Efficient and easy segment trees

By **Al.Cash**, 2 years ago,

This is my first attempt at writing something useful, so your suggestions are welcome.

Most participants of programming contests are familiar with segment trees to some degree, especially having read this articles http://codeforces.com/blog/entry/15890, http://e-maxx.ru/algo/segment_tree (Russian only). If you're not — don't go there yet. I advise to read them after this article for the sake of examples, and to compare implementations and choose the one you like more (will be kinda obvious).

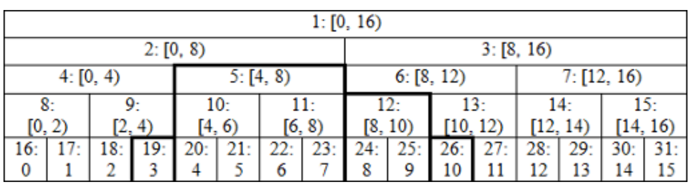
Segment tree with single element modifications

Let's start with a brief explanation of segment trees. They are used when we have an array, perform some changes and queries on continuous segments. In the first example we'll consider 2 operations:

1. modify one element in the array;
2. find the sum of elements on some segment. .

Perfect binary tree

I like to visualize a segment tree in the following way: image link



Notation is *node_index: corresponding segment* (left border included, right excluded). At the bottom row we have our array (0-indexed), the leaves of the tree. For now suppose it's length is a power of 2 (16 in the example), so we get perfect binary tree. When going up the tree we take pairs of nodes with indices $(2*i, 2*i+1)$ and combine their values in their parent with index i. This way when we're asked to find a sum on interval $[3, 11)$, we need to sum up only values in the nodes 19, 5, 12 and 26 (marked with bold), not all 8 values inside the interval. Let's jump directly to implementation (in C++) to see how it works:

```
const int N = 1e5;  // limit for array size
int n;  // array size
int t[2 * N];

void build() {  // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void modify(int p, int value) {  // set value at position p
  for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];
}

int query(int l, int r) {  // sum on interval [l, r)
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
```

```
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%d", t + n + i);
  build();
  modify(0, 1);
  printf("%d\n", query(3, 11));
  return 0;
}
```

That's it! Fully operational example. Forget about those cumbersome recursive functions with 5 arguments!

Now let's see why this works, and works very efficient.

1. As you could notice from the picture, leaves are stored in continuous nodes with indices starting with n, element with index i corresponds to a node with index $i+n$. So we can read initial values directly into the tree where they belong.

2. Before doing any queries we need to build the tree, which is quite straightforward and takes $O(n)$ time. Since parent always has index less than its children, we just process all the internal nodes in decreasing order. In case you're confused by bit operations, the code in *build()* is equivalent to `t[i] = t[2*i] + t[2*i+1]`.

3. Modifying an element is also quite straightforward and takes time proportional to the height of the tree, which is $O(log(n))$. We only need to update values in the parents of given node. So we just go up the tree knowing that parent of node p is $p/2$ or `p>>1`, which means the same. `p^1` turns $2*i$ into $2*i+1$ and vice versa, so it represents the second child of p's parent.

4. Finding the sum also works in $O(log(n))$ time. To better understand it's logic you can go through example with interval $[3, 11)$ and verify that result is composed exactly of values in nodes 19, 26, 12 and 5 (in that order).

General idea is the following. If l, the left interval border, is odd (which is equivalent to `l&1`) then l is the right child of its parent. Then our interval includes node l but doesn't include it's parent. So we add `t[l]` and move to the right of l's parent by setting $l=(l+1)/2$. If l is even, it is the left child, and the interval includes its parent as well (unless the right border interferes), so we just move to it by setting $l=l/2$. Similar argumentation is applied to the right border. We stop once borders meet.

No recursion and no additional computations like finding the middle of the interval are involved, we just go through all the nodes we need, so this is very efficient.

Arbitrary sized array

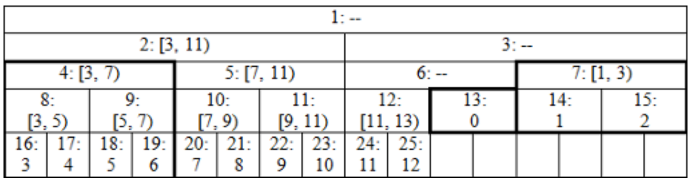
For now we talked only about an array with size equal to some power of 2, so the binary tree was perfect. The next fact may be stunning, so prepare yourself.

The code above works for any size n.

Explanation is much more complex than before, so let's focus first on the advantages it gives us.

1. Segment tree uses exactly $2 * n$ memory, not $4 * n$ like some other implementations offer.
2. Array elements are stored in continuous manner starting with index n.
3. All operations are very efficient and easy to write.

You can skip the next section and just test the code to check that it's correct. But for those interested in some kind of explanation, here's how the tree for $n = 13$ looks like: image link



It's not actually a single tree any more, but a set of perfect binary trees: with root 2 and height 4, root 7 and height 2, root 12 and height 2, root 13 and height 1. Nodes denoted by dashes aren't ever used in *query* operations, so it doesn't matter what's stored there. Leaves seem to appear on different heights, but that can be fixed by cutting the tree before the node 13 and moving its right part to the left. I believe the resulting structure can be shown to be isomorphic to a part of larger perfect binary tree with respect to operations we perform, and this is why we get correct results.

I won't bother with formal proof here, let's just go through the example with interval [0, 7). We have $l = 13, r = 20$, `l&1 => add t[13]` and borders change to $l = 7, r = 10$. Again `l&1 => add t[7]`, borders change to $l = 4, r = 5$, and suddenly nodes are at the same height. Now we have `r&1 => add t[4 = --r]`, borders change to $l = 2, r = 2$, so we're finished.

Modification on interval, single element access

Some people begin to struggle and invent something too complex when the operations are inverted, for example:

1. add a value to all elements in some interval;
2. compute an element at some position.

But all we need to do in this case is to switch the code in methods *modify* and *query* as follows:

```
void modify(int l, int r, int value) {
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) t[l++] += value;
    if (r&1) t[--r] += value;
  }
}

int query(int p) {
  int res = 0;
  for (p += n; p > 0; p >>= 1) res += t[p];
  return res;
}
```

If at some point after modifications we need to inspect all the elements in the array, we can push all the modifications to the leaves using the following code. After that we can just traverse elements starting with index n. This way we reduce the complexity from $O(nlog(n))$to $O(n)$ similarly to using *build* instead of n modifications.

```
void push() {
  for (int i = 1; i < n; ++i) {
    t[i<<1] += t[i];
    t[i<<1|1] += t[i];
    t[i] = 0;
  }
}
```

Note, however, that code above works only in case the order of modifications on a single element doesn't affect the result. Assignment, for example, doesn't satisfy this condition. Refer to section about lazy propagation for more information.

Non-commutative combiner functions

For now we considered only the simplest combiner function — addition. It is commutative, which means the order of operands doesn't matter, we have $a+b=b+a$. The same applies to min and max, so we can just change all occurrences of `+` to one of those functions and be fine. But don't forget to initialize query result to infinity instead of 0.

However, there are cases when the combiner isn't commutative, for example, in the problem 380C - Sereja and Brackets, tutorial available here http://codeforces.com/blog/entry/10363. Fortunately, our implementation can easily support that. We define structure `S` and *combine* function for it. In method *build* we just change `+` to this function. In *modify* we need to ensure the correct ordering of children, knowing that left child has even index. When answering the query, we note that nodes corresponding to the left border are processed from left to right, while the right border moves from right to left. We can express it in the code in the following way:

```
void modify(int p, const S& value) {
  for (t[p += n] = value; p >>= 1; ) t[p] = combine(t[p<<1], t[p<<1|1]);
}

S query(int l, int r) {
  S resl, resr;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) resl = combine(resl, t[l++]);
    if (r&1) resr = combine(t[--r], resr);
  }
  return combine(resl, resr);
}
```

Lazy propagation

Next we'll describe a technique to perform both range queries and range modifications, which is called lazy propagation. First, we need more variables:

```
int h = sizeof(int) * 8 - __builtin_clz(n);
int d[N];
```

h is a height of the tree, the highest significant bit in n. `d[i]` is a delayed operation to be propagated to the children of node i when necessary (this should become clearer from the examples). Array size if only `N` because we don't have to store this information for leaves — they don't have any children. This leads us to a total of $3 * n$ memory use.

Previously we could say that `t[i]` is a value corresponding to it's segment. Now it's not entirely true — first we need to apply all the delayed operations on the route from node i to the root of the tree (parents of node i). We assume that `t[i]` already includes `d[i]`, so that route starts not with i but with its direct parent.

