числа: P – маска с единици на свободните позиции, и R – стойностите на фиксираните позиции (с нули на свободните). Числото, което в началото е било a, в момента е равно на $(a \texttt{ AND } P) \texttt{ OR } R$. В началото $P = 2^{63} - 1$ и $R = 0$, а операциите ги променят така:

- `AND` с X: $\quad P = P \texttt{ AND } X, \quad R = R \texttt{ AND } X$;
- `OR` с X: $\quad P = P \texttt{ AND } (\texttt{NOT } X), \quad R = R \texttt{ OR } X$.

Отговорът след всяка операция е

$$\max_i (A_i \texttt{ AND } P) \texttt{ OR } R.$$

Стойността $\max_i(A_i \texttt{ AND } P)$ се променя само когато се промени P. Съответно можем след всяко фиксиране на нова позиция да пресмятаме тази стойност наново с едно линейно обхождане на числата, а това се случва най-много B пъти. Сложността е $O(BN + Q)$.

Подзадача 3 (само XOR)

Нужното наблюдение тук е, че няколко поредни операции XOR са равносилни на една, тъй като $(a \texttt{ XOR } X_1) \texttt{ XOR } X_2 = a \texttt{ XOR } (X_1 \texttt{ XOR } X_2)$. Затова е достатъчно да пазим едно число R – XOR на всички стойности X досега – и числото, което в началото е било a, в момента е $a \texttt{ XOR } R$.

Търсим $\max_i(A_i \texttt{ XOR } R)$. Това е класическа задача за двоично префиксно дърво (trie). Строим trie от всички числа, започвайки от най-старшия бит: всеки връх има до две деца – за бит 0 и за бит 1. При заявка слизаме от корена и на всяко ниво се опитваме да получим единица в отговора, като ако битът на R на тази позиция е r, искаме да тръгнем към детето $1 - r$; ако такова няма, тръгваме към другото. Тъй като по-старшите битове са по-важни от всички по-младши взети заедно, този избор винаги е правилен.

Построяването на trie-я е $O(BN)$, а всяка заявка е $O(B)$ – общо $O(B(N+Q))$. Дървото има най-много $BN+1$ върха.

Общият случай и подзадача 4 ($N, Q \leq 1.5 \times 10^5$)

Нека сега са разрешени и трите операции. XOR с бит $x = 1$ обръща бита на всяко число, така че:

- свободната позиция вече съдържа началния бит, евентуално обърнат – но това е еднакво за всички числа: или всички са обърнати, или нито едно;
- фиксираната позиция остава фиксирана – обръща се само общата ѝ стойност.

Следователно наблюдението от подзадача 2 остава в сила: нова позиция се фиксира най-много B пъти. Двете идеи се обединяват, като пазим маската P на свободните позиции и число R, което на фиксираните позиции съдържа общата им стойност, а на свободните показва дали битът е обърнат. Числото, което в началото е било a, в момента е равно на

$$(a \texttt{ AND } P) \texttt{ XOR } R.$$

Операциите променят P и R така (лесно се проверява за един бит):

- XOR с X: $R = R \texttt{ XOR } X$;
- AND с X: $P = P \texttt{ AND } X$, $R = R \texttt{ AND } X$;
- OR с X: $P = P \texttt{ AND } (\texttt{NOT } X)$, $R = R \texttt{ OR } X$.

След всяка операция търсим най-голямото от числата $(A_i \texttt{ AND } P) \texttt{ XOR } R$. Това е точно задачата от подзадача 3, само че за числата $A_i \texttt{ AND } P$ вместо за A_i. Съответно строим trie от тях и питаме за максимален XOR с R.

Числата в trie-я се променят само когато се промени P. Най-простата идея тогава е да строим дървото наново при всяка промяна на P. Това се случва най-много B пъти, така че сложността е $O(B^2N + BQ)$.

Подзадача 5 ($N, Q \leq 4 \times 10^5$)

Тази подзадача е за оптимизирани решения с подобна сложност, както и за по-бързи решения с лоша реализация.

Основната оптимизация е да строим дървото по-рядко: не при всяка новофиксирана позиция, а например на всеки три. Така в дървото има най-много две „остарели“ нива, чиято позиция вече е фиксирана, но в дървото числата все още са разделени по нея. При заявка на такова ниво трябва да продължим и в двете деца, така че обхождаме най-много 4 пътя вместо един.

Това забързва решението значително, защото двете части са с различна сложност. Всички строения заедно са със сложност $O(B^2N)$, а всички заявки заедно – $O(BQ)$. Затова си струва заявките да станат няколко пъти по-бавни, за да строим три пъти по-рядко.

