Ezoteryczne Kartki
Drzewo redukcyjne

Jakub Bachurski

wersja 1.0.1

1 Wstęp

Chcemy wymyślić strukturę, która dla statycznego ciągu umie szybko odpowiadać na zapytania o wartość pewnej operacji łącznej (niekoniecznie odwracalnej) na przedziale ciągu.

2 Idea

Weźmy ciąg a i operację łączną $\circ$. Przyjmijmy $\circ = +$ (operacja to dodawanie), żeby uprościć rozważania.

Nasza struktura będzie miała strukturę drzewa binarnego. Główna idea jest następująca: weźmy korzeń drzewa. Zapiszmy w nim informacje pozwalające na szybkie odpowiadanie na zapytania przechodzące przez środek ciągu a. Jeżeli odpowiadamy na zapytanie, które przechodzi przez środek ciągu, możemy policzyć wynik i go zwrócić. Natomiast jeżeli zapytanie w całości zawiera się w lewej połowie ciągu, to odpowiemy na nie schodząc do lewego poddrzewa, które jest zbudowane w ten sam sposób na lewej połowie. W przeciwnym wypadku skorzystamy z prawego poddrzewa, zbudowanego na prawej połowie ciągu.

$$a_1, \underbrace{a_2, a_3, a_4 \mid a_5, a_6, a_7}_{\text{Zapytanie przecinające środek}}, a_8$$

3 Zapytania

Pozostaje się zastanowić, jak skonstruować ministrukturę odpowiadającą na zapytania, które przechodzą przez środek ciągu.

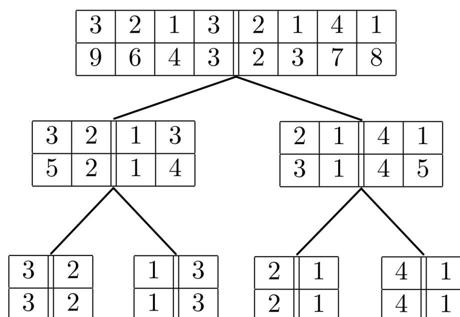
Skorzystajmy z prostej obserwacji: każde zapytanie przecinające środek składa się z sufiksu lewej połowy (lewosufiks) oraz prefiksu prawej połowy (prawoprefiks). Możliwych lewosufiksów i prawoprefiksów jest $O(n)$, więc możemy preprocesować dla nich wyniki. Następnie możemy skorzystać z łączności $\circ$ i złączyć

wyniki lewosufiksu i prawoprefiksu. Oznaczmy powstałą tablicę policzonych wyników r (są to sklejone wyniki lewosufiksów i prawoprefiksów). Sprawdźmy ten pomysł na przykładzie. Wartości używane w zapytaniu $[3, 7]$ są pogrubione.

a	3	2	**1**	**3**	**2**	**1**	**4**	1
r	9	6	**4**	3	2	3	**7**	8

Ministruktura konstruowana jest liniowo i odpowiada na zapytanie w $O(1)$.

Spójrzmy, jak teraz wygląda nasze drzewo. Nazwa „drzewo redukcyjne” pochodzi od tego, że albo wierzchołek umie odpowiedzieć na zapytanie, albo redukuje problem do odpowiadania na zapytanie na którejś połowie ciągu.



Ma ono wysokość $O(\log n)$, bo na każdym poziomie dzielimy ciąg na połowy. Co więcej, konstrukcja zajmuje $O(n \log n)$, bo ministruktura budowana jest w $O(n)$, a każdy element ciągu występuje na każdym poziomie raz.

Mamy jeszcze jeden problem: mimo tego, że istnieje dokładnie jeden wierzchołek, który umie odpowiedzieć na zapytanie (zrobi to w czasie stałym), znalezienie go zajęłoby nam $O(\log n)$ (możemy to robić idąc od korzenia i schodząc do poddrzew, w których zawiera się zapytanie). Gdybyśmy znaleźli ten wierzchołek szybciej, moglibyśmy odpowiadać na zapytania w $O(1)$.

