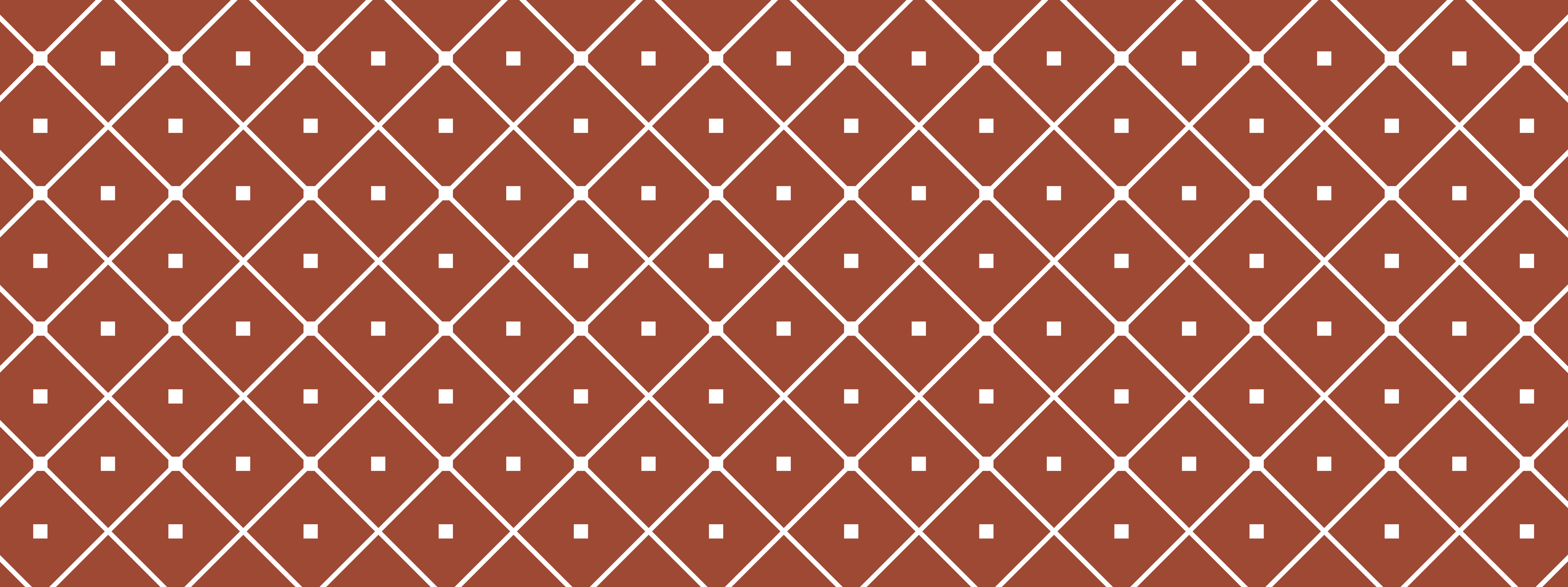
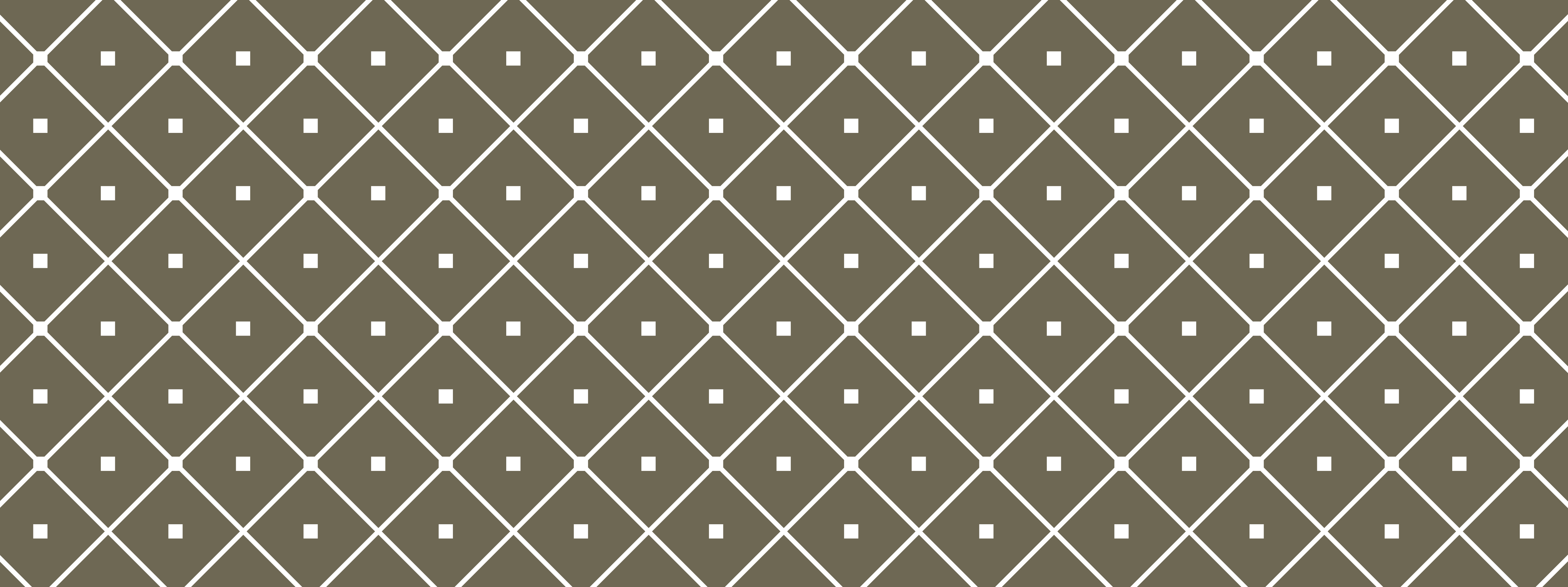


SEGMENT TREES & RELATED TOPICS

Tian Cilliers, Training Camp 2,
9-10 February 2019



INTRODUCTION

Why do we even use segment trees?

SAMPLE PROBLEM

You have a list of military boats with varying sizes, and in order to intimidate your opponents, you want to answer your superiors asking for the sum of the sizes of all boats in a certain range. Let N be the number of boats and Q be the number of queries. Brute forcing by looping over the range for each query runs in O(NQ) time.

5+9+23+7 = 44

POSSIBLE SOLUTION

Precalculate prefix sums, in other words the sum of the sizes of all boats up to a certain boat. This can be done in O(N) time by keeping a running sum. The sum of a range can then be calculated by subtracting the prefix sum one left of the leftmost item from the prefix sum up to the rightmost item. This totals to O(N) precomputing and O(1) runtime per query, for a total of O(N+Q).

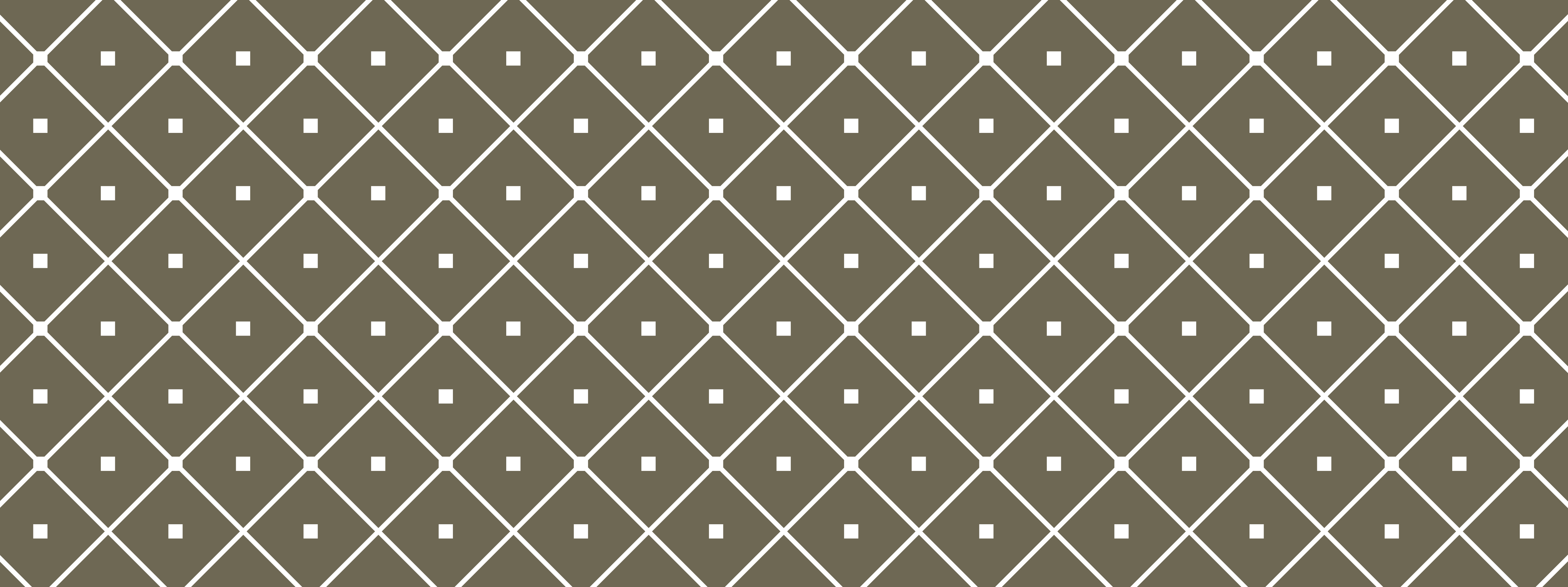
10	1	5	9	23	7	7	11
10	**11**	**16**	**25**	**48**	**55**	**62**	**73**

55-11 = 44

PROBLEMS WITH SOLUTION

As we can see, the proposed solution runs quite quickly, but has a few limitations. Firstly, if we needed the maximum of a range (RMQ) instead of the sum, prefix sums would not work. More importantly, however, prefix sums are unable to handle updates to the items. If any item in the list is updated, all the prefix sums including the item needs to be updated.