Let's get back to our first example with interval $[3, 11)$, but now we want to modify all the elements inside this interval. In order to do that we modify `t[i]` and `d[i]` at the nodes 19, 5, 12 and 26. Later if we're asked for a value for example in node 22, we need to propagate modification from node 5 down the tree. Note that our modifications could affect `t[i]` values up the tree as well: node 19 affects nodes 9, 4, 2 and 1, node 5 affects 2 and 1. Next fact is critical for the complexity of our operations:

Modification on interval $[l, r)$ affects `t[i]` values only in the parents of border leaves: `l+n` and `r+n-1` (except the values that compose the interval itself — the ones accessed in *for* loop).

The proof is simple. When processing the left border, the node we modify in our loop is always the right child of its parent. Then all the previous modifications were made in the subtree of the left child of the same parent. Otherwise we would process the parent instead of both its children. This means current direct parent is also a parent of leaf `l+n`. Similar arguments apply to the right border.

OK, enough words for now, I think it's time to look at concrete examples.

Increment modifications, queries for maximum

This is probably the simplest case. The code below is far from universal and not the most efficient, but it's a good place to start.

```
void apply(int p, int value) {
  t[p] += value;
  if (p < n) d[p] += value;
}

void build(int p) {
  while (p > 1) p >>= 1, t[p] = max(t[p<<1], t[p<<1|1]) + d[p];
}

void push(int p) {
  for (int s = h; s > 0; --s) {
    int i = p >> s;
    if (d[i] != 0) {
      apply(i<<1, d[i]);
      apply(i<<1|1, d[i]);
      d[i] = 0;
    }
  }
}

void inc(int l, int r, int value) {
  l += n, r += n;
  int l0 = l, r0 = r;
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1) apply(l++, value);
    if (r&1) apply(--r, value);
```

```
  }
  build(l0);
  build(r0 - 1);
}

int query(int l, int r) {
  l += n, r += n;
  push(l);
  push(r - 1);
  int res = -2e9;
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1) res = max(res, t[l++]);
    if (r&1) res = max(t[--r], res);
  }
  return res;
}
```

Let's analyze it one method at a time. The first three are just helper methods user doesn't really need to know about.

1. Now that we have 2 variables for every internal node, it's useful to write a method to *apply* changes to both of them. $p < n$ checks if p is not a leaf. Important property of our operations is that if we increase all the elements in some interval by one value, maximum will increase by the same value.

2. *build* is designed to update all the parents of a given node.

3. *push* propagates changes from all the parents of a given node down the tree starting from the root. This parents are exactly the prefixes of p in binary notation, that's why we use binary shifts to calculate them.

Now we're ready to look at main methods.

1. As explained above, we process increment request using our familiar loop and then updating everything else we need by calling *build*.

2. To answer the query, we use the same loop as earlier, but before that we need to push all the changes to the nodes we'll be using. Similarly to *build*, it's enough to push changes from the parents of border leaves.

It's easy to see that all operations above take $O(log(n))$ time.

Again, this is the simplest case because of two reasons:

1. order of modifications doesn't affect the result;
2. when updating a node, we don't need to know the length of interval it represents.

We'll show how to take that into account in the next example.

Assignment modifications, sum queries

This example is inspired by problem Timus 2042

Again, we'll start with helper functions. Now we have more of them:

```
void calc(int p, int k) {
  if (d[p] == 0) t[p] = t[p<<1] + t[p<<1|1];
  else t[p] = d[p] * k;
}

void apply(int p, int value, int k) {
  t[p] = value * k;
  if (p < n) d[p] = value;
}
```

These are just simple $O(1)$ functions to calculate value at node p and to apply a change to the node. But there are two thing to explain:

1. We suppose there's a value we never use for modification, in our case it's 0. In case there's no such value — we would create additional boolean array and refer to it instead of checking `d[p] == 0`.
2. Now we have additional parameter k, which stands for the lenght of the interval corresponding to node p. We will use this name consistently in the code to preserve this meaning. Obviously, it's impossible to calculate the sum without this parameter. We can avoid passing this parameter if we precalculate this value for every node in a separate array or calculate it from the node index on the fly, but I'll show you a way to avoid using extra memory or calculations.

Next we need to update *build* and *push* methods. Note that we have two versions of them: one we introduces earlier that processes the whole tree in $O(n)$, and one from the last example that processes just the parents of one leaf in $O(log(n))$. We can easily combine that functionality into one method and get even more.

```
void build(int l, int r) {
  int k = 2;
  for (l += n, r += n-1; l > 1; k <<= 1) {
    l >>= 1, r >>= 1;
    for (int i = r; i >= l; --i) calc(i, k);
  }
}

void push(int l, int r) {
  int s = h, k = 1 << (h-1);
  for (l += n, r += n-1; s > 0; --s, k >>= 1)
    for (int i = l >> s; i <= r >> s; ++i) if (d[i] != 0) {
      apply(i<<1, d[i], k);
      apply(i<<1|1, d[i], k);
      d[i] = 0;
    }
}
```

Both this methods work on any interval in $O(log(n) + |r - l|)$ time. If we want to transform some interval in the tree, we can write code like this:

```
push(l, r);
...  // do anything we want with elements in interval [l, r)
build(l, r);
```

Let's explain how they work. First, note that we change our interval to closed by doing `r += n-1` in order to calculate parents properly. Since we process our tree level by level, is't easy to maintain current interval level, which is always a power of 2. *build* goes bottom to top, so we initialize k to 2 (not to 1, because we don't calculate anything for the leaves but start with their direct parents) and double it on each level. *push* goes top to bottom, so k's initial value depends here on the height of the tree and is divided by 2 on each level.

Main methods don't change much from the last example, but *modify* has 2 things to notice:

1. Because the order of modifications is important, we need to make sure there are no old changes on the paths from the root to all the nodes we're going to update. This is done by calling *push* first as we did in *query*.
2. We need to maintain the value of k.

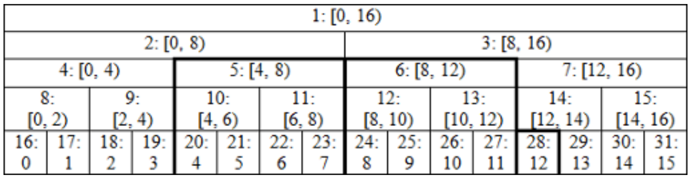
```
void modify(int l, int r, int value) {
  if (value == 0) return;
  push(l, l + 1);
  push(r - 1, r);
  int l0 = l, r0 = r, k = 1;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1, k <<= 1) {
    if (l&1) apply(l++, value, k);
    if (r&1) apply(--r, value, k);
  }
  build(l0, l0 + 1);
  build(r0 - 1, r0);
}

int query(int l, int r) {
  push(l, l + 1);
  push(r - 1, r);
  int res = 0;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (r&1) res += t[--r];
  }
  return res;
}
```

One could notice that we do 3 passed in *modify* over almost the same nodes: 1 down the tree in *push*, then 2 up the tree. We can eliminate the last pass and calculate new values only where it's necessary, but the code gets more complicated:

```
void modify(int l, int r, int value) {
  if (value == 0) return;
  push(l, l + 1);
  push(r - 1, r);
  bool cl = false, cr = false;
  int k = 1;
  for (l += n, r += n; l < r; l >>= 1, r >>= 1, k <<= 1) {
    if (cl) calc(l - 1, k);
    if (cr) calc(r, k);
    if (l&1) apply(l++, value, k), cl = true;
    if (r&1) apply(--r, value, k), cr = true;
  }
  for (--l; r > 0; l >>= 1, r >>= 1, k <<= 1) {
    if (cl) calc(l, k);
    if (cr && (!cl || l != r)) calc(r, k);
  }
}
```

Boolean flags denote if we already performed any changes to the left and to the right. Let's look at an example: image link



We call *modify* on interval $[4, 13)$:

1. $l = 20, r = 29$, we call *apply(28)*;
2. $l = 10, r = 14$, we call *calc(14)* — first node to the right of current interval is exactly the parent of last modified node;
3. $l = 5, r = 7$, we call *calc(7)* and then *apply(5)* and *apply(6)*;
4. $l = 3, r = 3$, so the first loop finishes.

Now you should see the point of doing `--l`, because we still need to calculate new values in nodes 2, 3 and then 1. End condition is `r > 0` because it's possible to get $l = 1, r = 1$after the first loop, so we need to update the root, but `--l` results in $l = 0$.

Compared to previous implementation, we avoid unnecessary calls *calc(10)*, *calc(5)* and duplicate call to *calc(1)*.

efficiency, segment trees

+354 Al.Cash 2 years ago 156

Comments (156)

Write comment?


Alex7

2 years ago, # | +4

Is lazy propagation possible this way?

→ Reply


Al.Cash

2 years ago, # ^ | +8

Of course. Modification is more complex of course, but almost all principles are already described.

I just want to polish the implementation before posting.

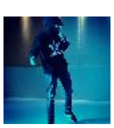
→ Reply


raihatneloy

2 years ago, # ^ | 0

i most of the time use this method of segment tree & i also coded lazy propagation! :)

→ Reply


punetharomil

18 months ago, # ^ | 0

I tried lazy propagation for long but couldn't code the iterative version of

it. Can you share your code for

it. Can you share your code for iterative lazy propagation?

