Added: 11 Jun 2008 11:00
EDIT: 25 Oct 2011 21:31

Wood pieces

Segment tree - a data structure that allows efficient (ie, the asymptotic behavior $O(\log n)$) to implement the operation of the following form: finding the amount / minimum of array elements in a given interval ($a[l \ldots r]$where $l$and $r$are input to the algorithm), thus further possible to change the elements of the array: as a change value for one element or elements on the change of the whole array subsegments (i.e., all elements are allowed to assign $a[l \ldots r]$a value, or add all the elements of the array any number).

e number of problems
n above operations with
ons (see. Section "versions
ree is easily generalized to
g the amount / minimum in
dy $O(\log^2 n)$).

e a linear memory: the
o work on the array size n.

Contents [hide]

Description of wood pieces in the base case

First, consider the simplest case of the tree segments - segment tree for sums. If you put the problem formally, then we have an array $a[0..n-1]$, and our tree segments should be able to find the sum of elements from lth to rth (this request amount), as well as handle the change in the value of the specified element in the array, ie, actually respond to the assignment $a[i] = x$(this modification request). Once again, the tree segments should handle both of these queries over time $O(\log n)$.

The tree structure of segments

So, what is a segment tree?

Count and remember somewhere sum of all elements of the array, ie, segment $a[0 \ldots n-1]$. Also calculate the sum of two halves of the array: $a[0 \ldots n/2]$and $a[n/2+1 \ldots n-1]$. Each of these two halves in turn will divide in half, calculate and store the sum of them, then we will divide in half again, and so on until it reaches the current segment length 1. In other words, we start from the segment $[0; n-1]$, and each time we divide the current segment in half (if it has not yet become a segment of unit length), then causing the same procedure on both halves; for each such segment we store the sum of numbers on it.

We can say that these segments in which we considered the amount, form a tree: the root of the tree - the segment $[0 \ldots n-1]$, and each vertex has exactly two sons (except the vertex-leaf, which has a length of the segment 1). Hence the name - "segment tree" (although the implementation is usually no tree is clearly not built, but more on that below in the implementation section).

So, we have described the structure of the tree segments. Immediately, we note that it has a **linear dimension** , namely, contains less $2n$vertices. This can be understood as follows: the first level of the tree segments contains one node (segment $[0 \ldots n-1]$), the second layer - in the worst case, the two peaks, the third level in the worst case there will be four peaks, and so on, until the number of vertices is reached n. Thus, the number of vertices in the worst case estimated amount $n + n/2 + n/4 + n/8 + \ldots + 1 < 2n$.

It is worth noting that $n$other than powers of two, not all levels of the tree segments are completely filled. For example, when $n = 3$ the left son of the root is a segment $[0 \ldots 1]$having two children, while the right son of the root - the segment $[2 \ldots 2]$being a sheet. No particular difficulties in implementing it is not, but nevertheless it should be borne in mind.

The height of the tree is of the segments $O(\log n)$- for example, because the length of the segment is at the root of the tree n, and when you go to one level down the length of the segments is reduced by about half.

Building

The process of constructing the tree segments for a given array $a$can be done effectively as follows, from bottom to top: first write down the values of the elements $a[i]$in the corresponding leaves of the tree, then on the basis of these count values for the vertices of the previous level as the sum of the two leaves, then similarly calculate values for another level, etc. Convenient to describe this operation recursively: we run a procedure for constructing from the root segments, and the procedure of construction, if it is not caused by leaf, causing themselves from each of the two sons and summarizes the calculated values, and if it is caused by the sheet - it simply writes themselves the value of this element of the array.

Asymptotics tree construction segments will be so $O(n)$.

Request amount

We now consider the request amount. Are input two numbers $l$and r, and we have for the time $O(\log n)$to calculate the sum of the numbers on the interval $a[l \dots r]$.

To do this, we will build a tree down the segments using to calculate the response previously counted amounts on each node of the tree. Initially, we get up in the root of the tree segments. Let's see in which of his two sons gets cut request $[l \dots r]$(recall that the sons of the root segments - it stretches $[0 \dots n/2]$and $[n/2+1 \dots n-1]$). There are two possibilities: that the segment $[l \dots r]$ gets only one son of the root, and that, on the contrary, the segment intersects with two sons.

The first case is simple: just move on to that son, which is the length of our request, and we apply the algorithm described here to the current node.

In the second case we can not remain other choice but to go first to the left child and find the answer to the query in it, and then - go to the right child, find the answer in it and add it to our account. In other words, if the left son was cut $[l_1 \dots r_1]$, and the right - a segment $[l_2 \dots r_2]$(note that $l_2 = r_1 + 1$), then we move on to the left child request $[l \dots r_1]$, and the right - with the request $[l_2 \dots r]$.

Thus, the amount of processing of the request is a **recursive function** that calls itself every time a son from the left or from the right (without changing the boundaries of the query in both cases), or by both at once (at the same time sharing our request for two relevant sub-query). However, recursive calls are not always going to do that if the current request coincided with the boundaries of the segment in the current top of the tree segments, then as a response will return the precomputed value of the sum on this segment recorded in the tree segments.

In other words, the calculation of the query is a descent of the tree segments, which is distributed to all the necessary branches of a tree, and for fast work has been used to count the sum of each segment in the segment tree.

Why is **the asymptotic behavior** of this algorithm will be $O(\log n)$? To do this, look at each level of the tree segments as maximum lengths could visit our recursive function in the processing of a request. It is argued that such segments could not be more than four; then, given the estimate $O(\log n)$for the height of the tree, we obtain the required asymptotic behavior of the running time.

We show that this estimate of the four segments is true. In fact, at the zero level of the tree query affected only vertex - the root of the tree. Next on the first level recursive call in the worst case is divided into two recursive calls, but it is important here is that these two queries calls will coexist, ie the number of l''query in the second recursive call is one more than the number of r'query in the first recursive call. It follows that at the next level, each of the two calls could produce two more recursive call, but in this case, half of the non-recursive queries will work, taking the required value from the top of the tree segments. Thus, every time we will have no more than two really working branches recursion (we can say that one branch is close to the left edge of the request, and the second branch - to the right), but only the number of affected segments could not exceed the height of the tree segments multiplied by four, i.e. it is a number $O(\log n)$.