If the problem is modified by stating that some amount U of updates to specific items are performed in between queries, the time complexity would be increased to O(UN+Q). Is this able to be improved on? Can we devise a fast way to do RMQ as well?

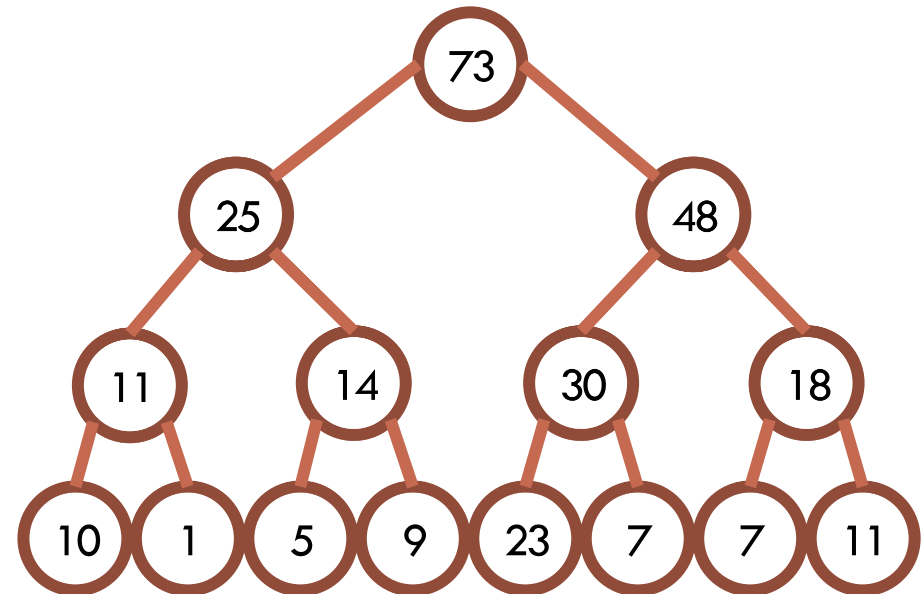


SEGMENT TREES

In all their O(logN) query and update beauty

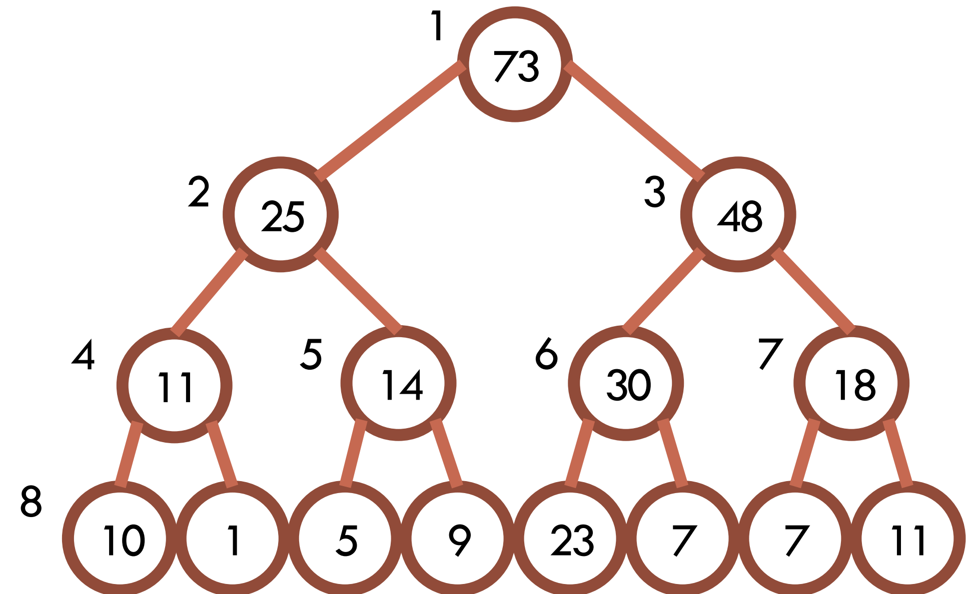
WHAT IS A SEGMENT TREE?

A segment tree is simply a binary tree, that is, a tree where each node has two child nodes. In a segment tree, the leaves represent data values, and each non-leaf node represents some associative operation on its children (for example the maximum or sum of its children).



EXAMPLE IMPLEMENTATION

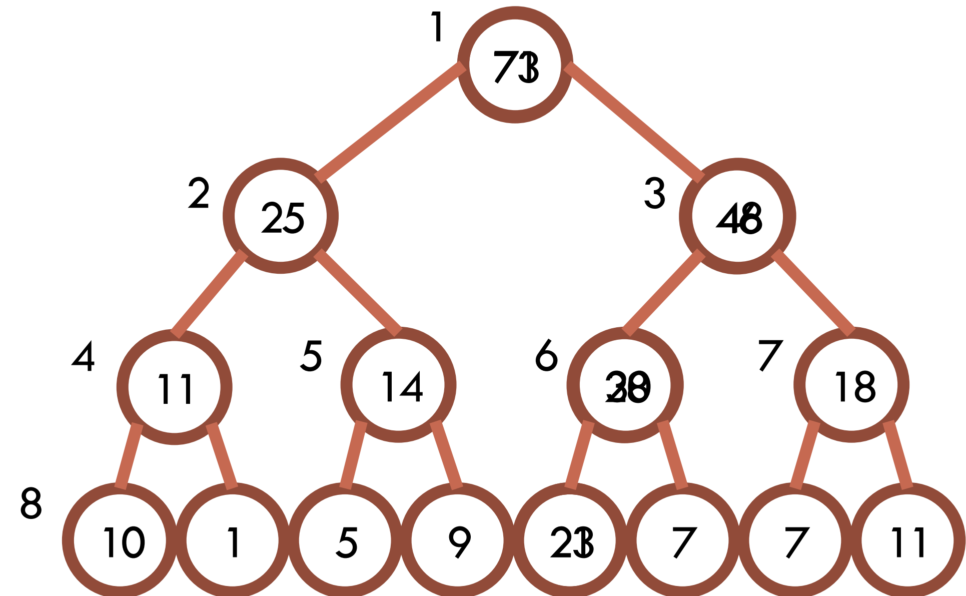
To implement the data structure, a simple array with two times the length of the amount of data values required could be used. Rooting the tree with the index 1 also allows easy movement between parent and child nodes by dividing by two and getting the floor of that to find the parent index.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
73	25	48	11	14	30	18	10	1	5	9	23	7	7	11

UPDATING A SINGLE ITEM

Since each parent node is reachable by using the floor of the child node's index divided by two, climbing the tree while updating is trivial. Care must be taken, however, to initially convert between the index of the value and the index in the tree. Due to the structure of the tree, updating can thus be done in O(logN) runtime.