→ Reply


cmd

2 years ago, # | +23

Also for reference: **Urbanowicz**'s post about non-recursive segment tree implementation: https://codeforces.com/blog/entry/1256

→ Reply


rajon_aust

2 years ago, # ^ | 0

Thanks ! First time I know something new that segment tree with no-recursive .

→ Reply


Al.Cash

2 years ago, # ^ | +12

True, my implementation is basically a refinement of the one in that post and in **Alias**'s comment to that post. But for some reason it's still not well known and not really searchable, and I wasn't aware of that post. Also I hope to provide more knowledge, especially after I add the section about lazy propagation.

→ Reply


rng_58

2 years ago, # | +41

Just wondering, how did you write codes like this?

for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1];

Did you write it in the first attempt or you wrote something else and compressed that?

→ Reply


Al.Cash

2 years ago, # ^ | +8

I don't remember, but of course it wasn't the first version, the code was revisited several times.

I think at first it looked like the code as I posted for non-commutative combiners, which is also used in *build* method.

^1 trick is used in max flow implementation where edge is stored adjacent to it's reverse, so we can use ^1 to get from one to another. I'm not sure, but probably it came from there.

→ Reply

ch_code
2 months ago, # ^ | 0

What happens when the code (first snippet on the article) runs forever?

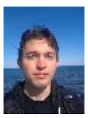
→ Reply


KADR
2 years ago, # ^ | +19

I believe smth like this is more understandable and is equivalently short:

```
for (t[p += n] = value; p /= 2; ) t[p] = t[p * 2] +
t[p * 2 + 1];
```

→ Reply


Al.Cash
2 years ago, # ^ | +10

Well, it's a matter of style. I prefer not to write modifications inside a loop condition.

→ Reply


KADR
2 years ago, # ^ | 0

I'm just saying that this way it's more similar to traditional segment tree implementation and thus easier to follow the logic. But you are right, it's just a matter of style :)

→ Reply


innok96
2 years ago, # | -16

it so useful for non-red participants

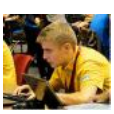
→ Reply

xrisk
2 years ago, # | ← Rev. 2 0

Hello, can you explain this line ? ~~~~~ t[p>>1] = t[p] + t[p^1] ~~~~~

Specifically, why are you xor-ing it?

→ Reply


Rubanenko
2 years ago, # ^ | +6

p>>1 is a parent of p in the tree. Another son of p>>1 is p^1, since x^1 gives you x with reversed parity(last bit).

→ Reply

xrisk
2 years ago, # ^ | 0

So basically, p^1 equals p/2 + 1 ?

→ Reply

2 years ago, # ^ | +18

No, p^1 equals to p+1 if p is

No, p^1 equals to p+1 if p is even, and p-1 if p is odd.

Rubanenko

→ Reply

2 years ago, # ^ | +1

mketa

xor will give the other child of the parent. For example if p is left child, p^1 will give the right one. Quite an interesting way of doing this in fact.

→ Reply

2 years ago, # ^ | 0

xrisk

Thank you!

→ Reply

12 months ago, # ^ | 0

nurlansofiyev

p>>1 equal to p/2, and this give us parent of node. and and p^1 is binary xor operator from discrete math. ^ operato copies the bit if it is set in one operand but not both. and if p is odd p^1 will be even and if p is even p^1 will be odd. this come from binary representation of p. example if 6^1 -> 110 ^ 001 -> 111 -> 7. other expamle : 8^1 -> 1000 ^ 0001 -> 9

→ Reply

2 years ago, # | 0

xrisk

Hello, sorry to ask another silly question. Al.Cash says :

"l, the left interval border, is odd (which is equivalent to l&1)"

I am not familiar with bit operations too much, and I am having difficulty how `x AND 1` gives odd/even value. Thanks in advance :)

→ Reply

2 years ago, # ^ | +3

edgarciarod

Every odd value ends with 1(3 = 11, 5 = 101) and every even value ends with 0(2 = 10, 4 = 100).

```
(xxx1 and 0001) --> 1 // odd
(xxx0 and 0001) --> 0 // even
```

→ Reply

2 years ago, # | -13

alanor1

Hi, I am new in competitive programming, I was reading this article and trying to use it , but i dont understand how the function query works, for example if I have these numbers 1 47 45 23 348 example if I have these numbers 1 47 45 23 348 and I would like the sum from 47 to 23, whats numbers should I put in the arguments???? please help, thanks

→ Reply

google

22 months ago, # ^ | 0

well your array starts from position 0. Position of 47 is 1 and of 23 is 3. The Query will give you an answer for [l, r) so you should pass as arguments (1, 4). Hope it helps!

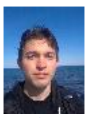
→ Reply

alanor1

22 months ago, # ^ | 0

thanks so much

→ Reply



Al.Cash

2 years ago, # | ← Rev. 2 +25

Section about lazy propagation has been added.

→ Reply

zscefn

2 years ago, # | +11

I did some speed comparison between recursive and non-recursive lazy segment trees. With array size of 1<<18 and 1e7 randomly chosen operations between modifications and queries. Array size of 1<<18 is of course easier for my recursive code which uses arrays of the size 2^n but on practise it doesn't affect much to the speed of the code.

My recursive (http://paste.dy.fi/BUZ): 6 s

Non-recursive from the blog: (http://paste.dy.fi/kBY): 3 s

That's quite big difference in my opinion. Unfortunately the non-recursive one seems to be a bit more painful to code but maybe it's just that I'm used to code it recursively.

→ Reply

zeura

2 years ago, # | 0

I think your modification on all element in an interval is not incrementing every element in an interval.You should check it.
Ex: n=8 0 1 2 3 4 5 6 7 modif(0,8,5); It should increment all places by 5.According to your algorithm array t is- 33 6 22 1 5 9 13 0 1 2 3 4 5 6

7

,
but clearly all element isn't incremented by 5.

→ Reply


Al.Cash

2 years ago, # ^ | 0

Probably you're talking about the last example, but the operation there is assignment, not increment. And array looks absolutely correct except it should start with 40 not 33 (I believe 33 is a typo — let me know if it's not the case).

That's the point of 'lazy propagation' — not to modify anything until we really need it. You shouldn't access tree elements directly unless you called *push(0, 8)*.

→ Reply


zeura

2 years ago, # ^ | 0

I was talking about this modify function: void modify(int l, int r, int value) { for (l += n, r += n; l < r; l >>= 1, r >>= 1) { if (l&1) t[l++] += value; if (r&1) t[--r] += value; } }

→ Reply


Al.Cash

2 years ago, # ^ | 0

I see. Again, if you want to get an element — call *query*, don't access the tree directly. And you don't need to call *build* in this example.

→ Reply


zeura

2 years ago, # ^ | 0

Thanks, I think I got it. :D

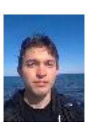
→ Reply


siddharths067

2 years ago, # | 0

Is it possible to extend this form to Multiple dimensions !

→ Reply


Al.Cash

2 years ago, # ^ | 0

Sure, just change one loop everywhere into two nested loops for different coordinates.

→ Reply


wodesuck

2 years ago, # | 0

How to perform find-kth query? For n that is not the power of 2.

→ Reply

satirmo

2 years ago, # | 0

Hi. I have a question. When should it be considered necessary to use lazy propagation?

→ Reply

Al.Cash

2 years ago, # ^ | 0

When you have both range modifications and range queries, or just range modifications for which order is important (for example, assignment).

→ Reply

siddharths067

2 years ago, # | ← Rev. 2 0

Is it possible to generalize the structure for the Lazy propagation Loop , as in both the cases the propagation format of the loop changes. In recursion , no matter what the context the function has the same structure for lazy propagation namely , checking the Boolean flag and changing the value of the child nodes..

It would be better if you generalize it ....

For Example in Function Build :

```
for (l += n, r += n-1; s > 0; --s, k >>= 1)
for (int i = l >> s; i <= r >> s; ++i) if (d[i] != 0)
```

From Sum Queries

```
for (; l < r; l >>= 1, r >>= 1)
```

To RMQ

There is a difference in initial values of l and r in each loop.

Note: My Concern here is generalizing a loop for propagating the Segment Tree Lazily , I don't Care about the additional factor K for Sum queries or any additional parameter for a particular purpose.

→ Reply

IAmNomad

2 years ago, # | ← Rev. 2 0

Understood. + for you!

→ Reply

2 years ago, # | 0

Can anyone please explain how the code in praveen14078

Can anyone please explain how the code in section "Modification on interval, single element access" actually work with a small example if possiple explaining how to we get sum over a range after the update(which also I am not able to understand!) ?

