Задача 3. Максимални суми

Красимир Георгиев

Увод

Още от пръв поглед става ясно, че в задачата се търси изграждане на специализирана структура от данни, която ефективно да изпълнява двете операции. От ограниченията става ясно, че двете операции трябва да се изпълняват на най-много логаритмично време. Нединамичният вариант на тази задача, наречен RMSQ (range maximum-sum segment query problem), е известна задача, която се свежда до обикновено RMQ[1], но методът не е особено удобен за динамичния вариант на задачата. Тук ще предложим ефективно решение на задачата, базирано на сегментни дървета.

Сегментни дървета

Сегментните дървета[2],[3] са структура от данни, позволяваща да се извлича структурирана информация от подинтервали от някаква редица от елементи. Те са една изключително удобна структура, чрез която могат да се реализират множество задачи, например динамично RMQ.

Формално, нека $A = \{a_1, a_2, \dots, a_n\}$ е редица от n елемента. С $A[i, j]$ ще означаваме редицата $\{a_i, a_{i+1}, \dots, a_j\}$. Две операции, които ефективно поддържа сегментното дърво, са промяна на i-тия елемент от редицата, и намиране на композицията $a_i \circ a_{i+1} \circ \dots \circ a_j$, където $' \circ '$ е някаква асоциативна бинарна операция[4],[5].

За да решим задачата със сегментни дървета, трябва да определим как да построим решение за даден интервал, като го разбием на по-малки и ги комбинираме. Ще ни е необходима и допълнителна информация.

Нека u е интервал, който е разбит на два интервала $a, b, u = ab$. Ето каква информация ни трябва за подинтервалите a, b:

- $C(a)$ - сумата на редицата с максиманта сума в интервала a. Това е стойността, която се търси в задачата.
- $S(a)$ - сумата на всички елементи в интервала a.
- $L(a)$ - сумата на редицата, започваща от левия край на a, която е с максимална сума и съдържаща поне един елемент.
- $R(a)$ - сумата на редицата, свършваща в десния край на a, която е с максиманта сума и съдържа поне един елемент.

Лесно се съобразява, че са в сила следните връзки:

$$S(u) = S(a) + S(b)$$
$$L(u) = \max\big(L(a), S(a) + L(b)\big)$$
$$R(u) = \max\big(R(b), S(b) + R(a)\big)$$
$$C(u) = \max\big(C(a), C(b), R(a) + L(b)\big)$$

Така, дървото ще е съставено от елемнти със следната структура:

```
struct node{
	node(num v = 0) : L(v), C(v), R(v), S(v) {}
	void init(num v){
		L = C = R = S = v;
	}
	num L, C, R, S;
};

void compose(const node& a, const node& b, node& c){
	c.C = max(a.R + b.L, max(a.C, b.C));
	c.L = max(a.L, a.S + b.L);
	c.R = max(b.R, b.S + a.R);
	c.S = a.S + b.S;
}
```

Решението на задачата е да се построи сегментно дърво, базирано на горните структура и бинарна операция.

В приложения код е реализирана разновидност на сегментното дърво – пълно двоично сегментно дърво, при което дървото е представено в масив.

```
#include <iostream>
#include <cstdio>
#include <algorithm>
using namespace std;

// inttree - generic complete interval tree implementation
template<typename T, void compose(const T&, const T&, T&)>
struct inttree{
	int size, offset, height;
	T* data, *index;

	// allocates the memory
	void allocate(int size_){
		size = size_;
		height = 0; offset = 1;
		while(offset < size){
			height ++;
			offset <<= 1;
		}
		data = new T[offset<<1];
		index = data + offset;
	}
```

```
// rebuilds the entire tree, based on the index information
void rebuild(){
	for(int i = offset - 1; i > 0; i--)
		compose(data[i<<1], data[1+(i<<1)], data[i]);
}

// updates the tree after single value change
void update(int a){
	int ia = (a + offset)>>1;
	while(ia > 0){
		compose(data[ia<<1], data[1+(ia<<1)], data[ia]);
		ia >>= 1;
	}
}

// interval arithmetic - finds the interval for a given node
inline void get_interval(int i, int h, int& start, int& end){
	int mask = (1<<(height - h)) - 1;
	i <<= (height - h);
	start = i & (~mask);
	end = i | mask;
}

inline void get_interval(int i, int& start, int& end){
	int h = 0, ci = i>>1;
	while(ci) ci >>= 1, h++;
	get_interval(i, h, start, end);
}

// interval search
// from the start to an index - include every left child
// BACK
inline void result(int a, T& res){
	int ia = offset + a;
	compose(res, data[ia], res);
	while(ia > 0){
		if(ia&1) compose(res, data[ia^1], res);
		ia >>= 1;
	}
}

void rec_result(int stree, int h, int a, int b, T& res){
	int left, right, child;
	get_interval(stree, h, left, right);
	// 1. if stree \in [a,b]
	if(a <= left && right <= b){
		compose(res, data[stree], res);
		return;
	}
	if(stree - offset >= 0) return;

	// 2. if it intersects with the left child
	child = stree<<1;
	get_interval(child, h+1, left, right);
	if((a <= left && left <= b) || (left <= a && a <= right))
		rec_result(child, h+1, a, b, res);
```

```
            // 3. if it intersects with the right child
            child++;
            get_interval(child, h+1, left, right);
            if((a <= left && left <= b) || (left <= a && a <= right))
                  rec_result(child, h+1, a, b, res);
      }

      void result(int a, int b, T& res){
            int ia = a + offset, ib = b + offset;
            rec_result(1, 0, ia, ib, res);
      }

};

typedef long long num;

const int max_n = 200100;

struct node{
      node(num v = 0) : L(v), C(v), R(v), S(v) {}
      void init(num v){
            L = C = R = S = v;
      }
      num L, C, R, S;
};

void compose(const node& a, const node& b, node& c){
      c.C = max(a.R + b.L, max(a.C, b.C));
      c.L = max(a.L, a.S + b.L);
      c.R = max(b.R, b.S + a.R);
      c.S = a.S + b.S;
}

inttree<node, compose> rmsq;

int main(){
      int i, n, m;
      num x;
      scanf("%d", &n);

      rmsq.allocate(n);

      for(i = 0; i < n; i++){
            scanf("%d", &x);
            rmsq.index[i].init(x);
      }
      rmsq.rebuild();

      scanf("%d", &m);
      int comm, p, a, b;
      for(int q = 0; q < m; q++){
            scanf("%d", &comm);
            if(comm == 0){
                  scanf("%d%d", &i, &p);
                  rmsq.index[i].init(p);
                  rmsq.update(i);
```

```
		}else{
			scanf("%d%d", &a, &b);
			node res(0);
			rmsq.result(a, b, res);
			printf("%lld\n", max(num(0),res.C));
		}
	}

	return 0;
}
```

Референции

1. On the range maximum-sum segment query problem
2. http://en.wikipedia.org/wiki/Segment_tree
3. http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Segment_Trees
4. http://en.wikipedia.org/wiki/Binary_operation
5. http://en.wikipedia.org/wiki/Associativity