EXAMPLE UPDATING CODE

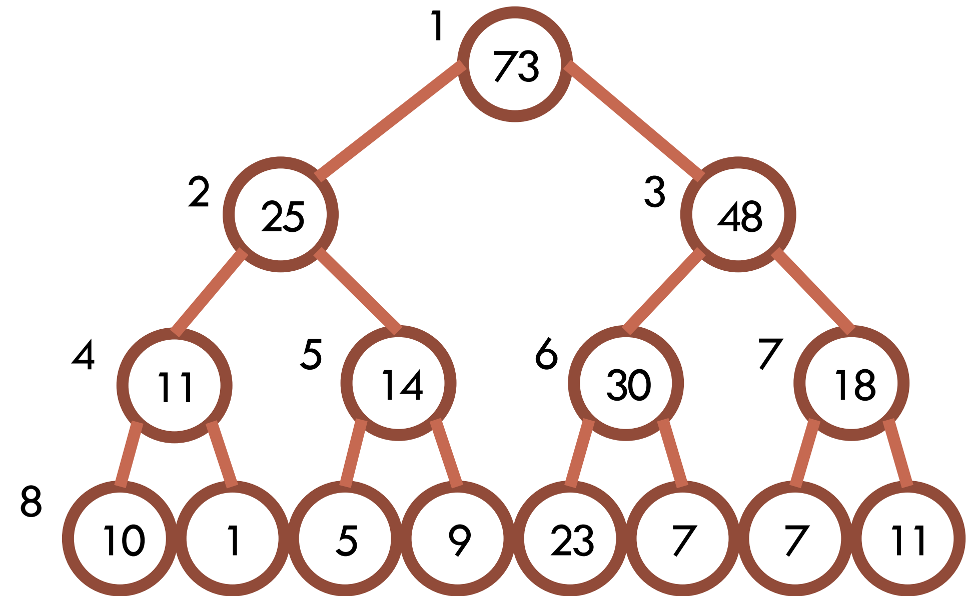
```cpp
#include <bits/stdc++.h>
#define MAXN 8
using namespace std;

int SEGTREE[2*MAXN];

void update(int node, int value) {
  node += MAXN;
  SEGTREE[node] = value;
  node /= 2;
  while (node >= 1) {
    SEGTREE[node] = SEGTREE[2*node] + SEGTREE[2*node+1];
    node /= 2;
  }
}
```

QUERYING A RANGE

The key when querying is to include the highest nodes that completely covers part of the queried range. This can be done by keeping a left and right pointer, if necessary processing the node and shifting the pointer inward to be on the outer node of a parent, moving both up a level, and repeating. Again, this takes O(logN) time.



EXAMPLE QUERYING CODE

```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4
5   int SEGTREE[2*MAXN];
6
7   int query(int left, int right) {
8     left += MAXN;
9     right += MAXN;
10    int sum = 0;
11    while ((left <= right) && (left >= 1)) {
12      if ((left % 2) == 1) {
13        sum += SEGTREE[left];
14        left++;
15      }
16      if ((right % 2) == 0) {
17        sum += SEGTREE[right];
18        right--;
19      }
20      left /= 2;
21      right /= 2;
22    }
23    return sum;
24  }
```

TIME AND SPACE COMPLEXITY

As we saw, both updating a single value and querying a range takes O(logN) runtime. In addition, it can be seen that the space required is 2N, or O(N).

This means that the problem mentioned earlier could be run in O((Q+U)logN) time instead of O(UN+Q) when using prefix sums.

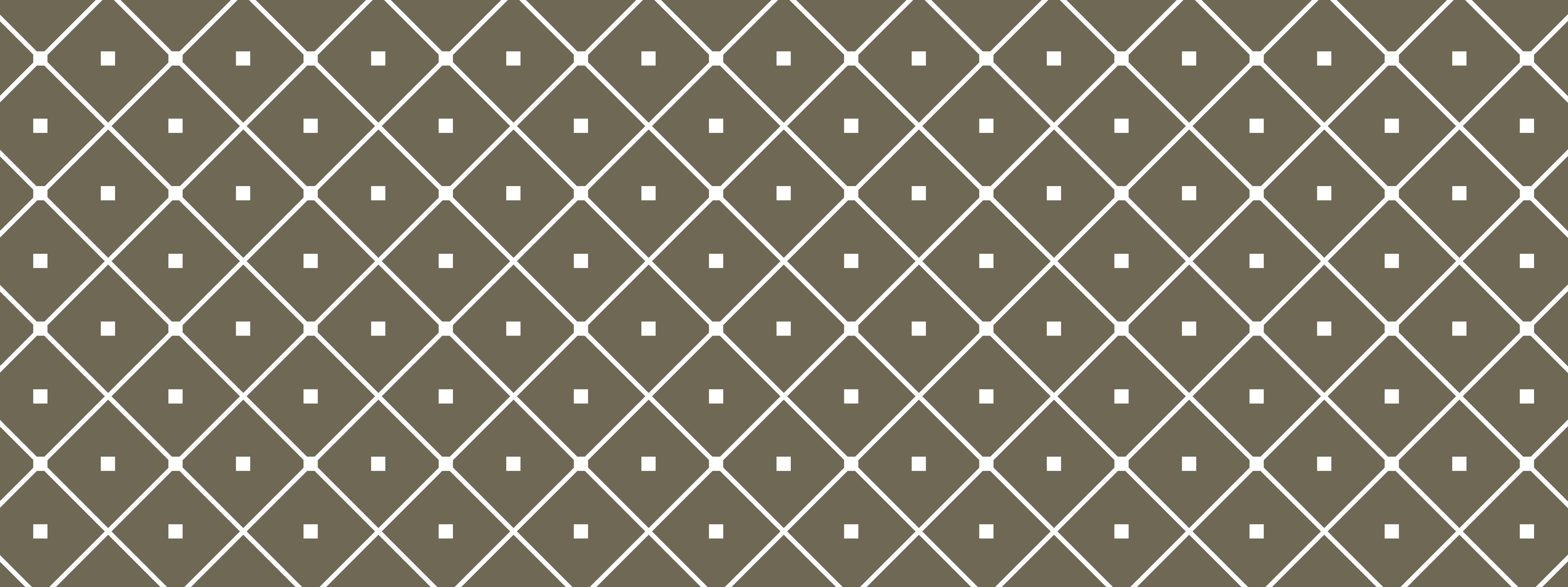
CONDENSED CODE

```
#include <bits/stdc++.h>
#define MAXN 8
using namespace std;

int SEGTREE[2*MAXN];

void update(int node, int value) {
  SEGTREE[node += MAXN] = value;
  while ((node /= 2) >= 1) SEGTREE[node] = SEGTREE[2*node] + SEGTREE[2*node+1];
}

int query(int left, int right) {
  left += MAXN; right += MAXN;
  int sum = 0;
  while ((left <= right) && (left >= 1)) {
    if ( (left &1)) sum += SEGTREE[left++];
    if (!(right&1)) sum += SEGTREE[right--];
    left /= 2; right /= 2;
  }
  return sum;
}
```