→ Reply

2 years ago, # ^ | 0

Al.Cash

This example doesn't support getting sum over a range, only single element access (I was hoping it's clear from the heading). It only shows that sometimes we can invert basic operations, but for more complex operations you need lazy propagation.

→ Reply

2 years ago, # ^ | 0

praveen14078

can you give an example of single element access after some update , i am not able to grasp how query() would return a[p] (as u mentioned in some earlier comment to call query to access an element or simply what does the query() do ?

→ Reply

2 years ago, # ^ | 0

Al.Cash

query() doesn't simply access the element — it calculates it as a sum of all the updates applied to that element.

→ Reply

2 years ago, # | ← Rev. 2 0

nishanthvydana

Can someone explain logic(bit operations) here (t[p += n] = value; p > 1; p >>= 1)

The above expression is from the below function void modify(int p, int value) { // set value at position p for (t[p += n] = value; p > 1; p >>= 1) t[p>>1] = t[p] + t[p^1]; }

→ Reply

2 years ago, # ^ | 0

xrisk

It is equivalent to this simplified code:

```
void modify(int p, long new_val)
        {
                p += n;
                tree[p] = new_val;
                for (; p > 1; p/= 2)
                {
                        tree[p/2] = tree[p] +
tree[p^1];
```

```
        }
    }
```

For the explanation of using xor(^) you can check the previous comment that I made.

→ Reply


nishanthvydana

2 years ago, # ^ | 0

Can u explain query function a bit more clearly Thanks for ur previous answer

→ Reply


xrisk

2 years ago, # | 0

Thanks for this post, it is very useful for beginners like me. But there is a small problem; the picture of the segment tree at the very beginning of the post is not showing. It would be nice if you fixed that :)

→ Reply

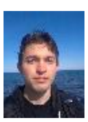

wadhwasahil

2 years ago, # | ← Rev. 2 0

Please can somebody explain how to change all the elements of an array in an interval [l,r]to a constant value v using lazy propagation.

Thanks.

→ Reply


Al.Cash

2 years ago, # ^ | 0

The last example does exactly that, assignment on an interval.

→ Reply


sierra101

2 years ago, # | 0

how to scale values in a given range say [L,R] with constant C by segment tree or BIT . Thanks in advance :D

→ Reply


V12

2 years ago, # ^ | 0

I dont know for BIT but for segtree you can store additional `factor` for all nodes (initialized by 1). For each scale query, just scale this factor properties as always.

→ Reply


t3rminated

2 years ago, # | ← Rev. 2 0

i didn't get this part -

```
When processing the left border, the node we modify is always
the right child of its parent. Then all the previous

modifications were made in the subtree of the left child of
```

```
the same parent. Otherwise we would process the parent instead
of both its children. This means current direct parent is also
a parent of leaf l+n
```

→ Reply

abinash_nitr

2 years ago, # | 0

I am new to segment tree and algorithms in general. I see a difference in implementation of query function at http://codeforces.com/blog/entry/1256. In my observation query function in this post does not give correct result. Did anyone else notice that?

→ Reply

ErdemKirez

2 years ago, # | +5

I have a little question. In this problem, 533A — Berland Miners, I used this. I used this before, so I thought it works perfectly.

My submission with $N = 10^6 + 5$ got WA. When I changed it to $N = 2^{20}$, it took AC.

$N = 10^6 + 5$ --
> http://codeforces.com/contest/533/submission/12482628

$N = 2^{20}$ --
> http://codeforces.com/contest/533/submission/12483080

2^{20} is bigger than $10^6 + 5$ but problem isn't this. I tried it with $N = 2 * 10^6 + 5$but I got WA again.

$N = 2 * 10^6 + 5$ --
> http://codeforces.com/contest/533/submission/12483105

Do you have any idea about it?

→ Reply

Urbanowicz

2 years ago, # ^ | +5

As said in this post:

> *It's not actually a single tree any more, but a set of perfect binary trees*

Not sure, where exactly that fails for you, but I guess it's here:

```
int mn = min(op[x + x], cl[x + x + 1]);
```

Seems like you intend to take left child of `op` and right child of `cl`, but when you

have a set of trees it doesn't always work have a set of trees it doesn't always work.

→ Reply


ErdemKirez

2 years ago, # ^ | +5

I didn't understand the problem on here. Is it different from maximum of range or another segment operation?

→ Reply

Urbanowicz

2 years ago, # ^ | +5

But it works with $N = 3 \cdot 10^6 + 5$... 12484538

→ Reply


Al.Cash

2 years ago, # ^ | +5

I think the problem is in this line: `if(op[1] == 0)`

Node 1 is a true root only if N is a power of 2, otherwise it's not trivial to define what's stored there (just look at the picture).

Everything is guaranteed to work only if you use queries and don't access tree element directly (except for leaves).

→ Reply

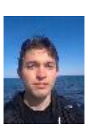
dadax85

2 years ago, # | +10

Great and efficient implementation. Could you please say more about "Modification on interval [l, r) affects t[i] values only in the parents of border leaves: l+n and r+n-1.". Because, the inc method modifies more than the parents of the border leaves.

If I well understood, you want to explain the fact that in query method (max implementation) we need only to push down ONLY to the left border and right border nodes. Since when computing the query we will be using only the nodes along the route. Moreover, this explains the fact that when we finish increment method, we need ONLY to propagate up to to root of the tree starting from the left and right border leaves, so that the tree and the lazy array will be consistent and representing correct values.

→ Reply


Al.Cash

2 years ago, # ^ | 0

You're right. I wanted to say what values are affected except the ones we modify directly in the loop (the ones that compose the interval). Will think how to formulate it the interval). Will think how to formulate it better.

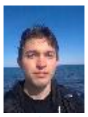
→ Reply

2 years ago, # | ← Rev. 2 0

rvelloso

```
// In case someone needs: query with [l, r] inclusive interval:
int query(int l, int r) {  // sum on interval [l, r]
  int res = 0;
  for (l += n, r += n; l <= r; l >>= 1, r >>= 1) {
    if (l&1) res += t[l++];
    if (!(r&1)) res += t[r--];
  }
  return res;
}
```

→ Reply



Al.Cash

2 years ago, # ^ | ← Rev. 2 0

If you prefer closed interval, just change `r += n` to `r += n+1`, otherwise it's easy to make mistakes.

→ Reply

Mr.Struggler

18 months ago, # ^ | 0

Is any problem to use query function mention by rvelloso? does it fails in any test case?

→ Reply

Mr.Struggler

18 months ago, # ^ | 0

int query(int l, int r) { int res = 0; for (l += n, r += n; l <= r; l >>= 1, r >>= 1) { if (l&1) res += t[l++]; if (!(r&1)) res += t[r--]; } return res; }

This implementation works fine. I solve a problemwith this implimentation.

→ Reply

RealNero

2 years ago, # | 0

You said that range modifications and point updates are simple, but the following test case doesn't give a correct example (if I understood the problem correctly): - 5 nodes - 1 2 4 8 16 are t[n, n + 1, n + 2, n + 3, n + 4] - one modification: add 2 to [0, 5) - query for 0

returns 59, while it should be 3.

Code (same as all the functions given above, but for unambiguity:

```
#include <cstdio>
```

```
const int N = 1e5;  // limit for array size
int n;  // array size
int t[2 * N];

void build() {  // build the tree
  for (int i = n - 1; i > 0; --i) t[i] = t[i<<1] + t[i<<1|1];
}

void modify(int l, int r, int value) {
  for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
    if (l&1) t[l++] += value;
    if (r&1) t[--r] += value;
  }
}

int query(int p) {
  int res = 0;
  for (p += n; p > 0; p >>= 1) res += t[p];
  return res;
}

int main() {
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%d", t + n + i);
  build();
  modify(0, n, 2);
  printf("%d\n", query(0));
  return 0;
}
```

Input: 5 1 2 4 8 16

→ Reply

AI.Cash

2 years ago, # ^ | 0

Just remove build — it doesn't belong to this example.

→ Reply

prvn

13 months ago, # ^ | ← Rev. 2 0

Why it does not belong to this example...should not we store the sum of the segment in its parent?

→ Reply

1theweblover007

2 years ago, # | 0

AI.Cash , I am finding the code inside the header of for-loop too confusing, Can you please tell me the easy and the common version of theirs ?

Like this :

```
for (l += n, r += n; l < r; l >>= 1, r >>= 1)
```

What does it mean ? Its tough for me to understand. Would be great if you could clarify this.

Thanks Alot. PS. Nice work on the tutorial, CF needs more articles like this one :)

→ Reply

yeputons

23 months ago, # ^ | +8

Step-by-step, in the simpler case (when $n = 2^k$ and we have full binary tree). Consider $n = 16$:

1. We have both `l` and `r` from $[0, 16]$, as outer world does not know anything about tree representation — it gives is bounds of the query.
2. First operation of `for` is executed exactly once before any iteration (as per definition). That is: add n (16 in our case) to both `l` and `r`, i.e. convert "array coordinates" to numbers of vertices in the tree.
3. Then we iterate while current segment is not-empty, that is $l < r$.
4. After each iteration we move 'up' the tree. We know that parent of vertex x is $\lfloor \frac{x}{2} \rfloor$, so we should divide both `l` and `r` by two. `>>= 1` means "bitwise shift by one bit", which works exactly like division by two with rounding down for non-negative integers.