Finally, you can lead a working understanding of the amount of the request: the input segment $[l \dots r]$is divided into several subsegments, the answer to each of which has already calculated and stored in the tree. If you do it the right way partition, thanks to the tree structure of segments required number of subsegments will always be $O(\log n)$that and gives the efficiency of the wood pieces.

Update request

Recall that the update request is received at the input index $i$value and x, and rearranges the tree segments so as to conform to the new value $a[i] = x$. The request must also be performed during $O(\log n)$.

This is more than a simple query compared with the request counting amount. The fact that the element $a[i]$involves only a relatively small number of vertices of the segments: namely in $O(\log n)$the tops - one on each level.

Then it is clear that the update request can be implemented as a recursive function: it is passed to the current top of the tree lines, and this function performs a recursive call from one of his two sons (of which contains the position $i$in its segment), and after that - counts the value of the sum in the current node in the same way as we did in the construction of wood segments (i.e., the sum of the values for both the current node sons).

Implementation

The main follow- up time - is how **to keep** the tree in memory segments. For simplicity, we will not keep the tree in an explicit form, and we use this trick: we say that the root of the tree has a number of 1his sons - the numbers $2$and 3their sons - rooms 4on 7, and so on. It is easy to understand the correctness of the following formula: if the vertex is numbered i, then let her son left - is the pinnacle of a number $2i$, and the right - with the number $2i+1$.

This technique greatly simplifies the programming of tree segments - now we do not need to be stored in the memory tree structure segments, and only make any array for sums on each segment of the tree segments.

One has only to note that the size of the array in such a numbering is not necessary to put $2n$and $4n$. The fact that this numbering does not work perfectly when $n$not a power of two - then there are missing numbers, which does not correspond to any tree nodes (actually numbering behaves just as if it $n$would be rounded up to the nearest power of two) . This does not create any difficulties in implementation, but leads to the fact that it is necessary to increase the size of the array to $4n$.

Thus, the segment tree, we **keep** just as an array $t[]$, the size of four times the size $n$of the input data:

```
int n, t[4*MAXN];
```

The procedure for **constructing the tree segments** for a given array $a[]$is as follows: it is a recursive function, it is passed to the array $a[]$, the number of $v$the current top of the tree, and the boundaries $tl$and $tr$the segment corresponding to the current top of the tree. The main program must call this function with parameters $v = 1, tl = 0, tr = n-1$.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

Further, the function to **query the amount** also represents a recursive function in the same way that the information is transferred to the current top of the tree (ie, the number of v, tl, tr which in the main program should pass $1, 0, n - 1$ respectively), and in addition to this - also border l and r the current request. In order to simplify this code fukntsy always makes two recursive calls, even if actually need one - just extra recursive call will request, in which $l > r$ it is easy to cut off additional check at the beginning of the function.

```
int sum (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return sum (v*2, tl, tm, l, min(r,tm))
                + sum (v*2+1, tm+1, tr, max(l,tm+1), r);
}
```

Finally, **a modification request** . He just passed information about the current top of the tree segments, and additionally, the index changing element, as well as its new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = new_val;
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = t[v*2] + t[v*2+1];
        }
}
```

It is worth noting that the function update is easy to make non-recursive, because it tail recursion, ie branching never occurs: one call can generate only one recursive call. When non-recursive implementation, speed can grow several times.

Other **optimizations** it is worth mentioning that the multiplication and division by two is necessary to replace Boolean operations - it is also slightly improves the performance of the tree segments.

Version of sophistication wood segments

Segment tree - a very flexible structure, and allows you to make generalizations in many different directions. Let's try to classify them below.

More advanced features and inquiries

Improvements wood segments in this direction may be quite obvious (as in the case of the minimum / maximum value instead of the amount) and a very, very nontrivial.

Minimum / maximum

Little to change the terms of the problem described above: instead of requesting the sum will produce now request the minimum / maximum on the interval.

Then the segment tree for such a task does not differ from the tree segments described above. Just need to change the method of calculating $t[v]$ in functions build and update, as well as the calculation of the returned response function sum (to replace the summation on the minimum / maximum).

Minimum / maximum value and the number of times it occurs

The task is similar to the previous one, but now in addition to the maximum required and return the number of its occurrences. This problem arises in a natural way, for example, in the solution of using wood pieces such task: to find the number of the longest increasing subsequence in the specified array.

To solve this problem at each node of the tree segments will store a pair of numbers: in addition to the maximum number of its occurrences in the corresponding interval. Then the construction of the tree we should just two such pairs obtained from the sons of the current node, get a pair for the current node.

The union of two such pairs one stands out in a separate function because this operation will have to produce and modify the query, and the query search for a maximum.

```
pair<int,int> t[4*MAXN];

pair<int,int> combine (pair<int,int> a, pair<int,int> b) {
        if (a.first > b.first)
                return a;
        if (b.first > a.first)
                return b;
        return make_pair (a.first, a.second + b.second);
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_pair (a[tl], 1);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

pair<int,int> get_max (int v, int tl, int tr, int l, int r) {
        if (l > r)
                return make_pair (-INF, 0);
        if (l == tl && r == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        return combine (
                get_max (v*2, tl, tm, l, min(r,tm)),
                get_max (v*2+1, tm+1, tr, max(l,tm+1), r)
        );
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                t[v] = make_pair (new_val, 1);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

Search greatest common divisor / least common multiple

Ie we want to learn to look for NOD / NOC all the numbers in a given segment of the array.

It's pretty interesting generalization tree segments obtained in exactly the same way as trees segments for the amount / minimum / maximum: simply stored in each node of the tree NOD / NOC all the numbers in the corresponding segment of the array.

Counting the number of zeros, search kth zero

In this task, we want to learn to respond to the request a predetermined number of zeros in the segment array, and finding a request k-th element zero.

Again slightly modify the data stored in the tree segments: will now be stored in an array of $t[]$the number of zeros that occur in their respective segments of the array. Understand how to maintain and use the data in the functions build, sum, update, - thus we solved the problem of the number of zeros in a given segment of the array.

Now learn how to solve the problem of finding the position of $k$the zero-th entry in the array. To do this we go down the tree segments, starting with the root and each time moving in a left or right child, depending on which of the segments is required kth zero. In fact, to understand what we need to pass a son, just look at the value recorded in the left son: if it is greater than or equal to k, the need to move to the left son (because it has at least a segment $k$of zeros), but otherwise - go to the right child.