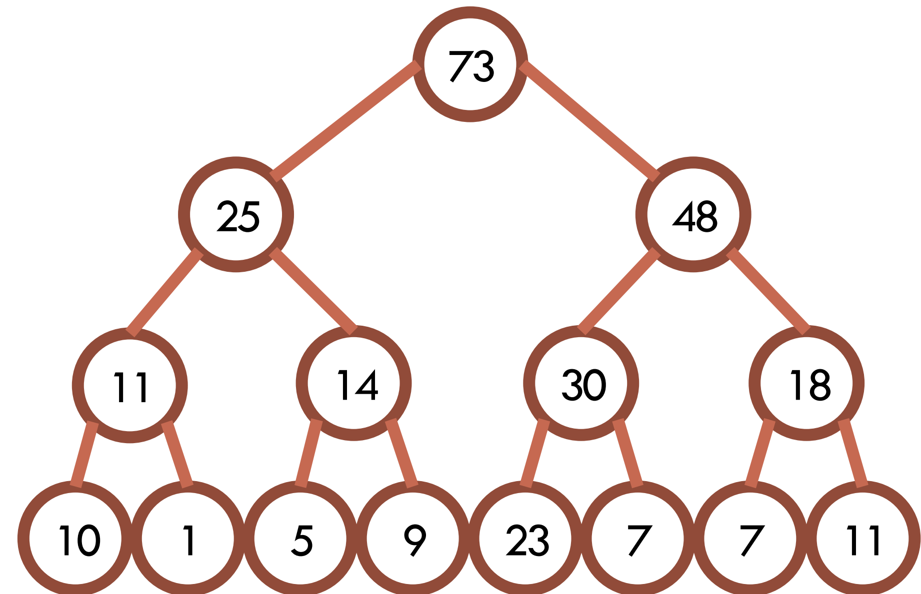


FENWICK TREES

Take segment trees and add a sprinkle of binary magic

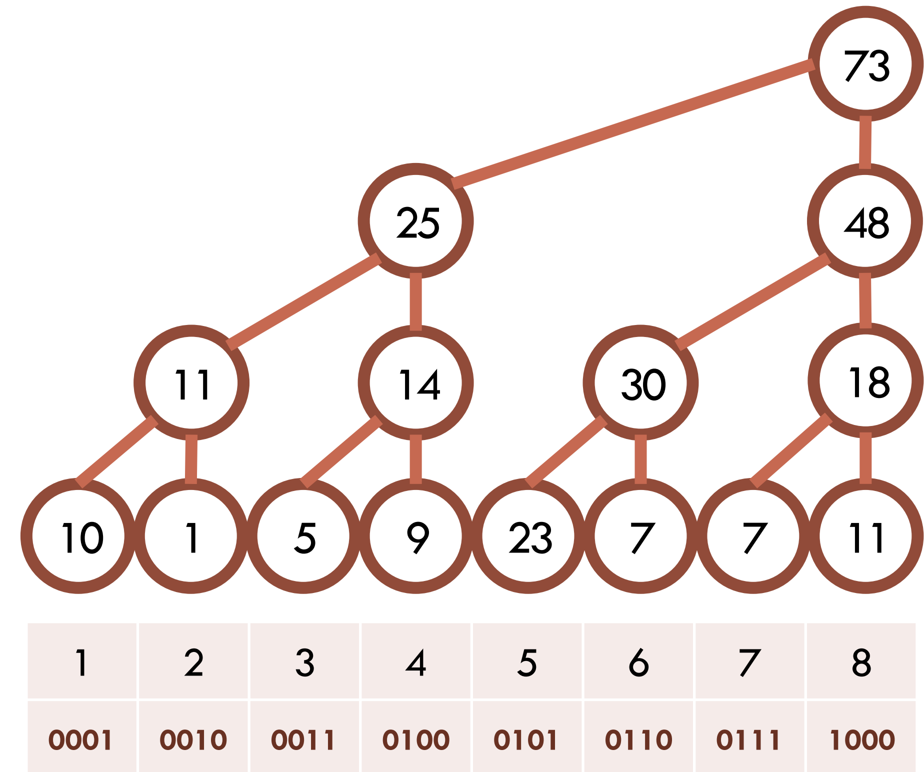
WHAT IS A FENWICK TREE?

Fenwick Trees, also known as Binary Indexed Trees, are a type of segment tree. To understand them, it is necessary to realize that segment trees storing a reversible operation, like sums, are in fact containing redundant information. When a node contains the sum of its children, one of the child nodes is unnecessary, since its value can be calculated. Fenwick Trees allow similar O(logN) updating and querying, but with a simplified query implementation.



EXAMPLE IMPLEMENTATION

These remaining nodes can them be organized into a specific structure and be stored in an array with length N. Considering the binary representation of the indices of each node leads to an important observation that will be useful when doing updates and queries.

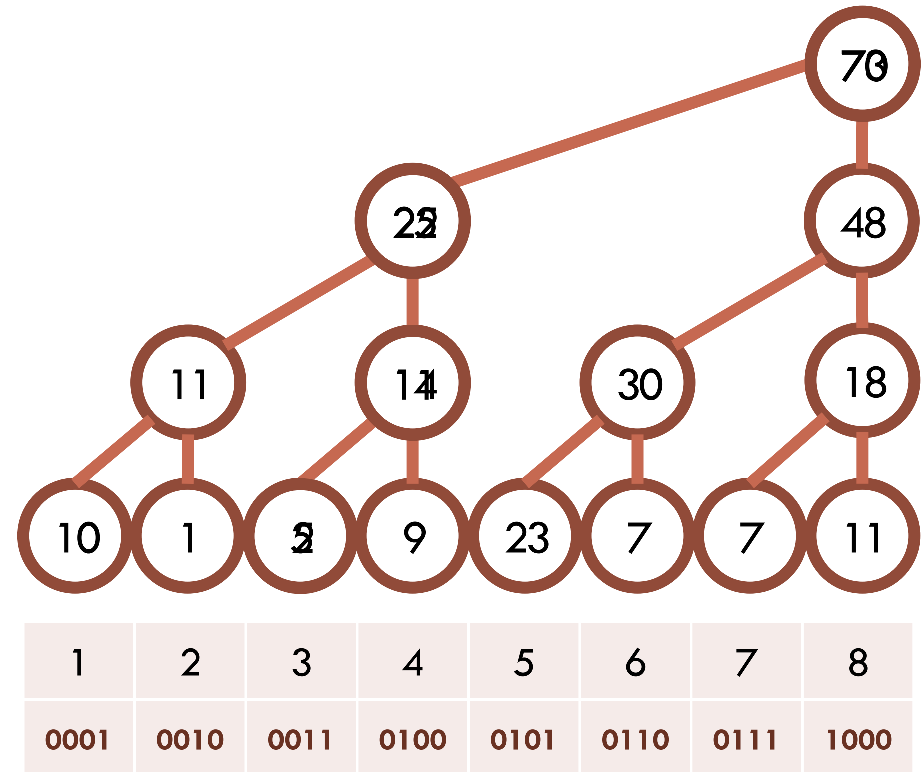


UPDATING A SINGLE ITEM

First, we just do an update like with a regular segment tree, keeping in mind that some nodes do not need to be visited since their values can be calculated using the rest.

Notice that the binary representation of the indices of nodes we updated was as follows: 0011 > 0100 > 1000

It can further be seen that, to update any index, the least significant bit has to be added until the root is reached.



FINDING THE LEAST SIGNIFICANT BIT

To find the LSB, we use the fact that, when subtracting one from a number, the LSB changes to zero and all the zeroes following the LSB are changed to ones. When inverting this result, the only bit that is on in the original and this inverted result is the LSB, since the bits before the LSB are all inverted and the bits following the LSB are all zero in the result.

1001011000 (original number)

1001010111 (subtracted one)

0110101000 (inverted)

0000001000 (bitwise and with original)

EXAMPLE UPDATING CODE

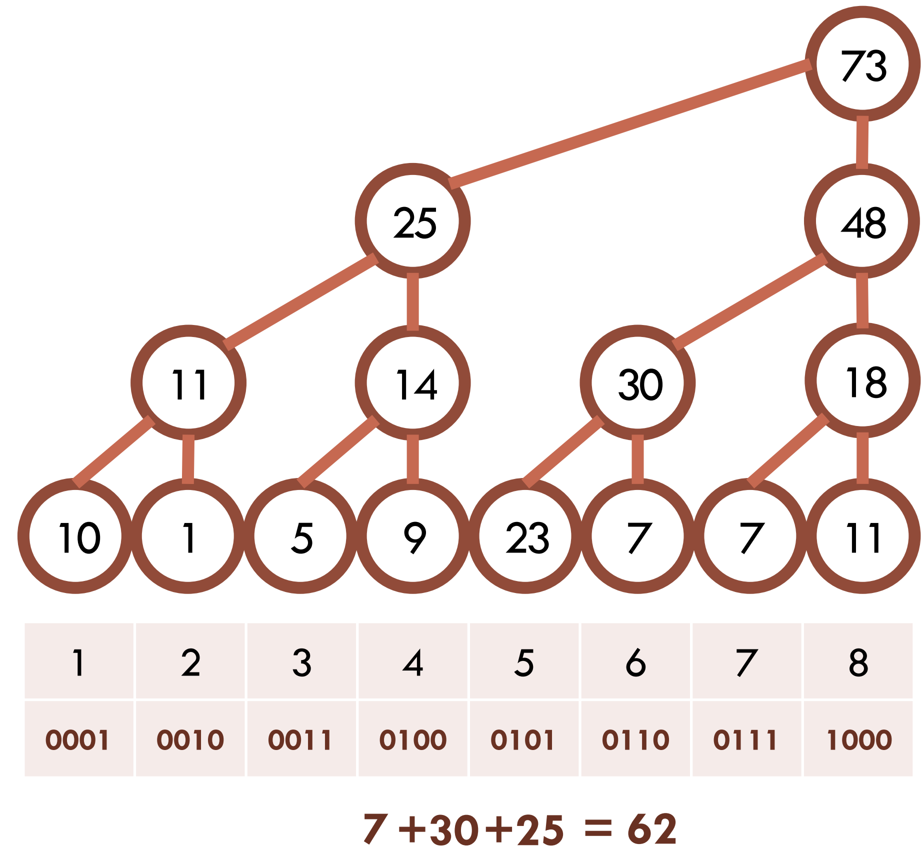
```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4   
5   int FENTREE[MAXN+1];
6   int DATA[MAXN+1];
7   
8   void update(int node, int value) {
9     int delta = value - DATA[node];
10    DATA[node] = value;
11    while (node <= MAXN) {
12      FENTREE[node] += delta;
13      node += node & ~(node-1);
14    }
15  }
```

QUERYING A RANGE

While it is impossible to directly query a specific range in a Fenwick tree, a prefix range can be queried, and due to the reversible nature required, two prefix ranges can be used to get any range.