→ Reply

saurabh.kakar05

23 months ago, # | ← Rev. 3 0

Hi, I am trying to solve SPOJ GSS1 according to this tutorial on Segement Trees. But I am not able to write a query method. Following is my code so far. I understood the logic but not able to write query method. Please help.

```
import java.util.Scanner;

public class GSS1_CanYouAnswerTheseQueriesI {


        static class SegNode {
                public SegNode(int left, int right, int
segsum, int bestsum) {
                        super();
                        this.left = left;
                        this.right = right;
                        this.segsum = segsum;
                        this.bestsum = bestsum;
                }
                public SegNode(){
                        this.left = Integer.MIN_VALUE;
                        this.right = Integer.MIN_VALUE;
                        this.segsum = Integer.MIN_VALUE;
                        this.bestsum = Integer.MIN_VALUE;

                }
```

```
		private int left,right,segsum,bestsum;
	};

	//array size
	static int n;
	static SegNode[] nodes;

	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		n = sc.nextInt();
		//Height of segment tree
	      int x = (int) (Math.ceil(Math.log(n) /
Math.log(2)));
	      //Maximum size of segment tree
	      int max_size = 2 * (int) Math.pow(2, x) - 1;
	      nodes = new SegNode[max_size];

		for (int i = 0; i < n; ++i){
			int in = sc.nextInt();
			SegNode node = new
SegNode(in,in,in,in);
			//nodes[n + i - 1] = node;
			nodes[n + i] = node;
		}


		build();
		int M = sc.nextInt();

		for(int i=0;i<M;i++){
			int l = sc.nextInt();
			int r = sc.nextInt();
			System.out.println(query(l,r));
		}

	}

	public static SegNode merge(SegNode cl,SegNode cr)
	{
		SegNode newNode = new SegNode();
		if(cl!=null && cr!=null){
			newNode.segsum = cl.segsum+cr.segsum;
			newNode.left =
Math.max(cl.segsum+cr.left,cl.left);
			newNode.right =
Math.max(cr.segsum+cl.right,cr.right);
			newNode.bestsum =
max3(cl.bestsum,cr.bestsum,cl.right+cr.left);
			return newNode;
		}
		if(cl==null){
			return cr;
		}else if(cr==null){
			return cl;
		}

		return newNode;
	}

	public static int max3(int a,int b,int c)
	{
		return Math.max(Math.max(a,b),c);
	}

	public static void build() {  // build the tree
		for (int i = n - 1; i >= 0; --i){
			nodes[i] = new SegNode();
			nodes[i] =
merge(nodes[i<<1],nodes[i<<1|1]);
			//nodes[i] =
merge(nodes[2*i+1],nodes[2*i+2]);
		}
	}

	public static int query(int l, int r) {  // sum on
interval (l, r)

		SegNode leftinMergeNode = null;
```

```
		SegNode rightinMerge = null;

		for (l += n-1, r += n-1; l <= r; l >>= 1, r
>>= 1) {
			//if l is the right child
			if ((l&1)>0) {
				leftinMergeNode =
merge(leftinMergeNode,nodes[l++]);
			}
			//if r is the left child
			if ((r&1)==0) {
				rightinMerge =
merge(rightinMerge,nodes[r--]);
			}

	}

		SegNode res =
merge(leftinMergeNode,rightinMerge);
		return res.bestsum;
	}
}
```

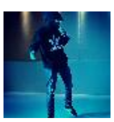
→ Reply

LaFuckinGioconda

23 months ago, # | 0

What a nice! I'm fucking wet.

→ Reply



punetharomil

23 months ago, # | 0

5 elements. (Read as 'count of 0 is 5') 0->5 1->0 2->0 3->0 4->0. Therefore for query <1, the answer should be 4 cuz 4 slots have value <1 (1,2,3 and 4).

according to this implementation when array is not of size 2^n, the tree is coming out to be wrong. n=5. so we start filling at n=5.

We have 4 slots with value [0,1) and 1 slot with value [5-6). The array would be like: (started filling at 5 cuz n=5, representing [0,1)) 0 1 2 3 4 5 6 7 8 9 10
0 0 0 0 0 4 0 0 0 0 1

now when making segment for array index 10, the value at index 5 gets overwritten. How to handle this?

→ Reply

eggeek

22 months ago, # | 0

To this line

```
Modification on interval [l, r) affects t[i] values only in the
parents of border leaves: l+n and r+n-1 (except the values
that compose the interval itself — the ones accessed in for
loop).
```

I have a question: If the **apply** operation does not satisfy the associative law, is it still work?

For example: `modify(3, 11, value1)`

For example: `modify(3, 11, value1)`

1 If I use **push(3, 11)**, operations on node 5 is:

1.1 apply(2, d[2]); (in the `push` loop);

1.2 d[5] += d[2] (in the `apply`);

1.3 d[2] is passed to 5's son, d[5] = 0 (in the `apply(5, d[5])` ;

1.4 d[5] = value (in the `modify` loop);

...

value is passed to 5's son 10 and 11, so:

1.5 **d[10] = (d[10] + d[2]) + value;** (same to 11)

2 If I use **push(3, 4), push(10, 11)**, operations on node 5 is:

2.1 apply(2, d[2]); (in the `push` loop);

2.2 d[5] += d[2]; (in the `apply`);

2.3 d[5] += value; (in the `modify` loop);

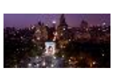
...

d[5] is passed to 5's son 10 and 11, so:

2.4 **d[10] = d[10] + (d[2] + value);**

1.5 is equal to 2.4, because (a+b)+c = a+(b+c), but if I replace the `+` to other special operation which does not satisfy the associative law, what should I do in `Lazy propagation` .

→ Reply


NYU

22 months ago, # | 0

Hi, I enjoy your post. Just wondering do you have templates for 2-D segment tree as well?

→ Reply


Arpa

22 months ago, # | 0

chz has post about data structures:Link.

And for segment tree:Link

And for segment tree.Link.

→ Reply

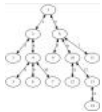

sheri.mori

21 month(s) ago, # ^ |

0

He is **amd** now

→ Reply


vatsalsharma376

9 months ago, # ^ |

0

He is **PrinceOfPersia** now.

→ Reply

-RooneY-

22 months ago, # |

← Rev. 2 0

N.A.

→ Reply

shyam81295

22 months ago, # |

0

I solved some questions based on this method (non-recursive segment trees) and it worked like a charm,but I think this method fails when building of tree depends on position of nodes for calculations, i.e. when there is non-combiner functions.
Example: https://www.hackerearth.com/problem/algorithm/2-vs-3/

Is it possible to solve this problem using above mentioned method ?

→ Reply


yeputons

22 months ago, # ^ |

0

Sure. Non-recursive bottom-top approach changes order of calculations and node visiting only. If that does not matter (and it does not if there are no complex group operations), you can apply all top-bottom tricks, including dependency on node's position. One way is to get node's interval's borders based on its id, another way is to simple 'embed' all necessary information into node inself, so it not only know value modulo 3, but also its length (or which power of 3 we should use when appending that node to the right).

→ Reply

shyam81295

22 months ago, # ^ |

0

Yes. Embedding would be useful addition in this method. Thanks for helping.

→ Reply

22 months ago, # |

0

the range modify function is not working can you

Ahmadshallouf

the range modify function is not working can you explain it please ?

void modify(int l, int r, int value) { for (l += n, r += n; l < r; l >>= 1, r >>= 1) { if (l&1) t[l++] += value; if (r&1) t[--r] += value; } }

→ Reply

21 month(s) ago, # | ← Rev. 3 0

sheri.mori

First thank you for this awesome tutorial.But one thing I was confused about ever since I read the first code was why do you always say : `if (r&1)` shouldn't it be `if ((r^1)&1)` ? because if r is odd then we have all of the children of its parent , so we do the changes to its parent rather than r itself.

→ Reply

21 month(s) ago, # ^ | ← Rev. 2 0

csssaz

remember that operations are defined as [l, r), so r always refers to the element in the right of your inclusive range.

→ Reply

21 month(s) ago, # ^ | 0

sheri.mori

Yeah. Thx. got it

→ Reply

21 month(s) ago, # | +8

niobium

This blog is brilliant! Can you also add a section on Persistent Segment Tree? We need to create new nodes when updating a node, how can it be done efficiently using this kind of segment tree? Thanks!

→ Reply

20 months ago, # | 0

theMonkeyKing

A wonderful implementation of segment tree . Thanks for this awesome article. If you write another blog or add a section in this blog on **persistent segment tree**(non-recursive implementation) that will be very helpful.