In the case of the implementation can be cut off when $k$there is no zero-th, even when entering a function returned as a response, for example -1.

```
int find_kth (int v, int tl, int tr, int k) {
        if (k > t[v])
                return -1;
        if (tl == tr)
                return tl;
        int tm = (tl + tr) / 2;
        if (t[v*2] >= k)
```

```
                return find_kth (v*2, tl, tm, k);
        else
                return find_kth (v*2+1, tm+1, tr, k - t[v*2]);
}
```

Search prefix array with a given sum

The challenge is this: is required by this value $x$quickly find such $i$that the sum of the first $i$element of the array $a[]$is greater than or equal to x(assuming that the array $a[]$contains only non-negative numbers).

This problem can be solved by binary search, calculating every once in a sum for a particular prefix array, but it will lead to the solution of the time $O(\log^2 n)$.

Instead, you can use the same idea as in the previous paragraph, and look for the desired position of a descent on the wood turning every once in a left or right child, depending on the value of the sum in the left son. Then the response to the search request would be one such descent of the tree, and therefore, will be carried over $O(\log n)$.

Search subsegment with the maximum amount

Is still given to the input array $a[0 \dots n-1]$, and receives requests (l, r), which means: to find a subsegment $a[l' \dots r']$that $l \le l'$, $r' \le r$and the sum of the length of $a[l' \dots r']$the maximum. Request for modification of individual elements of the array are allowed. Array elements may be negative (and, for example, if all the numbers are negative, the best subsegments will be empty - on it is zero sum).

This is a very non-trivial generalization of tree lengths obtained as follows. Will be stored in each node of the tree segments of four values: the amount on this interval, the maximum amount of all the prefixes of this segment, the maximum amount of all suffixes, as well as the maximum amount subsegment on it. In other words, for each segment of the tree segments response to it already predposchitan, as well as additional responses counted among all segments abutting against the left boundary of the segment, as well as among all the segments that are limited to the right border.

How do you build a tree segments such data? Again, get to it with a recursive point of view: if the current node for all four values in the left son and son in law had already counted, count them now for the summit. Note that the answer is in the very top:

- a response in the left son, which means that the best subsegment in the current top fits into the segment of the left son,
- a response in the right son, which means that the best subsegment in the current top fits into the segment of the right son,
- or the maximum amount of the suffix in the left son and the maximum prefix in the right son, which means that the best subsegment is its beginning in the left son, and the end - to the right.

So, the answer to the current node is the maximum of these three variables. Recalculate the maximum amount for the same prefix and suffix even easier. We present the implementation of a function $\mathrm{combine}$that, given two structures data, containing the data on the left and right sons, and that returns the data in the current top.

```
struct data {
        int sum, pref, suff, ans;
};

data combine (data l, data r) {
        data res;
        res.sum = l.sum + r.sum;
        res.pref = max (l.pref, l.sum + r.pref);
        res.suff = max (r.suff, r.sum + l.suff);
        res.ans = max (max (l.ans, r.ans), l.suff + r.pref);
        return res;
}
```

Thus, we learned how to build a tree segments. Hence it is easy to obtain and implement query modification: as in the simple tree segments, we do recalculation of values in all segments of changed the tree tops, which all use the same function $\mathrm{combine}$. To calculate the values in the tree leaves as an auxiliary function $\mathrm{make\_data}$, which returns a structure dataas calculated by a single number val.

```
data make_data (int val) {
        data res;
        res.sum = val;
        res.pref = res.suff = res.ans = max (0, val);
        return res;
}

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = make_data (a[tl]);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}

void update (int v, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
```

```
                t[v] = make_data (new_val);
        else {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
                t[v] = combine (t[v*2], t[v*2+1]);
        }
}
```

It remains to deal with the response to the request. To do this, we also, as before, we go down the tree, thereby breaking the query interval $[l \dots r]$ for a few subsegments, coincides with the segment tree segments, and merge the answers to them in a single answer to the whole problem. Then it is clear that the work is no different from ordinary wood work pieces, rather than just have a simple summation / minimum / maximum values use the function combine. The below implementation is slightly different from the implementation of the query sum: it does not allow for cases where the left boundary of l the query exceeds the right border r (otherwise any unpleasant incidents - what structure data comes back when the queue is empty segment? ..).

```
data query (int v, int tl, int tr, int l, int r) {
        if (l == tl && tr == r)
                return t[v];
        int tm = (tl + tr) / 2;
        if (r <= tm)
                return query (v*2, tl, tm, l, r);
        if (l > tm)
                return query (v*2+1, tm+1, tr, l, r);
        return combine (
                query (v*2, tl, tm, l, tm),
                query (v*2+1, tm+1, tr, tm+1, r)
        );
}
```

Saving the entire subarray in each node of the tree segments

This is a separate subsection, standing apart from the rest, since at each node of the tree segments, we will not keep some concise information on this subsegments (sum, minimum, maximum, etc.), and **all** elements of the array, which lie in the subsegments. Thus, the root of the tree segments will store all elements of the array, the left son of the root - the first half of the array, the right son of the root - the second half, and so on.

The simplest version of the application of this technique - where each node of the tree segments stored sorted list of all the numbers found in the corresponding interval. In more complex versions are not stored lists, and any data structures built on these lists (set, map etc.). But all these methods have in common is that each node of the tree segments stored some data structure that has a memory size of the order of the corresponding segment.

The first natural question arising when considering trees segments of this class - this is **the amount of memory consumed** . It is argued that if each node of the tree contains a list of all segments found on this segment of numbers, or any other data structure size of the same order, the sum of all segment tree will occupy $O(n \log n)$ memory. Why is this so? Because each number $a[i]$ falls into $O(\log n)$ pieces of wood segments (not least because the height of the tree segments there $O(\log n)$).

So, despite the apparent extravagance of the tree segments, it consumes memory is not much more than the usual wood segments.

Below described are some typical uses such a data structure. It should be noted immediately clear analogy trees segments of this type with **two-dimensional data structures** (in fact, in a sense, this is a two-dimensional data structure, but with a rather limited possibilities).

Search the smallest integer greater than or equal to the specified value, the specified interval. No modification request

Required to respond to requests from the following: (l, r, x) that means finding the minimum number in the interval $a[l \dots r]$ that is greater than or equal to x.