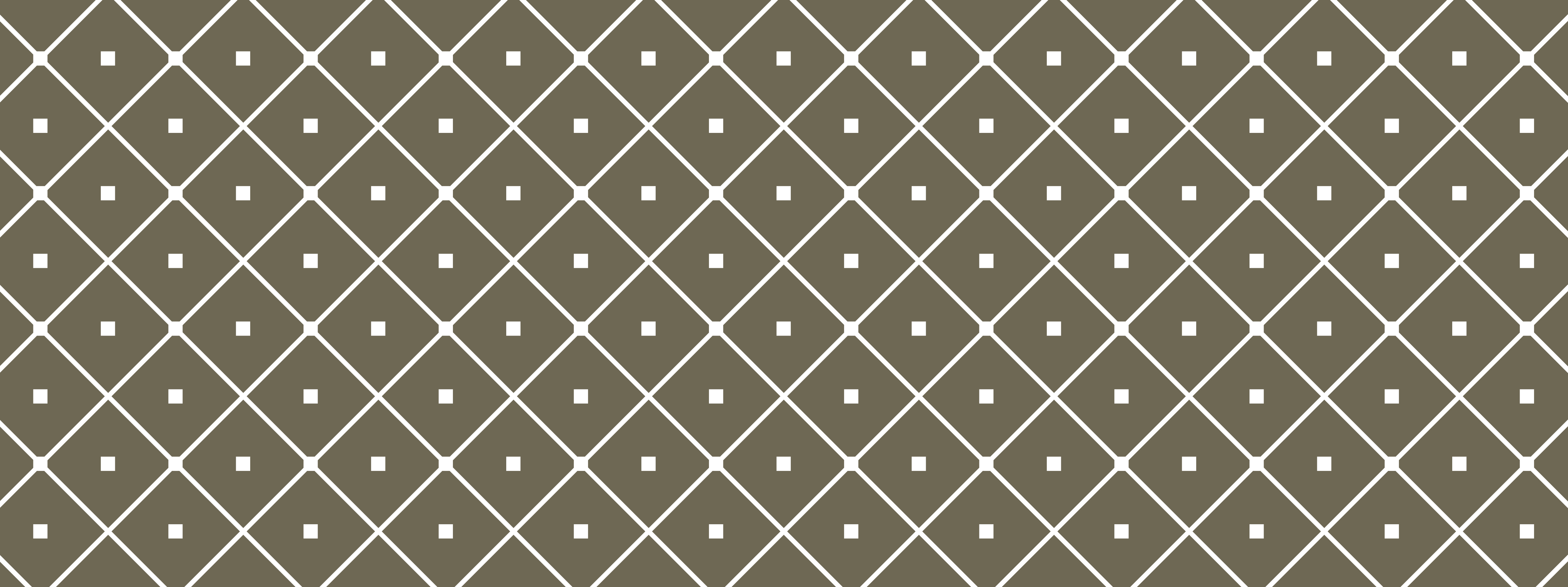
In order to query a prefix range, exactly the opposite is done as when updating. Instead of adding the LSB, it is subtracted. This includes the highest nodes that cover the entire range.

In the example the indices used are as follows: 0111 > 0110 > 0100 > 0000



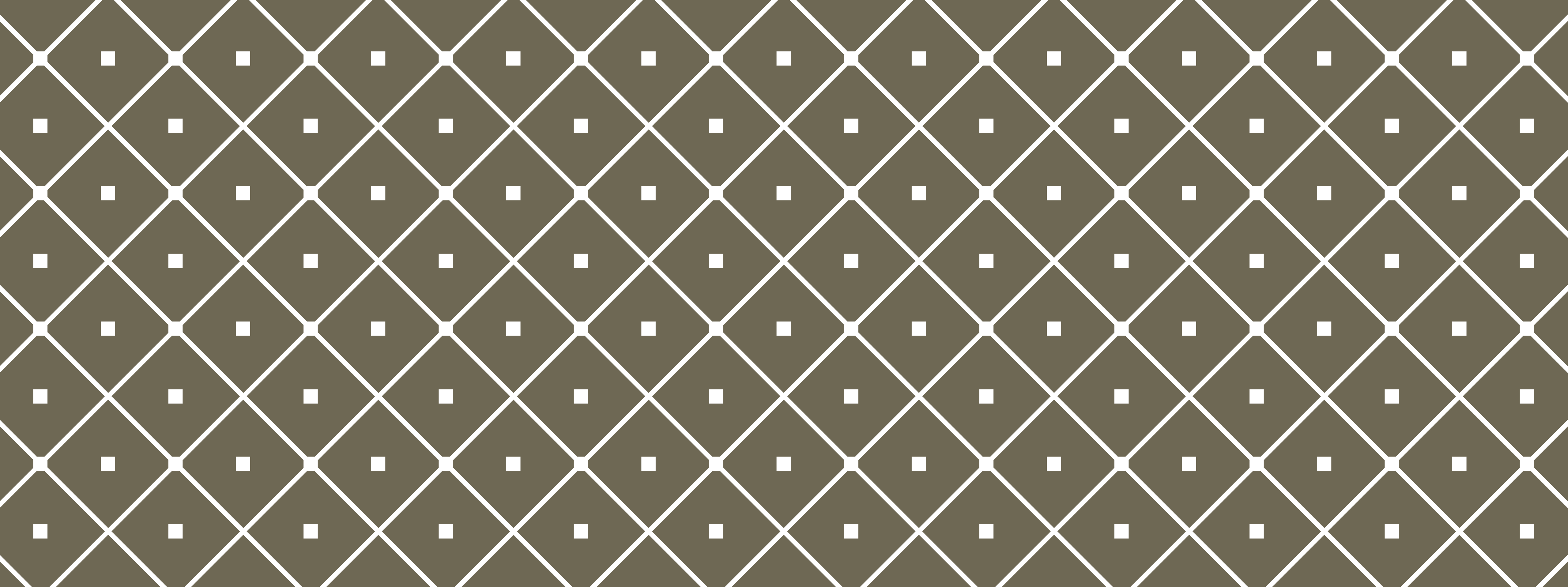
EXAMPLE QUERYING CODE

```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4
5   int FENTREE[MAXN+1];
6   int DATA[MAXN+1];
7
8   int query(int node) {
9     int sum = 0;
10    while (node > 0) {
11      sum += FENTREE[node];
12      node -= node & ~(node-1);
13    }
14    return sum;
15  }
```



WHY BRUCE IS A GOD

The comprehensive analysis of Bruce Merry's godlike nature



LAZY UPDATES

Not suitable for lazy coders, by the way

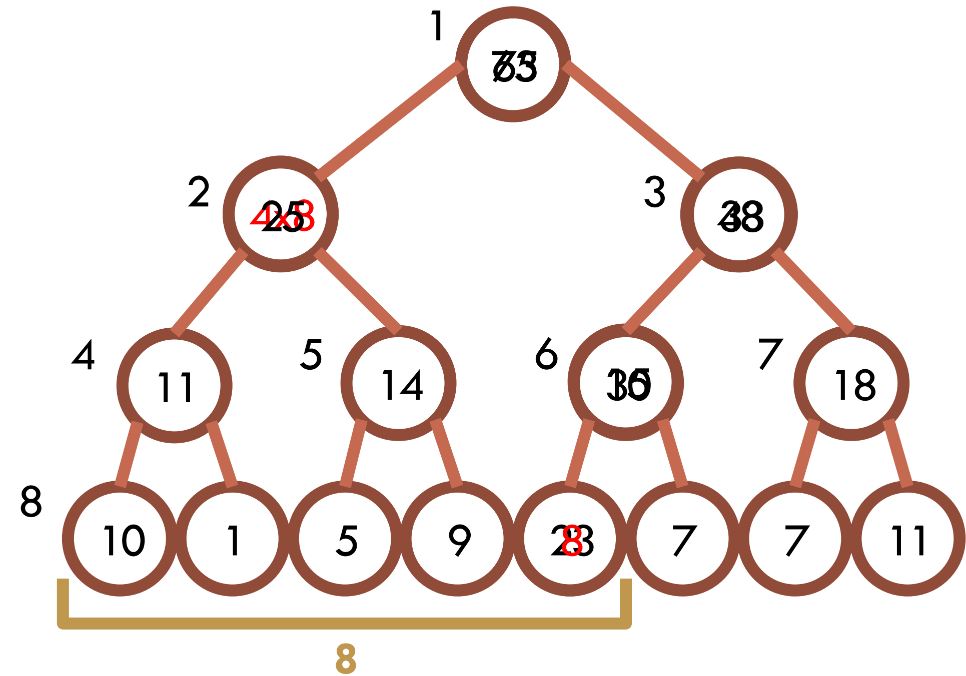
ON RANGE UPDATES

Sometimes it is necessary to not only update a single item at a time, but rather an contiguous range of items. Using a normal segment or Fenwick tree, this would run in O(NlogN). While still relatively fast, this can be improved on.

Lazy updating or lazy propagation is a way to improve the speed of range updates by updating top-down and overriding nodes where the entire range of the node is to be updated, and only updating children when querying or otherwise needed. This can get both range updating and querying done in O(logN) amortised time, but with a small constant penalty to both.