Thanks Again :) @Al.Cash

→ Reply

19 months ago, # | 0

YukimuraYukino

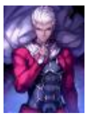
How to implement the query method for problems like this: 145E ?

I tried dividing the array into O(log n) disjoint

I tried dividing the array into O(log n) disjoint segment and then DP on them and got AC, but it made me implement a lot more thing, and I think it won't work if the problem asks to print answer for a segment [l; r], for example, GSS1 on SPOJ.

→ Reply

19 months ago, # | +8


adamant

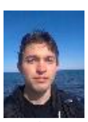
Hi, Al.Cash!

Frankly saying, I didn't watch the whole entry. But anyway I want to coin something.

The reason I use recursive implementation of segment trees besides that it is clear and simple is the fact that it is very generic. Many modifications of it come with no cost. For example it is the matter of additional 5-10 lines to make the tree persistent or to make it work on some huge interval like $[0;10^9]$. Your tree is heavily based on binary indexation. So, such modifications should be pretty hard. Am I correct?

→ Reply

19 months ago, # ^ | +23

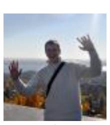

Al.Cash

True, is't impossible to modify this approach to support persistency or arbitrary intervals. However it handles all other cases better and they are the vast majority. Especially it's noticeable in the simplest (and the most common) case.

The choice is up to you, of course :)

→ Reply

19 months ago, # ^ | +5


NSV

BTW you can just use unordered_map instead of array in order to make it work on some huge interval like $[0;10^9]$. AFAIK it's well known trick used with Fenwick Tree.

→ Reply

19 months ago, # ^ | +5


adamant

Noooo. Very, very, very bad idea. The constant is too large even for Fenwick. Many problems where $[0;10^9]$ expect participant to compress the data instead of using dynamic structure, so it is usually

hard to get accepted even with fair

hard to get accepted even with fair dynamic tree. Using unordered_map instead of it is simply waste of time, in my opinion.

→ Reply


NSV

19 months ago, # ^ | +5

Yes, you are right, data compression of course better. But I guess that perfomance of BIT + unordered_map is not so much worse than performance of dynamic tree. From another hand it's extremely easy way to modify this data structure and it's also possible for some problems.

→ Reply

iskon

19 months ago, # | 0

can we find sum of elements(of different sets respectively) of a power set using segment tree? Explain.

→ Reply


kocko

19 months ago, # | ← Rev. 3 0

I'm not that much into C++, but shouldn't this statement

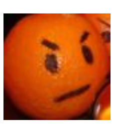
```
scanf("%d", t + n + i);
```

be actually:

```
scanf("%d", t[n + i]); // since t is an array
```

?

→ Reply


hellman_

19 months ago, # ^ | 0

Those are equivalent. In the first case C treats t as a pointer.

PS: in the second you would need to write `&t[n+i]`, that's why the first one is easier.

PPS: C/C++ is very crazy, `3["abcd"]` works and is equivalent to `"abcd"[3]`, this is shit.

→ Reply

17 months ago, # | 0

i am unable to solve this question using the above

AK_47_avi

I am unable to solve this question using the above implementation of segment tree. Can someone please provide me with a solution to this problem using this implementation. https://www.hackerearth.com/code-monk-segment-tree-and-lazy-propagation/algorithm/2-vs-3/

→ Reply

adarsh_1998

14 months ago, # | 0

I'm having difficulty in this problem -- >http://www.spoj.com/problems/DQUERY/the best i could think for a merge step is O(n) which would make my code run in O(n^2 log(n))+O(q*n*log(n)) definitely tle help me in improving my uppper bound;

→ Reply

hulk_baba

14 months ago, # | 0

I want to update every element E in range L,R with => E = E/f(E); I tried hard but can't write lazy propagation for it. function is LeastPrimeDivisor(E) . Can anybody help me?

→ Reply

dam_sehgal

14 months ago, # ^ | 0

Don't answer this question as it belongs to live contest

→ Reply

hulk_baba

14 months ago, # ^ | 0

Well I got it for the contest other way, but please answer when the contest is over.

→ Reply

suraj021

14 months ago, # | 0

Is lazy propagation always applicable for a segment tree problem ? Suppose we have to find the lcm of a given range, and there are range update operation of adding a number to the range. Is Lazy propagation applicable here?

→ Reply

aaaaajack

14 months ago, # ^ | 0

No. You should know how to fix the answer without knowing the exact elements in that range.

→ Reply

14 months ago, # ^ | 0

How do I optimize solution to such


suraj021

How do I optimize solution to such problems then? It will TLE for range updates (no lazy) for N <= 10^6.

→ Reply


aaaaajack

14 months ago, # ^ | 0

Can you provide the link or the full problem statement?

→ Reply


suraj021

14 months ago, # ^ | +10

I would but there is a similar problem in a live running contest, where I have to make range updates to a similar problem. I think i should discuss after the contest ends.

→ Reply


skavurskaa

14 months ago, # | 0

Could you please explain the theory behind the first query function? I get how it works but i don't know how to prove it is always correct. I am doing a dissertation on range queries and i am writing about iterative and recursive segment trees, but i have to prove why these functions are correct and i'm struggling right now.

Thanks in advance.

→ Reply


shahidul_brur

12 months ago, # | ← Rev. 3 0

Under "Modification on interval, single element access", in the query function, why we need to sum up the res? We are asked to give the ans 'What is the value at position p?'. We need to reply a single value, why we are summing multiple values in res?

```
//didn't get this part
 int query(int p) {
  int res = 0;
  for (p += n; p > 0; p >>= 1) res += t[p];
  return res;
}
```

I didn't understand. Can anyone please explain me?

→ Reply

4 months ago, # ^ | ← Rev. 2 0

That is something like lazy propagation

Mahilewets

That is something like lazy propagation.

For every segment, we have memorized, which value we have added for all elements belong to that segment.

If we just take tree[leaf], we would get **initial** value.

To get actual value we should apply all those modifications.

That means we should add all those modification flags belong to segments contain that particular element.

Look at recursive implementation of range add.

We return TREE[vertex] + answer for corresponding child segment.

In the iterative implementation, we are doing **absolutely** the same thing and we are doing it just in the reverse order.

→ Reply

saimor

12 months ago, # |

is it always give the right answer ??

int query(int l, int r) { // sum on interval [l, r)

int res = 0;

for (l += n, r += n; l < r; l >>= 1, r >>= 1) {

```
if (l&1) res += t[l++];

if (r&1) res += t[--r];
```

}

return res;

}

suppos int array[]={154,382,574,938,827,923,949,748,806,78};

if we build segment tree it will look like this t[1]=6379,t[2]=3117,t[3]=3262,t[4]=2581,t[5]=536,t[6]=1512,t[7]=1750,t[8]=1697,t[9]=884,t[10]=154,t[11]=382

int query(5,8);

int query(5,8);

answer should be 3426.

but it gives 2620

→ Reply

KADR

12 months ago, # ^ | +5

Right end should not be included into the sum.

→ Reply

saimor

12 months ago, # ^ | 0

Yes I knew it.But how i avoid right end.please explain.

→ Reply

nurlansofiyev

12 months ago, # ^ | ← Rev. 4 → 0

fucntion query(int l, int r) calculate sum of [l,r); it means if you want to calc sum of [5,8] you must call query(5,9). if you want your function calculate interval [l,r] you must implement by this way :

int sum_f(int l, int r) {

int sum = 0;

l += n, r += n;

while (l <= r)

{

```
if (l & 1) sum += v[l] ;
if (!(r & 1)) sum += v[r] ;
l = (l + 1) / 2, r = (r - 1) / 2;
```

}

return sum;

}

→ Reply

sheep0x

12 months ago, # | 0

For the sake of breaking language barrier, this bottom-up implementation is called “zkw segment tree” in China, because it was (supposedly) independently discovered by Zhang

Kunwei in his famous paper. Not sure who was

Kurfwei in his famous paper. Not sure who was the earliest inventor though.

→ Reply

haleyk100198

11 months ago, # | 0

Is there a way that we could do better for non-commutative segment trees? For this implementation sometimes node are just not being used, as extra merging were done. (eg: for size 13 in the example shown, we merged 2 and 3 even though it is meaningless)

→ Reply

sgc109

11 months ago, # | ← Rev. 7 0

I just wonder if i could make arbitrary size of array to some nearest power of 2 if i put useless values which don't affect to final result of each query to the rest of indices? For example, making a size of array from 13 to 16, or 29 to 32 and putting -inf for the additional indices in a max-segtree(or 0 in a sum-segtree)

→ Reply

haleyk100198

11 months ago, # ^ | 0

Sure you can.

