Segmentové stromy, sparse table a Fenwickův strom

Petr Ryšavý

23. dubna 2026

IDA, Katedra počítačů, FEL, ČVUT

Myšlenka na dnešní přednášku

- Budeme pracovat s dotazy na rozsah, typicky nás bude zajímat minimum nebo součet z rozsahu hodnot
- Vhodnou organizací dat dosáhneme lepšího času běhu než naivním přístupem

SEGMENTOVÝ STROM

Range Minimum Query

Definice *Mějme pole A, **range minimum query** je dotaz na hodnotu*

$$RMQ(i,j) = \min_{k=i}^{j} A_k.$$

Příklad

Naivní přístup

Naivní přístup

- Naivní přístup spočívá v počítání minima lineárním průchodem - $\mathcal{O}(1)$ konstrukce, ale $\mathcal{O}(n)$ dotaz
- Alternativou je předpočítat všechny intervaly, pak je $\mathcal{O}(1)$ dotaz, ale $\mathcal{O}(n^2)$ konstrukce
- Chtěli bychom nějaký kompromis - *segment tree*

Segment Tree

- Segmentový strom organizuje data podobně jako halda
- Kořen obsahuje minimum celého intervalu
- Levý podstrom obsahuje minimum levé poloviny
- Pravý podstrom minimum pravé poloviny
- Rekurzivně se tato vlastnost opakuje
- Listy obsahují konkrétní prvky

Příklad

Výhody a nevýhody segmentového stromu

- Strom řeší range query v $\mathcal{O}(\log n)$
- Konstrukce v $\mathcal{O}(n \log n)$
- Elegantní implementace, pokud ukládáme podobně, jako haldu
- Výhody se projeví, pokud potřebujeme provádět změny pole

Segmentový strom (S. Halim)

```
class SegmentTree { // the segment tree is stored like a heap array
    private: vector<int> st, A; int n;
        int left (int p) { return p << 1; } // same as binary heap operations
        int right(int p) { return (p << 1) + 1; } // p indexes the node in the tree
        void build(int p, int L, int R) { // O(n)
            if (L == R) // as L == R, either one is fine
                st[p] = L; // store the index
            else { // recursively compute the values
                build(left(p) , L , (L + R) / 2);
                build(right(p), (L + R) / 2 + 1, R );
                int p1 = st[left(p)], p2 = st[right(p)];
                st[p] = (A[p1] <= A[p2]) ? p1 : p2;
        } }
        int rmq(int p, int L, int R, int i, int j) { // O(log n)
            if (i > R || j < L) return -1; // current segment outside query range
            if (L >= i && R <= j) return st[p]; // inside query range
            // compute the min position in the left and right part of the interval
            int p1 = rmq(left(p) , L , (L+R) / 2, i, j);
            int p2 = rmq(right(p), (L+R) / 2 + 1, R , i, j);
            if (p1 == -1) return p2; // if we try to access segment outside query
            if (p2 == -1) return p1; // same as above
            return (A[p1] <= A[p2]) ? p1 : p2; } // as in build routine
    public:
        SegmentTree(const vi &_A) {
            A = _A; n = (int )A.size(); // copy content for local usage
            st.assign(4 * n, 0); // create large enough vector of zeroes
            build(1, 0, n - 1); } // recursive build
    int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); } // overloading
};
```

SPARSE TABLE

Pokud je pole statické?

- Pokud je pole statické a nemění se po volání funkce `build`, jsou segmentové stromy neoptimální
- Existuje efektivnější řešení pomocí dynamického programování s konstrukcí v $\mathcal{O}(n \log n)$ a query v $\mathcal{O}(1)$

