Sparse Table (разредена таблица)

Съдържание:

Sparse Table е структура от данни, която позволява отговарянето на заявки за обхват. Тя може да отговори на повечето заявки за $O(\log n)$, но истинската и сила е да отговаря на заявки за минимален елемент в интервал (или еквивалентно заявки за максимален елемент в интервал). За тези заявки може да се изчисляват чрез нея за $O(1)$ времева сложност.

Единствения недостатък на тази структура от данни е, че тя може да се използва само за неизменни масиви. Това означава, че масивът не може да бъде променян между две заявки. Ако някой елемент от масива се промени, цялата структура на данните трябва да бъде преизчислена, а построяването ѝ изисква $O(n \times \log n)$ времева сложност.

Интуитивни разсъждения

Всяко неотрицателно число може да бъде еднозначно представено като сбор от намаляващи степени на двойката. Това е вариант на двоично представяне на число. Например $13 = (1101)_2 = 8 + 4 + 1$ или $100 = (1100100)_2 = 2^6 + 2^5 + 2^2$. За числото x може да има най-много $\lceil \log_2 x \rceil$ събираеми.

Поради същата причина, всеки интервал може да бъде уникално представен като обединение на интервали с дължини намаляващи степени на двойката. Например $[2, 14] = [2, 9] \cup [10, 13] \cup [14, 14]$, където интервала е с дължина 13, а интервалите от разбиването му са с дължини $8, 4, 1$ съответно. Отново тук обединението се състои от най-много $\lceil \log_2(\text{length of interval}) \rceil$ броя интервали.

Главната идея зад Sparse Tables е да се преизчислят всички отговори за интервални заявки с дължини равни на степени на двойката. След това може да се отговори на различна заявка за интервал, като се раздели интервалът на интервали с дължини равни на степени на двойката и се потърсят в предварително изчислените отговори, като се комбинират за да получим искания отговор.

Предварително изчисляване

Ще използваме двумерен масив за съхраняване на отговорите на предварително изчислените заявки. $st[i][j]$ ще съхранява отговора за интервала $[i, i + 2^j - 1]$ с дължина 2^j. Размера на двумерния масив ще бъде $MAXN \times (K + 1)$, където $MAXN$ е най-голямата възможна дължина на масива. K трябва да изпълнява условието $K \geq \lfloor \log_2 MAXN \rfloor$, тъй като $2^{\lfloor \log_2 MAXN \rfloor}$ е най-голямата дължина на интервал, която е степен на двойката, която трябва да поддържаме. За масиви с разумна дължина ($\leq 10^7$ елемента), $K = 25$ е добра стойност.

```
int st[MAXN][K + 1];
```

Тъй като интервалът $[i, i + 2^j - 1]$ с дължина 2^j се разбива на интервалите $[i, i + 2^{j-1} - 1]$ и $[\, i + 2^{j-1}, i + 2^j - 1]$ и двата от които са с дължина 2^{j-1}, то може да генерираме таблицата ефективно като използваме динамично оптимиране.

```
for (int i = 0; i < N; i++)
    st[i][0] = f(array[i]);

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = f(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
```

Функцията f ще зависи от вида на заявката. За заявки за суми на интервали ще изчислява сумата, за заявки за минимум ще изчислява минимума.

Времевата сложност на предварителното изчисление е $O(N \log N)$.

Заявки за сума в интервал

За този тип заявки искаме да намерим сумата от всички стойности в интервал. Следователно естествената дефиниция на функцията f е $f(x, y) = x + y$. Може да изградим структурата на данните по следния начин:

```
long long st[MAXN][K + 1];

for (int i = 0; i < N; i++)
    st[i][0] = array[i];

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = st[i][j-1] + st[i + (1 << (j - 1))][j - 1];
```

За да отговорим на заявка за сума на интервала $[L, R]$, итерираме по всички степени на двойката, като започваме от най-голямата. Веднага щом степента на 2^j е по-малка или равна на дължината на интервала ($= R - L + 1$), обработваме първата част от интервала $[L, L + 2^j - 1]$ и продължаваме с останалата част на интервала $[L + 2^j, R]$.

```
long long sum = 0;
for (int j = K; j >= 0; j--) {
    if ((1 << j) <= R - L + 1) {
        sum += st[L][j];
        L += 1 << j;
    }
}
```

Времевата сложност за заявка за сума на интервал е $O(K) = O(\log MAXN)$.

Заявки за минимум в интервал (RMQ)

Това са заявките, за които Sparse Table структурата изпъква. Когато изчисляваме минимума от интервал, няма значение дали обработваме стойност в интервала веднъж или два пъти. Следователно вместо да разделим интервала на множество интервали, можем също да разделим интервала само на два припокриващи се интервала с дължина степен на двойката. Например може да разделим интервала $[1, 6]$ на интервалите $[1, 4]$ и $[3, 6]$. Минимума в интервала $[1, 6]$ е очевидно същия какъвто е минимума в интервалите $[1, 4]$ и $[3, 6]$. Следователно може да изчислим минимума в интервала $[L, R]$ чрез следната формула:

$\min(st[L][j], st[R - 2^j + 1][j])$, където $j = \log_2(R - L + 1)$

Това изисква да може да изчислим $\log_2(R - L + 1)$ бързо. Може да постигнете това, като предварително изчислим всички логаритми:

```
int log[MAXN+1];
log[1] = 0;
for (int i = 2; i <= MAXN; i++)
    log[i] = log[i/2] + 1;
```

След това трябва да изчислим структурата на Sparse Table. Този път дефинираме функцията f по следния начин: $f(x, y) = min(x, y)$.

```
int st[MAXN][K + 1];

for (int i = 0; i < N; i++)
    st[i][0] = array[i];

for (int j = 1; j <= K; j++)
    for (int i = 0; i + (1 << j) <= N; i++)
        st[i][j] = min(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
```

И минимумът в интервала $[L, R]$ може да се изчисли с:

```
int j = log[R - L + 1];
int minimum = min(st[L][j], st[R - (1 << j) + 1][j]);
```

Времевата сложност за заявка за минимум в интервал е $O(1)$.

Подобни структури от данни, поддържащи повече видове заявки

Една от основните слабости на $O(1)$ подходът, обсъден в предишния раздел е, че този подход поддържа само заявки на безсилни (могат да се прилагат многократно, без да променят резултата след първоначалното приложение) функции (idempotent functions: https://en.wikipedia.org/wiki/Idempotence). Т.е. той работи чудесно за заявки за минимален елемент в интервал, но е невъзможно да се отговори на заявките за суми в интервала, като се използва този подход.

Съществуват подобни структури от данни, които могат да се справят с всякакъв тип асоциативни функции и да отговарят на въпроси за интервали в $O(1)$ константно време. Един от тях се нарича Disjoint Sparse Table (https://discuss.codechef.com/t/tutorial-disjoint-sparse-table/17404). Друг подход е Sqrt дървото (https://cp-algorithms.com/data_structures/sqrt-tree.html).

Източник:
[1] Algorithms for Competitive Programming, https://cp-algorithms.com/