→ Reply

sgc109

11 months ago, # ^ | 0

thank you for your reply. But I'm just wondering how people usually implement codes for range operation. For example, fenwick tree with lazy propagation for some situation or non-recursive segment tree(like this article) or recursive segment tree. Which one is the most simple for specific situation? I just heard that it is easier to implement segment tree with lazy propagation with recursive way. So I just wanna know.

→ Reply

BasselBakr

11 months ago, # ^ | 0

Implementing lazy propagation in recursive way is much easier yet less efficient than iterative approach

I would go with Fenwick Tree if it's sum/xor or something similar and with iterative segment tree otherwise unless segment tree otherwise unless the update function is hard to implement and debug in time

→ Reply

sgc109

11 months ago, # ^ | Rev. 3 0

So you would go Fenwick Tree whenever you have to do range sum/xor stuff even if you need lazy propagation and would go iterative segtree if there is no need of lazy propagation or complicated stuff in operations except for sum/xor and recursive way for the others, right?

→ Reply

BasselBakr

11 months ago, # ^ | Rev. 2 0

Right :D

There is also a trick to do both range queries and lazy range updates with Fenwick tree here

→ Reply

Dpman

8 months ago, # ^ | 0

Can you explain how does it handle both of them more clearly?

→ Reply

Rezwan.Arefin01

8 months ago, # ^ | 0

Here is a good explanation — GeeskForGeeks — Binary Indexed Tree : Range Update and Range Queries

→ Reply

10 months ago, # | 0

How to find index of segment tree by given sum xsc

How to find index of segment tree by given sum.
example acm.timus.ru 1896

```
// Following code Gives WA38. How to fix it ??
const int N = 1.1E+6;
int tree[N + N]; //  tree[ n .. n + n ] = only 0 or 1
int find_index(int sum)
{
   int r = 1;
  for(  ; r < n; r <<= 1) if ( tree[r] < sum ) sum-= tree[r ++
] ;
   return r - n;
}
```

→ Reply

10 months ago, # ^ | 0


xsc

I mean, need find that i, which query(0,i) == sum.

→ Reply

9 months ago, # | 0

kanchicoder

What's wrong for **Sum of given Range**

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
ll t[(int)2e6+1], d[(int)1e6+1];
ll h, n;
void apply(ll i, ll v) {
        t[i] += v;
        if(i < n) d[i] += v;
}
void build(ll i) {
        ll p, count;
        while(i > 1) {
                i >>= 1;
                p = i;
                count = 0;
                while((p <<= 1) <= 2*n-1)
                        count++;
                t[i] = t[i<<1] + t[i<<1|1] + (d[i]<<count);
        }
}
void push(ll i) {;
        for(ll s = h; s > 0; --s) {
                ll p = i >> s;
                if(d[p]) {
                        apply(p<<1, d[p]);
                        apply(p<<1|1, d[p]);
                        d[p] = 0;
                }
        }
}
void update(ll l, ll r, ll value) {
  l += n, r += n;
  ll l0 = l, r0 = r;
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1) apply(l++, value);
    if (r&1) apply(--r, value);
  }
  build(l0);
  build(r0 - 1);
}

ll query(ll l, ll r) {
  l += n, r += n;
  push(l);
  push(r - 1);
  ll res = 0;
  for (; l < r; l >>= 1, r >>= 1) {
    if (l&1) res +=  t[l++];
    if (r&1) res +=  t[--r];
  }
```

```
  return res;
}
int main() {
        // your code goes here
        ll te, m, ch, l, r, v;
        cin >> te;
        while(te--) {
                cin >> n >> m;
                memset(t, 0, sizeof(t));
                memset(d, 0, sizeof(d));
                h = 64 - __builtin_clzll(n);
                while(m--) {
                        cin >> ch;
                        if(!ch) {
                                cin >> l >> r >> v;
                                update(--l, r, v);
                                print();
                        }
                        else {
                                cin >> l >> r;
                                cout << query(--l, r) << endl;
                                print();
                        }
                }
        }
        return 0;
}
```

→ Reply

9 months ago, # | 0

hunter1703

Please, can someone explain apply and build functions in lazy propagation in max queries...?? Really need help there.

→ Reply

xsc

9 months ago, # ^ | +3

Some theory: initial: we have array a[0...n-1] — initial array. t[0 ... n + n] — segment tree array. d[0...n + n] — lazy propogation array. d[i] = 0 (i=0..n+n-1) initially.

1) t[i + n] == a[i], if 0 <= i < n. So if a[i] increments a `value` , t[i+n] also increments the `value` .

2) t[i] = max(t[i * 2] , t[i * 2 + 1]) , if 0<= i < n. we call t[i] as i-th root of segment tree, because, t[i] = max{ a[i * 2^k + h — n], where i*2^k >=n, and 0<=h<2^k } for example, n = 8. so t[3] = max{ a[3*4 — 8], a[3*4 + 1 — 8], a[3*4+2 — 8], a[3*4+3-8]} = max{ a[4], a[5], a[6], a[7] }

3) d[i] = 0, if there not increment operation in the i-th root. d[i] > 0, need increment all of t[i*2^k + h] element by d[i]. see below.

4) if t[i*2] and t[i*2+1] — incremented same `value` so, t[i] — maximum of t[i*2] , t[i*2+1] also incremented to `value` . It's

t[i*2+1] also incremented to `value`. It's applied recursive.

5). if need increment i-th root to `value`, by standard algorithm need increment t[i], t[i*2], t[i*2+1], t[i*4], t[i*4+1], t[i*4+2], t[i*4+3], ..., t[i*2^k+h] nodes to `value. but, with lazy propogation, all these operations absent. Need only increment t[i] and d[i] to` value`. d[i] > 0 means, t[i*2^k+h] values will incremented some later.

```
   For example, n = 8. a[] = {3, 4, 1, 0, 8, 2, 5, 3}
and need increment a[4..7] elements to 5.
   t[8..15] = a[0..7] = { 3, 4,   1, 0,   8, 2,   5, 3
}
   t[4..7 ]             {   4       1       8       5
}
   t[2.. 3]             {       4           (8+5 d:5)
}
   t[1]                 {               8
}
   a[4] ==> t[4+8] = t[12], so need increment t[12],
t[13], t[14], t[15] to 5, also need increment  t[6] ,
because t[6] = max(t[12], t[13]), and need increment
t[7], t[7] = max(t[13], t[14]), at least need
increment t[3] , because t[3] = max(t[6], t[7]).
total: 7 increments.
   but with lazy propogation t[3] = root of all t[6],
t[7], t[12], t[13], t[14], t[16]  nodes. so need
increment only   t[3]  to 5, and increment d[3] to 5.
this operation is `apply`.

but after `apply` increment operation,  t[1] - become
incorrect value, it should be max(4, 8+5) = 13.  --
this is `build` operation.
```

GoOd LuCk.

→ Reply

hunter1703

9 months ago, # ^ | 0

Thank you very much...... Such a great help.... I was really stuck there.

→ Reply

hunter1703

9 months ago, # ^ | 0

Thanks for yur help. Can you tell how build and push works in sum queries (It says new build and push methods incorporated both types of build and push methods mentioned before). I can't seem to get it. Also why does both calc and apply update t[p]. NO understanding these both methods...

By compacting the code (by author), code longer is simple in

understanding although it is simple in

understanding although it is simple in implementing....

⟶ Reply

9 months ago, # ^ | 0

xsc

Which :

1) update: Assignment in range, and query: sum in range ??? OR

2) update: Increment in range, and query : sum in range ??

⟶ Reply

9 months ago, # ^ | 0

rahulhpatel1998

increment in range query.

⟶ Reply

xsc

9 months ago, # ^ | +8

This entry explains for Assigment in range, sum in range — and it is difficult than increment operation.

But, If you want increment in range, it simple than assignment.

Let Theory, again; 1<=n<=N ~ 10^5. a[0], a[1], ... , a[n-1] — given array. Need apply two operations:

1. increment a[L], a[L+1],... a[R-1] to `value`, i.e [L..R) range.
2. find sum a[L] + a[L+1] + ... + a[R-1] , i.e. [L..R) range.

t[0..n+n] — segment tree array. d[0..n+n] — lazy propogation.

where

where

t[i] = a[i — n], n<=i < n+n. and

t[i] = t[i*2] + t[i*2+1], for 0<=i<n.

1. If a[i] incremented by `value`, so t[i+n] also incremented by `value`.
2. if t[i*2] and t[i*2+1] incremented by `value`, they root t[i] incremented `2* value`. and its applied recursive down.

In general, if all `leaf` nodes of i-th root incremented by `value`, so t[i] should increment the `value` multiply number of `leaf` children of i-th root.

for example: n = 8. if a[4] , a[5], a[6], a[7] increment by 5, so t[12], t[13], t[14], t[15] incremented by 1*5, t[6], and t[7] — incremented by 2*5, and t[3] incremented by 4*5. t[3] — has 4 children.

1. Now, about lazy propogation d[] array. d[i] == 0, means there no increment operation i-th

operation i-th root. d[i] > 0, means all `leaf` nodes from i-th root of segment tree will increment by d[i], and applied 2.step operation recursivily.