We construct a segment tree, in which each vertex will store the sorted list of all the numbers appearing on the corresponding interval. For example, the root will contain an array $a[]$ in sorted order. How to build a segment tree as efficiently as possible? To do this, come to the problem, as usual, in terms of recursion: if the left and right children of the current node, these lists have already been built, and we need to build this list for the current node. In this formulation, the question becomes almost obvious that this can be done in linear time: we just need to merge two sorted lists into one that is done in one pass through it with two pointers. C ++ users even easier, because the merging algorithm is already included in the standard library STL:

```
vector<int> t[4*MAXN];

void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = vector<int> (1, a[tl]);
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
                merge (t[v*2].begin(), t[v*2].end(), t[v*2+1].begin(), t[v*2+1].end(),
                        back_inserter (t[v]));
```

```
        }
}
```

We already know that the constructed thus segment tree will take up $O(n \log n)$memory. And with such an implementation time of its construction also has value $O(n \log n)$- because every list is constructed in linear time with respect to its size. (By the way, here there is an obvious analogy with the algorithm **merge sort** : but here we keep the information from all stages of the algorithm, not just the outcome.)

Now consider the **response to the request** . Let's go down the tree, as does the standard response to a request in the tree lines, breaking our segment $a[l \dots r]$on several subsegments (some $O(\log n)$units). It is clear that the answer to the whole problem is minimized among the responses to each of these subsegments. Let us understand now how to respond to a request for one such subsegments, coinciding with a vertex of the tree.

So we came to some vertex of the tree segments and want to find the answer to it, ie, find the smallest number greater than or equal to this x. To do this we just need to perform a **binary search** on the list, count in this top of the tree and return the first number on the list, is greater than or equal to x.

Thus, the answer to the query in one subsegments takes place $O(\log n)$, and the entire request is being processed at the time $O(\log^2 n)$.

```
int query (int v, int tl, int tr, int l, int r, int x) {
        if (l > r)
                return INF;
        if (l == tl && tr == r) {
                vector<int>::iterator pos = lower_bound (t[v].begin(), t[v].end(), x);
                if (pos != t[v].end())
                        return *pos;
                return INF;
        }
        int tm = (tl + tr) / 2;
        return min (
                query (v*2, tl, tm, l, min(r,tm), x),
                query (v*2+1, tm+1, tr, max(l,tm+1), r, x)
        );
}
```

Constant INFequal to some large number, certainly more than any number in the array. It carries the meaning of "response in a given segment does not exist."

Search the smallest integer greater than or equal to the specified value, the specified interval. Permitted modification request

The task is similar to the previous one, but now resolved modification requests: assignment process $a[i] = y$.

The decision is also similar to the solution of the previous problem, but instead of simple lists at each node of the tree segments, we will keep a balanced list, which allows you to quickly search for the required number, delete it and insert a new number. Given that the general number in the input array may be repeated, the best choice is a data structure STL $\mathrm{multiset}$.

The construction of such a tree segments occurs about the same as in the previous problem, but now we must unite not sorted lists, and $\mathrm{multiset}$that will lead to the fact that the asymptotic behavior of building to deteriorate $n \log^2 n$(although, apparently, red-black trees allow to merge the two trees in linear time, but the STL does not guarantee).

Response to **a search request** has practically equivalent to the code above, but now $\mathrm{lower_bound}$need to call on $t[v]$.

Finally, **a modification request** . To handle it, we have to go down the tree by making changes to all $O(\log n)$lists containing affect a component. We simply remove the old value of this element (not forgetting that we do not need to remove it all together with repetitions of this number) and insert the new value.

```
void update (int v, int tl, int tr, int pos, int new_val) {
        t[v].erase (t[v].find (a[pos]));
        t[v].insert (new_val);
        if (tl != tr) {
                int tm = (tl + tr) / 2;
                if (pos <= tm)
                        update (v*2, tl, tm, pos, new_val);
                else
                        update (v*2+1, tm+1, tr, pos, new_val);
        }
        else
                a[pos] = new_val;
}
```

Processing this request also occurs during $O(\log^2 n)$.

Search the smallest integer greater than or equal to the specified value, the specified interval. Acceleration using the technique of "partial cascading"

Improve response time to the search time $O(\log n)$by applying the technique **of "partial cascading"** ("fractional Cascading").

Partial cascading - this is a simple technique that helps to improve the work of several binary search being conducted by the same value. In fact, the response to the search request is that we divide our task into several subtasks, each of which is then solved by the number of binary search x. Partial cascading allows you to replace all the binary search for one.

The simplest and most obvious example is the partial cascading **following problem** : there are several sorted lists of numbers, and we should find in each list the first number is greater than or equal to the specified value.

If we solve the problem "head" that would have to run a binary search on each of these lists, that if a lot of these lists, it becomes a very important factor: if the entire list k, the asymptotic behavior happens $O(k \log(n/k))$where n- the total size of all the lists (asymptotic behavior is because the worst case - when all lists are approximately equal in length to each other, ie equal n/k).

Instead, we could combine all of these lists into one sorted list in which each number $n_i$will keep a list of position: the position in the first list, the first number is greater than or equal to n_ia similar position in the second list, and so on. In other words, for each occurrence of the number we store at the same time the number of results binary search on it in each list. In this case, the asymptotic behavior of the answer to the query is obtained $O(\log n + k)$, which is much better, but we are forced to pay a large consumption of memory: namely, we need $O(nk)$memory.

Appliances partial cascading goes further in solving this problem and is working memory consumption $O(n)$at the same time respond to the request $O(\log n + k)$. (To do this, we keep not one big list length n, and come back to $k$the list, but with each list contains every second element from the following list, we again have with each number to record its position in both lists (current and next), but it will continue to effectively respond to the request: we do a binary search on the first list, and then go to these lists in order, each time passing in the following list using predposchitannyh pointers, and taking one step to the left, thereby taking into account that half the following list of numbers was not taken into account).

But we in our application to the tree line segments **do not need** the full power of this technique. The fact that the current list contains all of the top, which can occur in the left and right sons. Therefore, to avoid a binary search through the list of his son, it is sufficient for each list in the tree segments for each count the number of its position in the list of left and right sons (more precisely, the position of the first day, less than or equal to the current).