Sparse Table

- Pro každou mocninu dvou, která se vejde do délky pole si přepočteme minimum ze všech intervalů této délky
- Máme pole `st[i][j]`, které udává

$$RMQ(j, j + 2^i - 1)$$

- Pole má tedy délku n krát $\log(n)$
- Konstrukce je rekurzivní v $\mathcal{O}(n \log n)$
- Dotaz je v $\mathcal{O}(1)$ - najdeme maximální mocninu 2^i, která se vejde do intervalu, pak

$$RMQ(L, R) = \min\{st[L][L + 2^i - 1], st[R - 2^i + 1][R]\}.$$

- Pro updates nepraktické - museli bychom měnit více než $\log n$ položek

Příklad

FENWICK TREE (BINARY INDEXED TREE)

Range Sum Query

Definice *Mějme pole A, **range sum query** je dotaz na hodnotu*

$$RSQ(i,j) = \sum_{k=i}^{j} A_k.$$

Proč ne segmentové stromy?

- Sparse table lze pro součty snadno použít, jen query je v $\mathcal{O}(\log(n))$
- Pro součty je ale zbytečně komplikovaný - jednodušším řešením je Fenwickův strom
- Segmentový strom se vyplatí, pokud potřebujeme složitější operace, například přičíst hodnotu k celému rozsahu
- Fenwickův strom je naopak neefektivní při počítání minima - používá přefixové sumy, kde pro minimum není inverzní operace jako odečítání ke sčítání.

Fenwick Tree (Binary Indexed Tree, BIT)

- Fenwickův strom kóduje součty na podintervalech
- Na indexu /číslujeme od 1/ i je suma

$$\sum_{k=i-LSB(i)+1}^{i} A_i$$

- Položka $LSB(i)$ vrací index nejméně signifikantního bitu

Příklad

Dotaz na hodnotu

- Pro výpočet prefix-sum dotazu, tj. $RMS(0, j)$ nám postačí sčítat maximálně $\log j$ hodnot
- Využijeme binárního zápisu j
- Za každou jednotku v binárním zápisu přičteme jedno číslo
- Trik: $LSB(i) = i\&(-i)$
- Důvod: $-i$ je bitová negace i, k níž přičteme 1

Příklad

Jak na RSQ

- Postačí odečíst od sebe

$$RSQ(i,j) = RSQ(0,j) - RSQ(0,i-1).$$

Co update?

- Musíme updatovat všechny segmenty, které zahrnují současnou hodnotu
- Opět skáčeme přes LSB

Fenwick Tree (S. Halim)

```
class FenwickTree {
    private: vector<int> ft;
    public: FenwickTree(int n) { ft.assign(n + 1, 0); } // init n + 1 zeroes
    int rsq(int b) { // returns RSQ(1, b)
        int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
        return sum;
    } // note: LSOne(S) (S & (-S))
    int rsq(int a, int b) { // returns RSQ(a, b)
        return rsq(b) - (a == 1 ? 0 : rsq(a - 1));
    }
    // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
    void adjust(int k, int v) { // note: n = ft.size() - 1
        for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v;
    }
};
int main() {
    int f[] = { 2,4,5,5,6,6,6,7,7,8,9 }; // m = 11 scores
    FenwickTree ft(10); // declare a Fenwick Tree for range [1..10]
    // insert these scores manually one by one into an empty Fenwick Tree
    for (int i = 0; i < 11; i++) ft.adjust(f[i], 1); // this is O(k log n)
    printf("%d\n", ft.rsq(1, 1)); // 0 => ft[1] = 0
    printf("%d\n", ft.rsq(1, 2)); // 1 => ft[2] = 1
    printf("%d\n", ft.rsq(1, 6)); // 7 => ft[6] + ft[4] = 5 + 2 = 7
    printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
    printf("%d\n", ft.rsq(3, 6)); // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
    ft.adjust(5, 2); // update demo
    printf("%d\n", ft.rsq(1, 10)); // now 13
} // return 0;
```

References

- Halim, S., Halim, F., Skiena, S. S., & Revilla, M. A. (2013). Competitive Programming 3. Lulu Independent Publish.
- `https://cp-algorithms.com/data_structures/sparse-table.html`
- `https://cp-algorithms.com/data_structures/segment_tree.html`

Děkuji za pozornost.

Čas na otázky!