```
If we need increment
[L..R)  range by `value`
, need increment only
root nodes by  `value` *
children(i), and d[i]
increments by `value`.
Other nodes will
calculated some later.
```

For example: if need increment a[4..7] to 5. There only increment t[3] to 5*4, and put d[3] = 5. This `apply` operation. Other nodes t[6], t[7], t[12], t[13], t[14], t[15] nodes will incremented some later (may never).

```
#define N 100008

int t[N+N];
int d[N+N];
//int ch[N+N]; // number
of children .
int n;

void apply (int p, int
value, int children)
{
    t[ p ] += value *
children;
    if (p < n ) d[p] +=
value.
}

void build(int p) // fixes
applied nodes and down
nodes.
{
    int children = 1;
     while (p > 1)
     {
           p = p  /  2 ;
           children =
children * 2;
           t[p] = t[ p *
2] + t[ p * 2 + 1] + d[ p
] * children;
     }
}
```

```
void push(int p) //
calculate lazy operations
from p nodes.
{  // h = log2(n).
     for(int s = h ; s > 0
; --s)
     {
            int i = p >>
s; // i = p / 2^s
            if ( d[i] != 0
)
            {
 // so i*2 and i*2+1 nodes
will increase by
//  d[i] * children,
where children = 2^(s-1)
                   int
children = 1 << (s-1);

apply(i*2, d[i],
children);

apply(i*2+1, d[i],
children);

              // now d[i]
- is dont need no more,
clear it.
                  d[i] =
0;
            }
     }
}


int increment(int L, int
R, int value)
{
     L += n, R += n;
     int L0 = L, R0 = R;
//presave they.
for( int children = 1 ; L
< R ;  L/=2, R/=2,
children *= 2)
     {
           if (L % 2 == 1)
// L - root node
              apply(L++,
value, children);

           if ( R % 2 ==
1) // R - 1  root node
              apply(--R,
value, children);
      }

    // now fix applied
nodes and downsite nodes.
     build(L0);
     build(R0-1);
}


int sumRange(int L, int R)
{
      // need calculate
applied nodes from L and
R.
      L += n, R+= n;
      push(L); push(R-1);
// it clears all lazy d[]
nodes.
      int sum = 0;
      for( ; L < R; L /=
2, R/=2)
      {
            if (L % 2 ==
```

```
1) sum += t[L++];
            if (R % 2 ==
1) sum += t[ --R];
      }
    return sum;
}
```

⟶ Reply

hunter1703

9 months ago, # ^ | 0

You are great, man!!! Really. Very very much helpful

⟶ Reply

hunter1703

9 months ago, # ^ | 0

Last very silly question. What if n is not power of 2. Do we have to first increase n to the nearest power of 2 and fill values with zeroes or there is a way getting around it??

⟶ Reply

xsc

9 months ago, # ^ | ← Rev. 3 0

Hmm, Did not test for n != 2^k . There may some bug when calculate children of i-th root, iff n != 2^k, be careful.

In competitive programming, I, always, use n = 2^k.

For example:

For example:
example

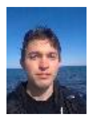
→ Reply

hunter1703

9 months ago, # ^ | 0

Thanks man for all your help. I was stuck here for very long time......

→ Reply



Al.Cash

9 months ago, # ^ | 0

All the codes in the article work for any n.

→ Reply

hunter1703

9 months ago, # ^ | 0

but for n!=2^n wouldn't there be one entry left which will have parent whose range will be 2 already and on adding this additional child its range become (2+1) ????

Thanks in advance

→ Reply

is-it-down

9 months ago, # | ← Rev. 2 0

Hi everyone ... I was trying a 2D Segtree question Census I am getting stuck in the query function. Can someone tell me how to implement query using the method in the post.

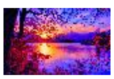
→ Reply

avisheksanvas

9 months ago, # | 0

Could you explain how the query function for this question would be? 380C - Sereja and Brackets I've done it with the recursive approach, but am not able to code the query function using this method.

→ Reply

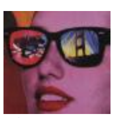


kunnu96

8 months ago, # ^ | +5

I too have encountered the same problem.BRCKTS is the easier version of 380C — Sereja and Brackets but how to modify query function according to the problem as t[p>>1] = t[p] + t[p^1] logic will work for only all problems related with associative operations but here it is not.So I solved this problem by changing the logic to t[p]=combine(t[i << 1], tree[i << 1 | 1]) where p goes from (n+p)/2 to 1 in reverse for loop.Anyone has much more easier modification idea please share.

→ Reply

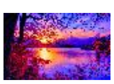


CountZero

8 months ago, # ^ | 0

my incorrect solution of that problem: 5684965
how I fixed it: 5685269

→ Reply



kunnu96

8 months ago, # ^ | +5

Actually in 380C — Sereja and Brackets no update type of query is present so no need of modify function is there which will be easy to do with the above mentioned optimized segment tree implementation. Try this BRCKTS using the same modify function as mentioned by **Al.Cash** you will realize it will not work in that way.

→ Reply

CountZero

8 months ago, # ^ | 0

there are at least 3 ways to solve this problem.
if memory is not an issue then you can just set array length to the power of 2
modify query function: 6026442
split queries: 6006570

→ Reply

f2lk6wf90d

6 months ago, # | 0

I implemented a generic segment tree based on this article: Code. Usage: simply write two structures `struct node` and `struct node_update` representing your tree nodes and updates, then provide function pointers that merge two nodes together, apply an update to a node, and merge two updates.

Example: 27103823

→ Reply

itachi_2016

5 months ago, # | ← Rev. 3 +1

As a noob I am still confused that what is the requirement of modify() function. As you are just doing the same thing that you did in build() function. Please explain.... But I must say you explained the whole concept in a nice manner..

→ Reply

itu

5 months ago, # ^ | ← Rev. 2 +5

The idea is that modify() runs in O(logn) so doing it for every point is O(nlogn). build(), on the other hand, does the same thing but runs only in O(n).

This might not be significant in simple cases but will make a difference for complicated operations, or when num_queries < n.

→ Reply

donbox5

5 months ago, # | ← Rev. 4 0

noob here, Say the input is of size 6

Following parent child relationships are made:

```
5->  left child 10,  right child 11
4->  left child  8,  right child 9
3->  left child  6,  right child 7
2->  left child  4,  right child 5

1->  left child  2,  right child 3
```

```
Now, we can see in segment array,  1 has 2 as left child   and
3 as right child


            1
         /      \
       /          2
      /        /     \
    3         4        5
   /  \      /  \   /   \
  6    7    8   9   10   11


// Should not 2 ideally be a right child of 1
```

If the usage context is such that the rules are different for processing left and right childs, then we will have wrong result, since node 2 should ideally be processed as a right child .

e.g. http://www.spoj.com/problems/BRCKTS/

→ Reply

inception_95

4 months ago, # | 0

How can I find gcd(L,R) with the non-recursive segment tree?

→ Reply

shashwatchandra

3 months ago, # | 0

Can somebody please explain what the bitwise operators do here cause its very difficult to catch up?

P.S — I know about Bitwise operators but have never used them in my code.

Thanks.

→ Reply

f2lk6wf90d

3 months ago, # ^ | +3

If p is a node in the tree, then `p >> 1` is its parent, `p >> s` is its s^{th} ancestor, `p << 1` is its left child, and `p << 1 | 1` is its right child.

→ Reply

shashwatchandra

3 months ago, # ^ | 0

Thank you soo much!!!

→ Reply

ilovema

43 hours ago, # | 0

Is Binary Search impossible like classic Segmented Tree ?

int ask(int id, int ID, int k, int l = 0 int r = n){// id is int ask(int id, int ID, int k, int l = 0,int r = n){// id is the index of the node after l-1-th update (or ir) and ID will be its index after r-th update if(r — l < 2) return l; int mid = (l+r)/2; if(s[L[ID]] — s[L[id]] >= k)// answer is in the left child's interval return ask(L[id], L[ID], k, l, mid); else return ask(R[id], R[ID], k — (s[L[ID]] — s[L[id]]), mid, r);// there are already s[L[ID]] — s[L[id]] 1s in the left child's interval }

http://codeforces.com/blog/entry/15890

→ Reply

41 hour(s) ago, # ^ | ← Rev. 2 0

Jakube

If n is a power of 2, than you can use the identical code as in the classic recursive Segment Tree.

If n is not a power of 2, it is still possible, but it is definitely harder.

One possible approach: - collect all indices of the segements that partition the query segment (be careful here, the iterative version alternatively finds such segments from the left and right border): there are O(log n) many - then iterate over them from the most left segment to the most right one, until you find the segment that contains the element you want to find. - then continue traversing to either the left or right child until you reach the leave node (this can be done iteratively or recursively).

→ Reply