Thus, instead of the usual list of all the numbers we keep a list of triples: the number itself, the position in the list of the left son, the position in the list, right son. This will allow us for $O(1)$the list to determine the position of the left or right child, instead of doing a binary list on it.

The easiest way to apply this technique to the problem when there is no modification request - then these positions are simply numbers and counting them in the construction of the tree is very easy within the algorithm merge two sorted sequences.

In the event that allowed modification requests, everything becomes more complicated: these positions now be stored in the form of iterators inside $\mathrm{multiset}$, and when you request an update - the right to decrease / increase for those items for which it is required.

Anyway, the problem has already been reduced to net realizable subtleties, but the basic idea - replacing $O(\log n)$a binary search binary search through the list in the root of the tree - fully described.

Other possible areas of

Note that this technique implies a whole class of possible applications - everything is determined by the structure of the data selected for storage in each node of the tree. We have examined applications using $\mathrm{vector}$and $\mathrm{multiset}$, while generally used may be any other compact data structure: Other segment tree (this is a little discussed below in the section on multivariate trees segments), Fenwick tree , Treap etc.

Update on the interval

We have considered only a problem when the modification request affects only one element of the array. In fact, the segment tree allows queries that apply to whole segments of contiguous elements, with fulfill these requests during the same time $O(\log n)$.

The addition of the segment

To begin consideration of such trees segments with the simplest case: modification request is the addition of all the numbers in some subsegments $a[l \dots r]$of a number x. Read request - continue reading the value of a number $a[i]$.

To make a request for the addition of efficient will be stored in each node of the tree segments as necessary to add all the numbers of this segment as a whole. For example, if a request comes in, "added to the entire array $a[0 \dots n-1]$number 2 ", we will deliver at the root number 2. Thus we will be able to process a request for the addition of any subsegments effectively, rather than to change all $O(n)$values.

If a request comes now read the value of a number, it is sufficient to go down the tree, summing all encountered on the way the values recorded in the tree tops.

```
void build (int a[], int v, int tl, int tr) {
        if (tl == tr)
                t[v] = a[tl];
        else {
                int tm = (tl + tr) / 2;
                build (a, v*2, tl, tm);
                build (a, v*2+1, tm+1, tr);
        }
}

void update (int v, int tl, int tr, int l, int r, int add) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] += add;
```

```
        else {
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), add);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, add);
        }
}

int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return t[v] + get (v*2, tl, tm, pos);
        else
                return t[v] + get (v*2+1, tm+1, tr, pos);
}
```

Assigning the interval

Suppose now that modification request is assigned to all the elements of a certain length of $a[l \dots r]$some value p. As a second request will be considered read the array values $a[i]$.

To make modifications to the whole segment will have at each node of the tree segments stored, whether painted this piece entirely in any number or not (and if painted, the store itself is a number). This will allow us to do **"retarded" update** tree segments: the modification request, we, instead of changing the values in the set of vertices of segments, changing only some of them, leaving flags "painted" for the other segments, which means that this whole segment together with its subsegments to be painted in this color.

So, after the query modification segment tree becomes, generally speaking, irrelevant - there were shortfalls in some modifications.

For example, if the request arrived modification "to assign the entire set $a[0 \dots n-1]$some number ", the tree segments we will only change - mark the root of the tree that he painted entirely in that number. The rest of the top of the tree will remain unaltered, even though it is actually the tree should be painted in the same number.

Suppose now that in the same tree segments came second modification request - to paint the first half of the array $a[0 \dots n/2]$at any other number. To handle such a request, we need to paint the entire left child of the root in this new color, but before you do that, we must deal with the root of the tree. Subtlety here is that should remain in the tree that the right half is colored in the old color, and currently no tree in the right-half data is not stored.

Output is as follows: make **pushing** information from the root, ie if the root of the tree was painted in any number, the color in the number of its right and left a son, and from the root to remove this mark. After that, we can safely paint left child of the root, without losing any relevant information.

Summing up, we obtain for any queries with the tree (request modification or reading) while descending the tree, we should always do the pushing of information from the current node in both of her sons. You can understand it so that when descending the tree we use lagging modification, but only as much as necessary (not to worsen with the asymptotic behavior $O(\log n)$).

When implemented, this means that we need to make a function pushthat will be transmitted top of the tree lines, and it will produce pushing information from this vertex in both her sons. Call this function should be at the very beginning of the function query (but do not call it from the leaves, because of the push plate information is not necessary, and nowhere else).

```
void push (int v) {
        if (t[v] != -1) {
                t[v*2] = t[v*2+1] = t[v];
                t[v] = -1;
        }
}

void update (int v, int tl, int tr, int l, int r, int color) {
        if (l > r)
                return;
        if (l == tl && tr == r)
                t[v] = color;
        else {
                push (v);
                int tm = (tl + tr) / 2;
                update (v*2, tl, tm, l, min(r,tm), color);
                update (v*2+1, tm+1, tr, max(l,tm+1), r, color);
        }
}

int get (int v, int tl, int tr, int pos) {
        if (tl == tr)
                return t[v];
        push (v);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return get (v*2, tl, tm, pos);
        else
                return get (v*2+1, tm+1, tr, pos);
}
```

Function getcould be implemented in another way: do not do it delayed the update, and immediately return a response as soon as it enters the top of the tree segments, entirely painted in a particular color.

The addition of the interval, the maximum request

Now let modification request will again request the addition of all numbers a subsegment of the same number, and read request is to find the maximum in some subsegments.

Then at each node of the tree segments will have to additionally store up to all this subsegments. But subtlety here is how to recalculate the values.

For example, suppose there was a request "added to the entire first half, i.e. $a[0 \dots n/2]$, number 2 ". Then it will be reflected in the tree record number 2in the left child of the root. How to calculate the new value is now high on the left, and his son at the root? Here it becomes important not to get confused - what the maximum is stored in the top of the tree: maximum without considering adding on top of all this, or considering it. You can choose any of these approaches, but the main thing - to consistently use it anywhere. For example, if the first approach, the maximum will be obtained at the root of a maximum of two numbers: the maximum in the left son, plus the addition of the left son, and the son of a maximum in the right plus the addition in it. In the second approach is at the root of the maximum will be obtained as the addition of a root plus a maximum of the peaks in the left and right sons.

Other Destinations

There were considered only the basic application segments trees in problems with modifications on the segment. The remaining tasks are obtained based on the same ideas that are described here.