4 Płaszczaki

Najpierw musimy uprościć strukturę drzewa, spłaszczając tablice r do $O(\log n)$ tablic R_i (niech i będzie wysokością poziomu). Aby była ona bardziej regularna, ustalmy, że n jest potęgą dwójki. Trzymając się naszego przykładu $a = (3, 2, 1, 3, 2, 1, 4, 1)$, chcemy otrzymać takie R (wyniki lewosufiksów są jasnoszare, a prawoprefiksów ciemnoszare):

a	3	2	1	3	2	1	4	1
R_2	9	6	4	3	2	3	7	8
R_1	5	2	1	4	3	1	4	5
R_0	3	2	1	3	2	1	4	1

Aby zapis był bardziej czytelny, i-ta wartość na wysokości h to $R_h(i)$. Oznaczmy przedział pewnego zapytania jako $[a, b]$.

Zauważmy, że gdybyśmy znali wysokość H, z której chcemy przeczytać odpowiedź na zapytanie, to wystarczyłoby zwrócić $R_H(a) \circ R_H(b)$. Naszym głównym problemem jest znalezienie H.

Zacznijmy rozważania od obserwacji, że szukamy poziomu, na którym a leży w lewosufiksach, a b w prawoprefiksach. Co więcej, są to lewosufiksy i prawoprefiksy, które wcześniej znajdowały się w r tego samego wierzchołka.

Prowadzi nas to do zastanowienia się nad postacią indeksów, na których zaczyna się nowy blok (lewosufiksów lub prawoprefiksów). Przyjrzyjmy się indeksom początkowym zapisanym binarnie:

h	Indeksy początkowe bloków							
0	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
1	**00**0		**01**0		**10**0		**11**0	
2	**0**00				**1**00			

W h-tym poziomie pogrubione zostały liczby bez ostatnich h bitów. Zauważmy, że w każdym rzędzie pogrubione liczby tworzą kolejne liczby naturalne – zatem nazwijmy je indeksami poziomowymi. Kolejną obserwacją jest fakt, że lewosufiksy i prawoprefiksy, które leżały w tym samym wierzchołku, różnią się tylko ostatnim bitem indeksu poziomowego. Poza tym lewosufiksy mają parzysty indeks poziomowy, a prawoprefiksy nieparzysty.

Pozostaje ostatnia obserwacja: sprowadziliśmy problem do znalezienia takiego poziomu H, że indeks poziomowy a i b różni się tylko na ostatnim bicie – a ma tam 0, a b ma tam 1. Zatem jest to kwestia znalezienia pierwszego bitu, na którym a i b różnią się. Możemy go znaleźć sprawdzając najbardziej znaczący bit $a \oplus b$, gdzie $\oplus$ to xor bitowy. Najbardziej znaczące bity możemy preprocesować w $O(n)$, a xor liczymy w $O(1)$ operatorem `^`.

Wszystkie te rozważania doprowadziły nas do bardzo prostego rozwiązania: H to numer najbardziej znaczącego bitu $a \oplus b$. Tym samym konstruujemy strukturę w $O(n \log n)$, i odpowiadamy na zapytania w $O(1)$.

5 Podsumowanie

Mimo bardziej skomplikowanych rozważań niż przy tabeli potęgowej, drzewo redukcyjne (nazywane *disjoint sparse table*) stanowi przydatną strukturę. Implementacja jest bardzo prosta, a czas działania podobny do tabeli potęgowej.

W kartce o operacjach bitowych jest opisany lepszy sposób znajdowania najbardziej znaczącego bitu: możemy wykorzystać `std::__lg(a ^ b)`.

Disjoint sparse table jest dosyć mało znaną strukturą. Korzystałem z:

- `https://discuss.codechef.com/t/tutorial-disjoint-sparse-table/17404` – opis działania.
- `https://ei1333.github.io/algorithm/sparse-table.html` – przykładowa implementacja.