По-генерално, ако строим дървото на всеки k фиксирани позиции, при добре избрано k (като $2^k \approx B/\log_2 B$) общата сложност става $O(B^2N/\log B)$ (при $Q \approx N$), така че на теория получаваме по-добра асимптотична сложност. На практика решението е просто няколко пъти по-бързо, но това е достатъчно за подзадачата. Оптималният избор за нашите ограничения наистина е около $k = 3$ или $k = 4$.

Друга възможна по-малка оптимизация е в дървото да участват само свободните позиции. Ако вече има t фиксирани позиции, строенето прави $N(B - t)$ стъпки вместо NB. Стъпките са около два пъти по-малко, но на практика това само по себе си не е достатъчно.

Друг трик, който помага, е числата да се сортират предварително. Тогава последователните вмъквания минават през близки места в паметта, а и е лесно да се махнат повторенията.

Пълно решение

Вместо да строим дървото наново, ще го променяме. Когато позиция b стане фиксирана, във всички числа A_i `AND` P битът на тази позиция става 0. За дървото това означава следното: при всеки връх от нивото на бит b поддървото на детето „1“ трябва да се *слее* с поддървото на детето „0“, след което върхът остава само с дете „0“.

Сливането на две поддървета с корени u и v става рекурсивно:

- ако едното е празно, резултатът е другото;
- иначе сливаме децата „0“ на u и v, сливаме децата „1“ на u и v, закачаме двата резултата за u, а върха v изтриваме.

Всяко извикване, в което и двете поддървета са непразни, изтрива един връх, а извикване с празно поддърво отнема $O(1)$ и прекратява рекурсията. Нови върхове никога не се създават, а в началото те са най-много $BN + 1$. Следователно всички сливания през целия тест отнемат общо $O(BN)$ време, независимо кога и в какъв ред се фиксират позициите. С други думи, общата сложност на всички модификации в рамките на един тест е $O(BN)$.

За да намираме бързо върховете от дадено ниво, един вариант е при построяването да записваме върховете на всяко ниво в отделен списък. Всяко ниво се обработва най-много веднъж, защото една позиция става фиксирана най-много веднъж.

Заявката е същата като в подзадача 3. На фиксирано ниво всеки връх има само дете „0“, така че на тази позиция в отговора попада точно битът на R – както трябва.

Общата сложност е $O(B(N + Q))$ по време и $O(BN)$ по памет. Дървото може да има до $63 \cdot 6 \cdot 10^5 \approx 3.8 \cdot 10^7$ върха, затова ограничението за памет е голямо.

Алтернативно решение без дърво

Съществува и решение, използващо само сортиран масив и двоично търсене. Идеята е подобна: работим с числата A_i AND P и търсим максимален XOR с R.

Заявка. Нека числата A_i AND P са сортирани в масив. Вървим по позициите от най-старшата към най-младшата и поддържаме интервал $[l, r]$ от масива – числата, които на по-старшите позиции имат точно избраните досега битове. Те наистина образуват интервал, защото в сортиран масив числата с общо начало са съседни. В началото интервалът е целият масив.

- Ако позицията е фиксирана, всички числа имат 0 на нея и интервалът не се променя.
- Ако позицията е свободна, числата в интервала съвпадат на по-старшите позиции, така че първо са тези с бит 0, а след тях тези с бит 1. Границата между двете части намираме с двоично търсене. Искаме частта, в която битът е различен от бита на R – тя дава единица в отговора. Ако тази част не е празна, стесняваме интервала до нея, а иначе го оставяме същия.

Накрая всички числа в интервала са равни на числото, което дава максимума, и отговорът е неговият XOR с R. Правим най-много B двоични търсения, така че една заявка отнема $O(B \log N)$.

Промяна на P. Когато някоя позиция стане фиксирана, нулираме бита на тази позиция във всички числа, но след това масивът може вече да не е сортиран. Най-простата идея е да го сортираме отново. Това се случва най-много B пъти, така че цялото решение е

$$O(B \cdot N \log N + B \cdot Q \log N).$$

Програмата е много кратка и използва само $O(N)$ памет. На теория тя е по-бавна от пълното решение, но работи с малко памет, а сортирането и двоичното търсене са много бързи на практика, така че добре написано такова решение минава всички тестове.

Сортирането може да се замени и с едно линейно обхождане. Когато позиция b стане фиксирана, разглеждаме групите от числа с еднакви по-старши битове – те са съседни в масива. Във всяка група първо са числата с бит b равен на 0, а после тези с бит 1, и всяка от двете части е сортирана по по-младшите си битове. След като нулираме бита, двете части остават сортирани поотделно, така че е достатъчно да ги слеем линейно, както правим при merge sort. Така при фиксиране на позиция можем да сортираме новополучения масив за $O(N)$ и решението става $O(BN + BQ \log N)$.