It is only important to be very careful when dealing with pending modifications: it must be remembered that even if the current top we have "pushed" a modification is pending, then the left and right sons, most likely, have not done so. Therefore it is often necessary to call is pushalso on the left and right children of the current node, or else to carefully consider the pending modifications to them.

The generalization to higher dimensions

Segment tree is generalized quite naturally on the two-dimensional and multi-dimensional case at all. If the one-dimensional case we broke array indexes into segments, the two-dimensional case will now first break all on the first index, and for each segment on the first index - to build an ordinary tree segments for the second index. Thus, the basic idea of the solution - a segment tree for insertion of the second index into the wood pieces on the first index.

Let us illustrate this idea by the example of a specific task.

The two-dimensional segment tree in its simplest form

Dana rectangular matrix $a[0 \dots n-1, 0 \dots m-1]$, and enter search queries amount (or minimum / maximum) on some podpryamougolnikah $a[x_1 \dots x_2, y_1 \dots y_2]$and requests modifications of individual elements of the matrix (ie, queries of the form $a[x][y] = p$).

So, we will build a two-dimensional tree segments: first segment tree in the first coordinate (x), and then - for the second (y).

To **the process of building** more understandable, it is possible to forget that the original two-dimensional array has been, and leave only the first coordinate. We will construct the usual one-dimensional segment tree, working only with the first coordinate. But as the value of each segment, we will not record a certain number, as in the one-dimensional case, and the whole tree segments: ie at this point we remember that we have more and the second coordinate; but since at this point already recorded that the first coordinate is a segment $[l \dots r]$, we actually work with the band $a[l \dots r, 0 \dots m-1]$, and it builds a tree segments.

We present the implementation of operations for constructing a two-dimensional tree. It actually consists of two separate units: the construction of the tree segments in the coordinate x($\mathrm{build\_x}$) and the coordinate y($\mathrm{build\_y}$). If the first function is almost no different from the usual one-dimensional tree, the latter is forced to deal separately with the two cases: when the current segment in the first coordinate ($[tlx \dots trx]$) has unit length, and when - a length greater than one. In the first case, we simply take the required value from the matrix $a[][]$, and the second - combine the values of two tree lengths of the left and right son, son-coordinate x.

```
void build_y (int vx, int lx, int rx, int vy, int ly, int ry) {
        if (ly == ry)
                if (lx == rx)
                        t[vx][vy] = a[lx][ly];
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        else {
                int my = (ly + ry) / 2;
                build_y (vx, lx, rx, vy*2, ly, my);
                build_y (vx, lx, rx, vy*2+1, my+1, ry);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void build_x (int vx, int lx, int rx) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                build_x (vx*2, lx, mx);
                build_x (vx*2+1, mx+1, rx);
        }
        build_y (vx, lx, rx, 1, 0, m-1);
}
```

This segment tree takes still linear memory space, but with a more constant: $16nm$memory cells. It is clear that it is constructed in

the above procedure build_xalso in linear time.

We now turn to the **query processing** . Respond to a two-dimensional query will on the same principle: first break request to the first coordinate, and then when we got to the top of some wood pieces in the first coordinate - initiate a request from the relevant sections of the tree on the second coordinate.

```
int sum_y (int vx, int vy, int tly, int try_, int ly, int ry) {
        if (ly > ry)
                return 0;
        if (ly == tly && try_ == ry)
                return t[vx][vy];
        int tmy = (tly + try_) / 2;
        return sum_y (vx, vy*2, tly, tmy, ly, min(ry,tmy))
                + sum_y (vx, vy*2+1, tmy+1, try_, max(ly,tmy+1), ry);
}

int sum_x (int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {
        if (lx > rx)
                return 0;
        if (lx == tlx && trx == rx)
                return sum_y (vx, 1, 0, m-1, ly, ry);
        int tmx = (tlx + trx) / 2;
        return sum_x (vx*2, tlx, tmx, lx, min(rx,tmx), ly, ry)
                + sum_x (vx*2+1, tmx+1, trx, max(lx,tmx+1), rx, ly, ry);
}
```

This function works in the time $O(\log n \log m)$since she first down on a tree in the first coordinate, and each passed the top of the tree - makes a request of a conventional wood pieces on the second coordinate.

Finally, consider the **modification request** . We want to learn how to modify the tree segments in accordance with the change of the value of any element $a[x][y] = p$. It is clear that changes will occur only in the top of the first wood segments that cover the coordinate x(and they will $O(\log n)$), and for trees segments corresponding to them - the changes will be only those tops that cover the coordinate y(and there is $O(\log m)$). Therefore, the implementation of the modification request will not be much different from the one-dimensional case, but now we will first down in the first coordinate, and then - on the second.

```
void update_y (int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new_val) {
        if (ly == ry) {
                if (lx == rx)
                        t[vx][vy] = new_val;
                else
                        t[vx][vy] = t[vx*2][vy] + t[vx*2+1][vy];
        }
        else {
                int my = (ly + ry) / 2;
                if (y <= my)
                        update_y (vx, lx, rx, vy*2, ly, my, x, y, new_val);
                else
                        update_y (vx, lx, rx, vy*2+1, my+1, ry, x, y, new_val);
                t[vx][vy] = t[vx][vy*2] + t[vx][vy*2+1];
        }
}

void update_x (int vx, int lx, int rx, int x, int y, int new_val) {
        if (lx != rx) {
                int mx = (lx + rx) / 2;
                if (x <= mx)
                        update_x (vx*2, lx, mx, x, y, new_val);
                else
                        update_x (vx*2+1, mx+1, rx, x, y, new_val);
        }
        update_y (vx, lx, rx, 1, 0, m-1, x, y, new_val);
}
```

Compression of two-dimensional pieces of wood

Suppose that the problem is as follows: there are $n$points in the plane defined by its coordinates (x_i, y_i), and get requests like "count the number of points lying in a box $((x_1, y_1), (x_2, y_2))$. " It is clear that in the case of such a task becomes unnecessarily wasteful to build a two-dimensional segment tree with $O(n^2)$elements. Much of this memory will be wasted, because every single point can be reached only in the $O(\log n)$segments of wood pieces in the first coordinate and, therefore, the total "useful" size of all segments of the trees on the second coordinate is the value $O(n \log n)$.