LAZY UPDATING

The idea when updating a range lazily is to start at the root, and if a node is completely contained in the range, override its value and mark it. If not, recursively do the same on both children. If it is entirely out of the range, do nothing.



EXAMPLE LAZY UPDATE CODE

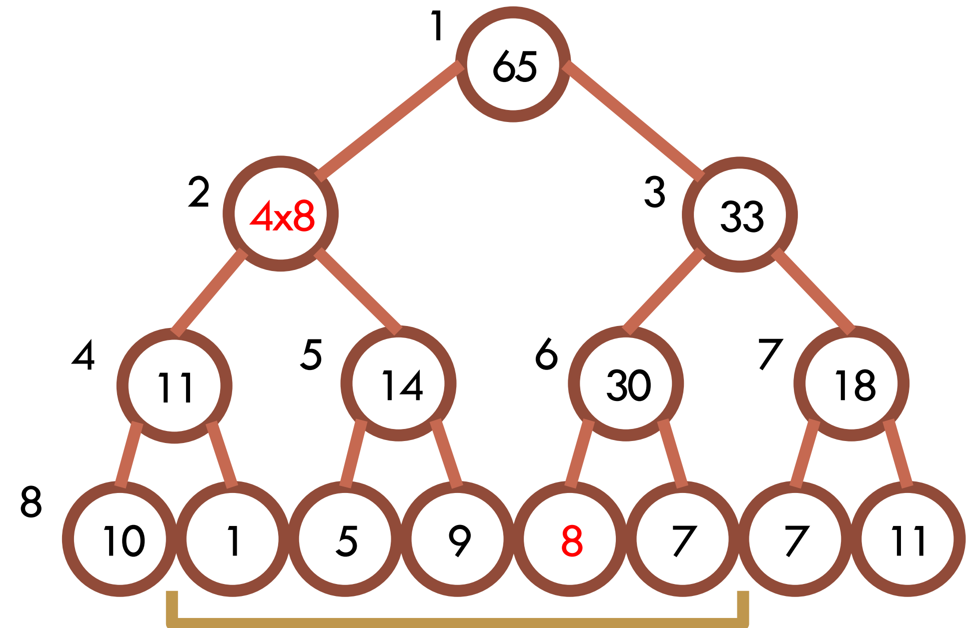
```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4   
5   int SEGTREE[2*MAXN];
6   int LAZY[2*MAXN];
7   
8   void updatepush(int node, int i_left, int i_right, int value) {
9     SEGTREE[node] = (i_right - i_left + 1) * value;
10    if (i_left < i_right) {
11      LAZY[node*2] = value;
12      LAZY[node*2+1] = value;
13    }
14  }
```

EXAMPLE LAZY UPDATE CODE

```
16  void update(int node, int s_left, int s_right, int i_left, int i_right, int value) {
17    if (LAZY[node] != 0) {
18      updatepush(node, i_left, i_right, LAZY[node]);
19      LAZY[node] = 0;
20    }
21    if ((i_left > i_right) || (s_right < i_left) || (i_right < s_left)) return;
22    if ((s_left <= i_left) && (i_right <= s_right)) updatepush(node, i_left, i_right, value);
23    else {
24      int i_mid = (i_left+i_right)/2;
25      update(node*2, s_left, s_right, i_left, i_mid, value);
26      update(node*2 + 1, s_left, s_right, i_mid+1, i_right, value);
27      SEGTREE[node] = SEGTREE[node*2] + SEGTREE[node*2+1];
28    }
29  }
30
31  void update(int left, int right, int value) {
32    update(1, left+MAXN, right+MAXN, MAXN, 2*MAXN-1, value);
33  }
```

QUERYING LAZY UPDATES

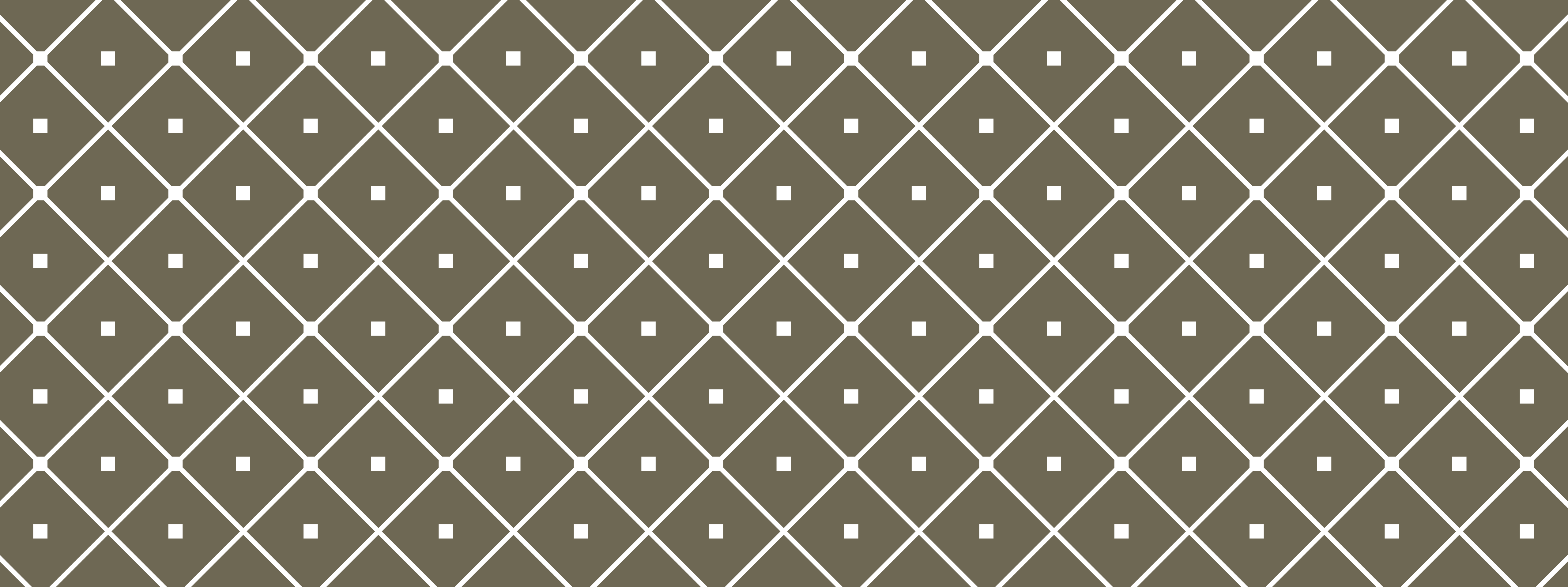
When querying from the root down, it is important to know that a node's value can be used directly if it is contained in the range, but if not, any previous updates applied to it needs to be carried over to its children before it can be used.



EXAMPLE LAZY QUERYING CODE

```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4
5   int SEGTREE[2*MAXN];
6   int LAZY[2*MAXN];

16  int query(int node, int s_left, int s_right, int i_left, int i_right) {
17    if ((i_right < s_left) || (s_right < i_left)) return 0;
18    if (LAZY[node] > 0) {
19      updatepush(node, i_left, i_right, LAZY[node]);
20      LAZY[node] = 0;
21    }
22    if ((s_left <= i_left) && (i_right <= s_right)) return SEGTREE[node];
23    int i_mid = (i_left+i_right)/2;
24    return query(node*2, s_left, s_right, i_left, i_mid) + query(node*2+1, s_left, s_right, i_mid+1, i_right);
25  }
26
27  int query(int left, int right) {
28    return query(1, left+MAXN, right+MAXN, MAXN, 2*MAXN-1);
29  }
```