Then proceed as follows: at each node of the tree segments in the first coordinate will be stored segment tree built only on the second coordinates, which are found in the current segment of the first frame. In other words, the construction of the tree segments within some vertex with the number $vx$and boundaries tlx, trx, we consider only those points that fall in this segment $x \in [tlx; trx]$, and build a segment tree just above them.

In this way we will achieve that every tree segments the second coordinate will occupy as much memory as it should. As a result, the total **amount of memory** decreases to $O(n \log n)$. **Responding to a request** , we will continue for $O(\log^2 n)$just now in the call

request from the wood pieces on the second coordinate, we'll have to do a binary search on the second coordinate, but it will not worsen the asymptotic behavior.

But the payback will be impossible to make an arbitrary **modification request** : in fact, if a new point, then it will lead to the fact that we will have in any tree segments the second coordinate add a new element into the middle, impossible to do that effectively.

In conclusion, we note that a concise manner described two-dimensional segment tree is practically **equivalent to** the above-described modification of the one-dimensional tree segments (see. "Saving the entire subarray in each node of the tree segments"). In particular, it turns out that the herein described two-dimensional segment tree - it's just a special case of the subarray conservation at each vertex of the tree where he subarray is stored in a tree segments. It follows that if you have to abandon the two-dimensional pieces of wood because of impossibility of performance of a query, it makes sense to try to replace the embedded tree segments in any more powerful data structure, for example, the Cartesian tree .

Segment tree while preserving the history of its values (better to persistent-data structure)

Persistent-data structure called such a data structure that stores every modification of its previous state. This allows to apply to any version that we are interested in the data structure and execute its request.

Segment tree is one of the data structures that can be converted into a persistent-data structure (of course, we consider the persistent-efficient structure, and not one that copies all himself entirely before each update).

In fact, any change in the query tree segments leads to a change in data $O(\log n)$peaks, moreover along a path starting from the root. So, if we keep the tree segments in the index (ie pointers to the left and right sons do pointers stored at the top), you can view the update we should just change instead of having a vertex to create new vertices, links of which are directed to the old top. Thus, when requesting an update will create $O(\log n)$new peaks, including will create a new root of the tree segments, and all prev version tree, suspended for the old root, will remain unchanged.

Here is an example implementation of the simplest pieces of wood when there is only a request for calculation of the amount of subsegments and modification request singular.

```
struct vertex {
        vertex * l, * r;
        int sum;

        vertex (int val)
                : l(NULL), r(NULL), sum(val)
        { }

        vertex (vertex * l, vertex * r)
                : l(l), r(r), sum(0)
        {
                if (l)  sum += l->sum;
                if (r)  sum += r->sum;
        }
};

vertex * build (int a[], int tl, int tr) {
        if (tl == tr)
                return new vertex (a[tl]);
        int tm = (tl + tr) / 2;
        return new vertex (
                build (a, tl, tm),
                build (a, tm+1, tr)
        );
}

int get_sum (vertex * t, int tl, int tr, int l, int r) {
        if (l > r)
                return 0;
        if (l == tl && tr == r)
                return t->sum;
        int tm = (tl + tr) / 2;
        return get_sum (t->l, tl, tm, l, min(r,tm))
                + get_sum (t->r, tm+1, tr, max(l,tm+1), r);
}

vertex * update (vertex * t, int tl, int tr, int pos, int new_val) {
        if (tl == tr)
                return new vertex (new_val);
        int tm = (tl + tr) / 2;
        if (pos <= tm)
                return new vertex (
                                update (t->l, tl, tm, pos, new_val),
                                t->r
                        );
        else
                return new vertex (
                                t->l,
                                update (t->r, tm+1, tr, pos, new_val)
                        );
```

```
}
```

With this approach can be turned into persistent-data structure virtually any segment tree.

• Ответить • Поделиться ›

dimir • 3 месяца назад

А как реализовать дерево с операцией прибавления числа к отрезку и нахождения суммы на отрезке ?
В статье есть дерево с обновлением одного элемента и нахождением суммы и с обновлением отрезка и нахождения значения одного элемента.

• Ответить • Поделиться ›

dimir ↗ dimir • 3 месяца назад

вопрос снят.

• Ответить • Поделиться ›

w0w • 3 месяца назад

Подскажите пожалуйста, как можно решить вот эту задачу http://codeforces.ru/contest/4...

• Ответить • Поделиться ›

NBoy • 4 месяца назад

Можете дать код
Прибавление на отрезке, запрос максимума?
а то не понял.Заранее спасибо!

• Ответить • Поделиться ›

w0w • 4 месяца назад

Подскажите пожалуйста/ как можно реализовать функцию прибавления к отрезку основного массива [l..r] отрезок второго маасива [l..r]
Заранее спасибо!

• Ответить • Поделиться ›

e_maxx Модератор ↗ w0w • 4 месяца назад

Задача не до конца сформулирована (какие запросы требуется уметь делать?), но из того, что я понял - разве нельзя завести дерево отрезков, в котором будет храниться, сколько раз соответствующие элементы второго массива должны быть прибавлены к первому? Т.е. изначально дерево заполнено нулями, а прибавление на отрезке [l..r] соответствует инкременту в дереве на отрезке [l..r].

• Ответить • Поделиться ›

Artem • 5 месяцев назад

Что такое pos

• Ответить • Поделиться ›

e_maxx Модератор ↗ Artem • 4 месяца назад

Позиция, т.е. индекс изменяемого/запрашиваемого элемента.

• Ответить • Поделиться ›

Sparik • год назад

"стандартному дереву отрезков требуется порядка элементов памяти для работы над массивом размера ."

достаточно 2n

• Ответить • Поделиться ›

Raid ↗ Sparik • 8 месяцев назад

Это верно только для n=2^k. Например, для n=127 не хватит 254 единиц памяти.

1 • Ответить • Поделиться ›

Vlad • 2 года назад

Помогите, пожалуйста. Вот я допустим на отрезке присвоил или прибавил процедурой update. Как мне теперь подсчитать новые значения в дереве? Т.е после выполнения update я сразу буду готов отвечать на запрос суммы/минимума/чего-то еще, или нужно как-то вызвать get? Потому что я так и не понял, что делает get и как ее вызвать и что она сделает. Помогите, пожалуйтса. Заранее спасибо!

• Ответить • Поделиться ›

e_maxx Модератор ↗ Vlad • 4 месяца назад

Дополнительно ничего вызывать не надо, все попутные манипуляции должны уже делаться при обработке следующих запросов.

Функция get - отвечает на требуемый в задаче запрос, например, возвращает значение в заданной позиции, или что-либо ещё. Получается, когда она спускается по дереву, она должна попутно учитывать те присвоенные/прибавленные значения, которые были выставлены на предыдущих запросах.

Т.е. главный принцип - явно не пересчитывать все значения при каждой модификации, а вместо этого лишь оставить компактные "пометки" для будущих запросов.

• Ответить • Поделиться ›

Ivan • 2 года назад

A зачем в "присвоение на отрезке" в функции "int get(int v, int tl, int tr, int v)" в функции "push(v)" t[v] = -1;

• Ответить • Поделиться ›

e_maxx Модератор ➚ Ivan • 2 года назад

Чтобы снять отметку о том, что всё поддерево с корнем в вершине v надо перекрашивать: эту отметку мы уже "отдали" детям: v*2 и v*2+1. В противном случае, если не сделать t[v]=-1, то в дальнейшем, к примеру, если вершина v*2 целиком перекрасится в какой-то другой цвет, вершина v всё равно перезатрёт эти изменения своим цветом.

• Ответить • Поделиться ›

Steve_jobs • 2 года назад

можно вычислять количество разных элементов в интервале с помощью дерево отрезков? Заранее спасибо.

• Ответить • Поделиться ›

e_maxx Модератор ➚ Steve_jobs • 2 года назад

Давайте начнём с оффлайнового решения, когда все запросы известны заранее. Т.е. мы начинаем с пустого массива, дальше на каждом шаге мы добавляем в рассмотрение очередной элемент массива, и отвечаем на все запросы, правая граница которых оканчивается на текущем элементе. Как это делать: давайте для каждого числа X будем поддерживать позицию LAST[X] его последнего встреченного вхождения (изначально все они равны минус бесконечности). Тогда ответ на запрос "количество различных чисел на отрезке, начинающемся в L и заканчивающемся в текущей позиции" будет равен количеству чисел X, для которых LAST[X] >= L. Таким образом, построив и поддерживая дерево отрезков над этим массивом LAST[], мы можем отвечать на запрос количества различных чисел за O(log N) в оффлайне.

Чтобы перейти к онлайновому решению, (кажется), достаточно перейти к персистентному дереву отрезков: т.е. заранее промоделировать и сохранить деревья отрезков после добавления каждого элемента массива, а потом отвечать на запрос, обращаясь к дереву нужной версии.

• Ответить • Поделиться ›

Steve_jobs ➚ e_maxx • 2 года назад

Спасибо большое!!!

• Ответить • Поделиться ›

quest • 2 года назад

а есть ли какая-нибудь структура данных, которая может поддерживать операции: прибавить на отрезке число x, найти нод на отрезке? и есть ли какие-нибудь задачи на эту штуку?

• Ответить • Поделиться ›

e_maxx Модератор ➚ quest • 2 года назад

Почти наверняка нет, поскольку после применения прибавления - факторизация чисел и, следовательно, НОД, могли сильно и труднопредсказуемо измениться (например, к массиву [2,7] с НОД=1 прибавили 3 - получился [5,10] с НОД=5).

• Ответить • Поделиться ›

guest ➚ e_maxx • год назад

Это можно сделать.

Заметим следующий факт: gcd(a + x, b + x) = gcd(b - a, b + x) = gcd(a - b, a + x) = gcd(b - a, a + x) = gcd(a - b, b + x).

Теперь, допустим, у нас есть числа a, b, c, d, к которым надо прибавить x.

gcd{a + x, b + x, c + x, d + x} = gcd{gcd(a + x, b + x), gcd(b + x, c + x), gcd(c + x, d + x)}.
Исходя из факта выше: gcd{a + x, b + x, c + x, d + x} = gcd{gcd(b - a, b + x), gcd(c - b, c + x), gcd(d - c, d + x)} =

gcd{gcd{b - a, c - b, d - c}, gcd{b + x, c + x, d + x}} = ... = gcd{b - a, c - b, d - c, d + x}.

Это значит, что чтобы прибавить к отрезку X и посчитать gcd на отрезке надо взять

gcd(gcd{разностей соседних элементов}, самый правый элемент на отрезке + x).
Т.к. при прибавлении на отрезке разности соседних элементов не меняются, то gcd{разности соседних элементов} можно поддерживать в

дереве отрезков.
Итак, что будем хранить в вершинах дерева:

показать больше

3 • Ответить • Поделиться ›

e_maxx Модератор ➚ guest • 4 месяца назад
Очень красиво, спасибо!
• Ответить • Поделиться ›

Miras Mirzakerey • 3 года назад
Можно ли с помощью дерева отрезков перевернуть какой-то отрезок от l до r, сохраняя при этом дерево (не перестраивать) и с сравнительно небольшой асимптотикой?
• Ответить • Поделиться ›

e_maxx Модератор ➚ Miras Mirzakerey • 3 года назад
На всякий случай напишу, что известная структура данных, позволяющая делать перевороты на отрезке, - это декартово дерево (http://e-maxx.ru/algo/treap).
1 • Ответить • Поделиться ›

e_maxx Модератор ➚ Miras Mirzakerey • 3 года назад
Лично мне способа переворачивать с помощью "обычного" дерева отрезков (такого рода, как описаны в статье) неизвестно.
1 • Ответить • Поделиться ›

Nurzhan Dyussenaliyev ➚ e_maxx • 3 года назад
Можно же также как и в декартовом дереве. Номер левого потомка уже не будет v * 2, а будет L[v], и аналогично для правого. Это если нет запросов вставлений и удалений элементов из массива.
2 • Ответить • Поделиться ›

e_maxx Модератор ➚ Nurzhan Dyussenaliyev • 2 года назад
Что-то непонятно, как так можно. Допустим, длина массива равна 4 (т.е. индексы от 0 до 3), и нас попросили перевернуть с 1 по 3. Получается, одна половинка переворачиваемого (с 1 по 1) лежит в левом поддереве, а другая (с 2 по 3) - в правом поддереве, и как их обменять местами (особенно учитывая, что они имеют разную длину)?
1 • Ответить • Поделиться ›