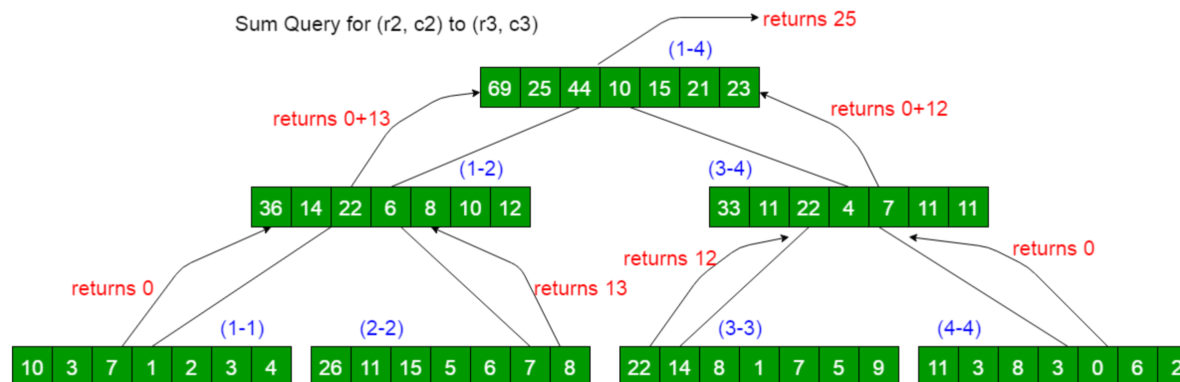
HIGHER DIMENSIONS

Honestly I just included this part because of my beautiful method

THE NEED FOR HIGHER DIMENSIONS

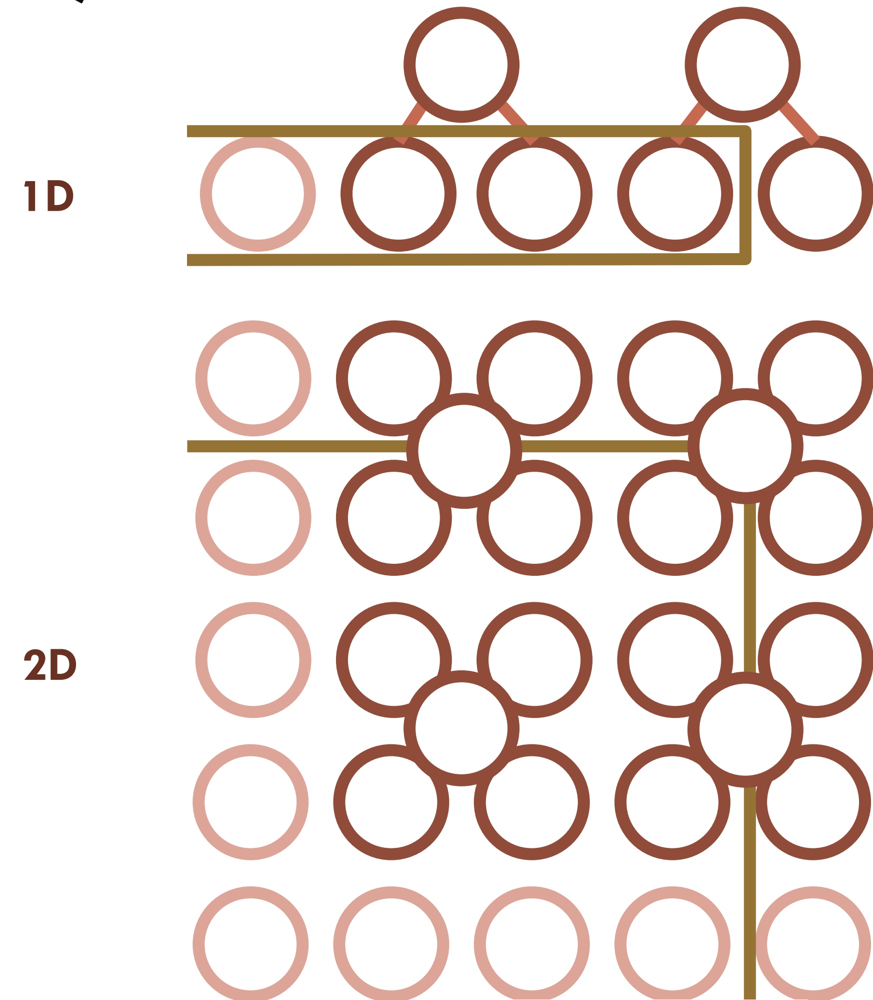
Higher dimension segment trees exist, and it is in fact possible to generalize a lot of the code to make a K-dimensional segment tree. This can be used to get the sum, maximum or minimum of a rectangle, box, or any K-dimensional region in $O((\log N)^K)$.

While there are a few methods that can be used to do this, I have not put the effort in to understand the usual way of having a 'segment tree of segment trees', as seen in this diagram, but will rather teach my own method.



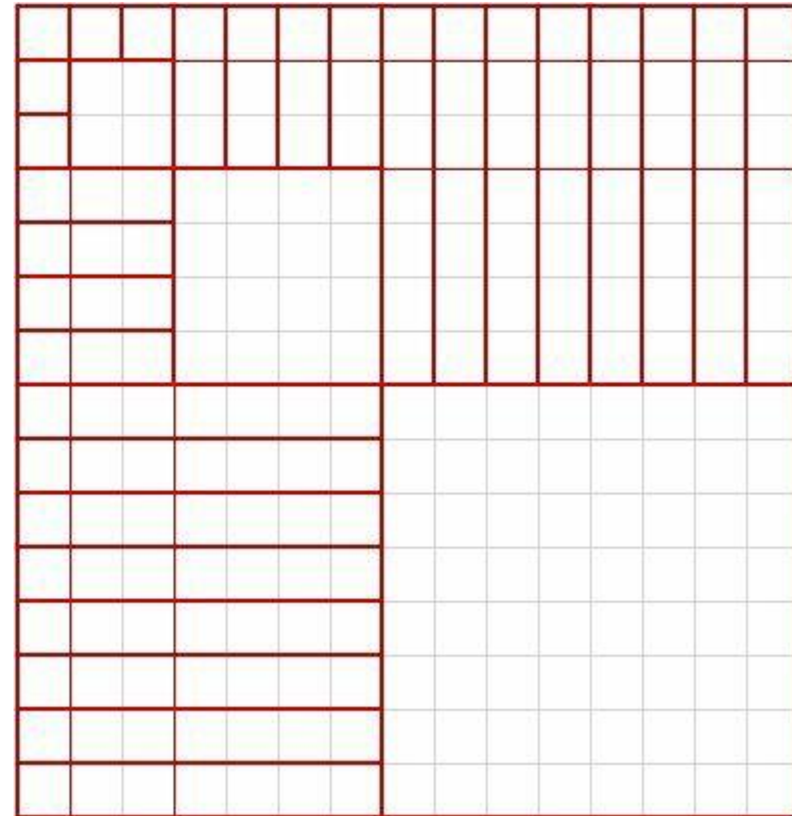
EXAMPLE IMPLEMENTATION (2D)

In order to represent the data properly, we apply a quadtree, or a tree where each node has 4 children in two rows and two columns. This quadtree is queried in much the same way as a segment tree is queried from the bottom up, leading to an intuitive and easy to figure out method. This quadtree, however, requires some extra information about specific ranges in a row or column of the quadtree, which we use segment trees for.



OPTIMAL STRUCTURE (2D)

This quadtree, with accompanying segment trees for each row and column of each level of the quadtree, can conveniently be laid out in a 2Nx2N array, with the layout as shown right. This has the additional advantage of allowing providing easy queries in all trees, as both X and Y need to be halved to move up one level in the quadtree, and only one of them to move one level in the corresponding segment tree.



EXAMPLE UPDATING CODE (2D)

```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4
5   int SEGTREE[2*MAXN][2*MAXN];
6
7   int updatex(int x, int y) {
8     while ((x /= 2) >= 1) SEGTREE[x][y] = SEGTREE[2*x][y] + SEGTREE[2*x+1][y];
9   }
10
11  int updatey(int x, int y) {
12    while ((y /= 2) >= 1) SEGTREE[x][y] = SEGTREE[x][2*y] + SEGTREE[x][2*y+1];
13  }
14
15  void updatexy(int x, int y) {
16    while (((x /= 2) >= 1) && ((y /= 2) >= 1)) {
17      SEGTREE[x][y] = SEGTREE[2*x][2*y] + SEGTREE[2*x][2*y+1] + SEGTREE[2*x+1][2*y] + SEGTREE[2*x+1][2*y+1];
18      updatex(x, y); updatey(x, y);
19    }
20  }
21
22  void update(int x, int y, int value) {
23    SEGTREE[x += MAXN][y += MAXN] = value;
24    updatex(x, y); updatey(x, y);
25    updatexy(x, y);
26  }
```

EXAMPLE QUERYING CODE (2D)

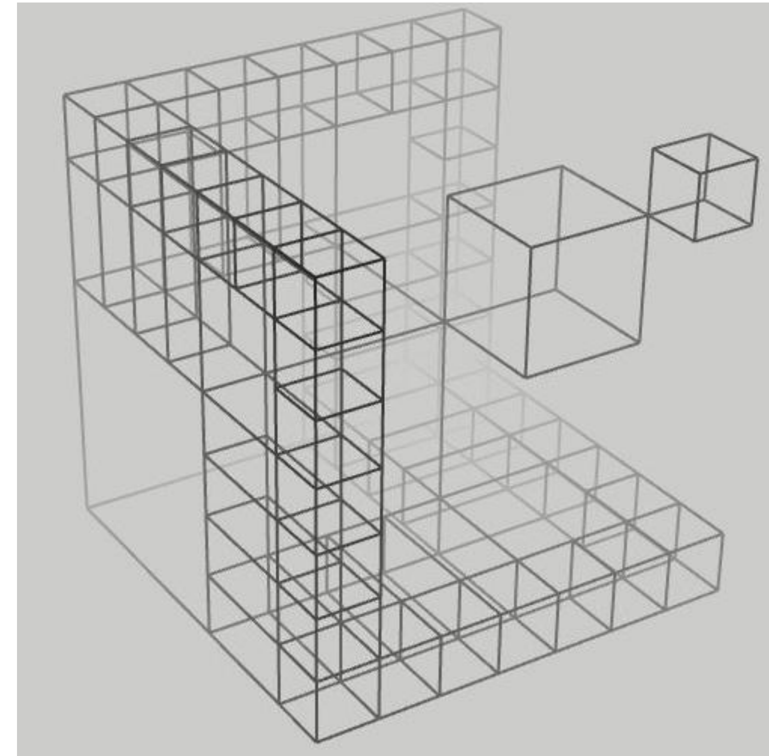
```
1   #include <bits/stdc++.h>
2   #define MAXN 8
3   using namespace std;
4
5   int SEGTREE[2*MAXN][2*MAXN];
6
7   int queryx(int xl, int xr, int y) {
8     int sum = 0;
9     while ((xl <= xr) && (xl >= 1)) {
10      if ( (xl&1)) sum += SEGTREE[xl++][y];
11      if (!(xr&1)) sum += SEGTREE[xr--][y];
12      xl /= 2; xr /= 2;
13    }
14    return sum;
15  }
16
17  int queryy(int x, int yl, int yr) {
18    int sum = 0;
19    while ((yl <= yr) && (yl >= 1)) {
20      if ( (yl&1)) sum += SEGTREE[x][yl++];
21      if (!(yr&1)) sum += SEGTREE[x][yr--];
22      yl /= 2; yr /= 2;
23    }
24    return sum;
25  }
```

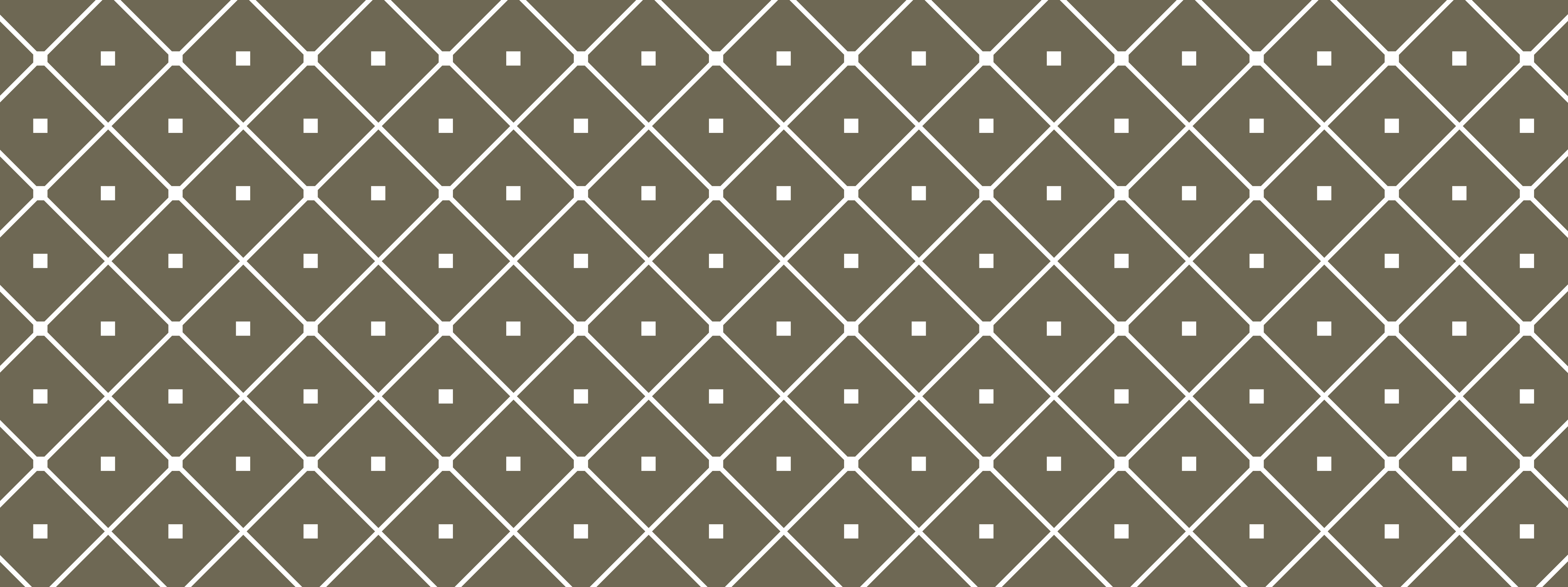
EXAMPLE QUERYING CODE (2D)

```
27  int queryxy(int xl, int xr, int yl, int yr) {
28    int sum = 0;
29    while ((xl <= xr) && (yl <= yr) && (xl >= 1) && (yl >= 1)) {
30      if ( (xl&1)) sum += queryy(xl++, yl, yr);
31      if ( (yl&1)) sum += queryx(xl, xr, yl++);
32      if (!(xr&1)) sum += queryy(xr--, yl, yr);
33      if (!(yr&1)) sum += queryx(xl, xr, yr--);
34      xl /= 2; yl /= 2; xr /= 2; yr /= 2;
35    }
36    return sum;
37  }
38
39  int query(int xl, int yl, int xr, int yr) {
40    xl += MAXN; yl += MAXN; xr += MAXN; yr += MAXN;
41    return queryxy(xl, xr, yl, yr);
42  }
```

THREE-DIMENSIONAL AND FURTHER

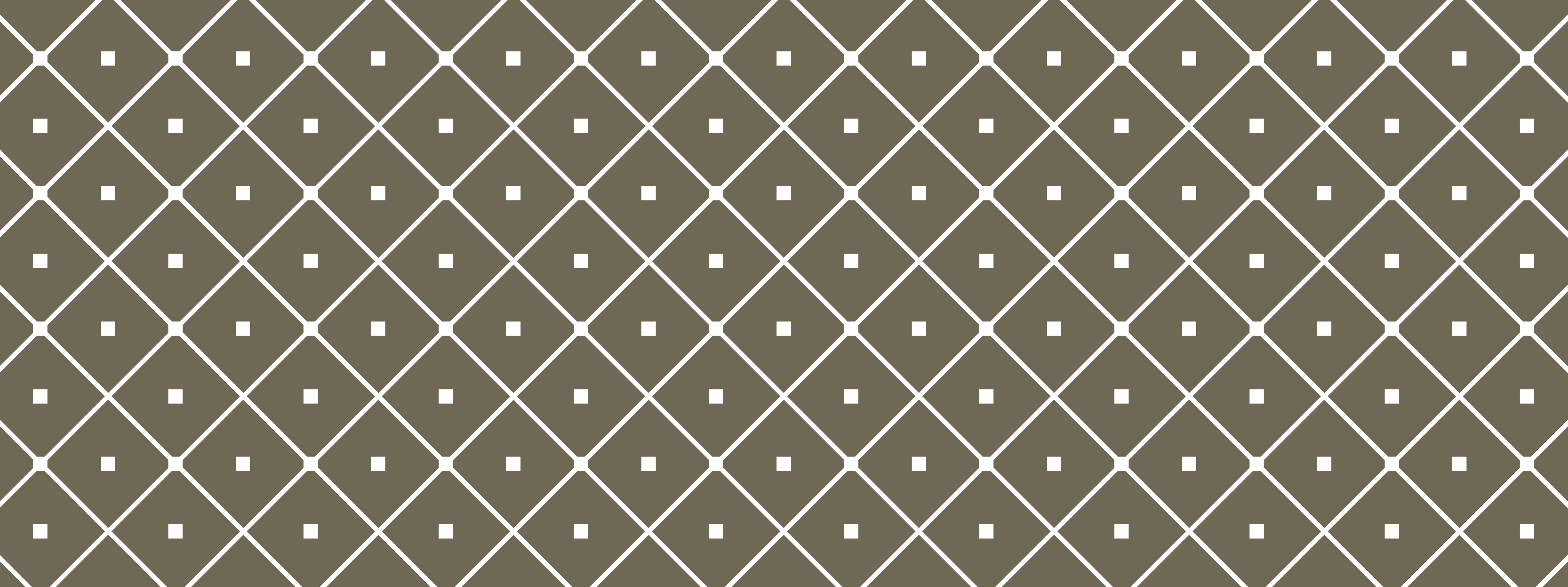
While it is possible to extend the above code to any amount of dimensions, it does become quite cumbersome, even at 3 dimensions. This might be possible to be alleviated by using a top-down recursive strategy instead of a bottom-up iterative one, but the basic idea would remain the same. For interest, here is a simplified model of what the 3D structure would look like.





EXAMPLE PROBLEMS

Because we all love to see how easy something is in retrospect



QUESTIONS

I was really confused myself when first introduced to segtrees