FENWICK TREES

ADVANCED ALGORITHMS

MARK FLORYAN

MATERIAL FROM:
HTTPS://CP-ALGORITHMS.COM/DATA_STRUCTURES/FENWICK.HTML

ADVANCED ARRAY STRUCTURES

In this deck we will look at:

- ***Fenwick Trees***

FENWICK TREES

MOTIVATION

Goal: Given a list of integers $A = \{a_1, a_2, \dots, a_n\}$ and a function that operates on continuous ranges in A, called $f(l, r)$ where $l, r \in Z; l \leq r$

Support the following operations:

1. Calculate (for any l,r) the value of $f(l, r) = f(a_l, \dots, a_r)$ in $O(\log n)$ time
2. Update the value of an element of A in $O(\log n)$ time.
3. Use no more than $O(n)$ memory (so no more than list A itself)

***Note: This will work for any function f(), but sum(l, r) is a common first one to start with*

WHY WOULD WE WANT THIS?

$$A = \{5, 10, 1, 11, 29, 3, 2, 209, 85, 6, 9, 11\}$$

Suppose: That our stream of integers are stock prices over time, or sensor data across many time points, or sales per day (etc.).

Perhaps our company needs to be able to pull f() over any arbitrary range in A thousands of time per day

Suppose: That f() is a function we care about over any range. Max and min (for stock prices), average (for sensor data), or sum (for sales)

NAÏVE APPROACH 1

$$A = \{5, 10, 1, 11, 29, 3, 2, 209, 85, 6, 9, 11\}$$

Store: A as normal with no alterations

Compute ***sum*** by simply traversing the array. This is O(n) time. TOO SLOW!

Update value by simply updating it directly. This is O(1) time. Good!!

Assume: For now that f() is just the sum function:

$$f(l, r) = \sum_{i=l}^{r} a_i$$

NAÏVE APPROACH 2

$A = \{5, 10, 1, 11, 29, 3, 2, 209, 85, 6, 9, 11\}$

$T = \{5,15,16,27, 56, 59,61,270,355,361,\ldots\}$

Store: sum from index 1 to i in cell i.

How would we compute ***sum*** from l to r ?? What is the time complexity?

How do we update a value?

Assume: For now that f() is just the sum function:

$$f(l,r) = \sum_{i=l}^{r} a_i$$

GENERALIZING THE NAÏVE APPROACH

IDEA: Let's balance the two approaches. Store prefixes but not always the range from 0 to i such that we can get fast runtimes for computing f(l,r) AND fast runtimes for updating a value.

Assume: Update our new array T to store partial prefixes:

$$T_i = \sum_{j=g(i)}^{i} a_j$$

$0 \leq g(i) \leq i$

**Note:

For Naïve Approach 1: $\forall_i \; g(i) = i$

For Naïve Approach 2: $\forall_i \; g(i) = 0$

What should g(i) be??

GENERALIZING THE NAÏVE APPROACH

Implementation: Regardless of what we choose for g(i), the following pseudo-code will work (here we are assuming f() is the sum function for simplicity).

Notice: sum() calculates the sum from index 0 to r. So, if we want to calculate the sum of a range we still use:

$$f(l,r) = sum(r) - sum(l-1)$$

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

def increase(int i, int delta):
    for all j with g(j) <= i <= j:
        t[j] += delta
```

Not obvious to see how to make this one fast? We will see how in a moment.

DEFINING FUNCTION G

$$g(i) = ??$$

Requirements:

- Should be fast ($O(1)$) to compute
- For any i, should take $O(\log i)$ iterations to reach zero
- Needs to be ”reversible” so we can efficiently find, for some i, the set satisfying $\forall_j\ g(j) \leq i \leq j$

DEFINING FUNCTION G

Bitwise ***AND***

$$g(i) = i \,\&\, (i+1)$$

Strengths:

- Extremely fast to compute
- For any i, takes $O(\log i)$ iterations to reach zero
- Is easily reversible (We will see how in a moment)

Intuitively:

- Turns all trailing 1's in binary representation to 0s

Examples:

$$g(11) = g(1011_2) = 1000_2 = 8$$
$$g(12) = g(1100_2) = 1100_2 = 12$$
$$g(13) = g(1101_2) = 1100_2 = 12$$

EXAMPLE ITERATION OF SUM(14)

$$g(i) = i \,\&\, (i+1)$$

$$T = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}\}$$

DONE!

add t_7 to res
$g(7)$
$= g(0111_2)$
$= 0000_2 = 0$

add t_{11} to res
$g(11)$
$= g(1011_2)$
$= 1000_2 = \mathbf{8}$

add t_{13} to res
$g(13)$
$= g(1101_2)$
$= 1100_2 = \mathbf{12}$

add t_{14} to res
$g(14)$
$= g(1110_2)$
$= 1110_2 = \mathbf{14}$

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

def increase(int i, int delta):
    for all j with g(j) <= i <= j:
        t[j] += delta
```

*** Notice: The number of trailing 1's increases by 1 each time, this is why we will terminate in $\Theta(\log n)$*

CONCRETE EXAMPLE!

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$T = \{ \quad \}$$

g() values for reference:

$g(0) = g(0000_2) = 0000_2 = 0$
$g(1) = g(0001_2) = 0000_2 = 0$
$g(2) = g(0010_2) = 0010_2 = 2$
$g(3) = g(0011_2) = 0000_2 = 0$
$g(4) = g(0100_2) = 0100_2 = 4$
$g(5) = g(0101_2) = 0100_2 = 4$
$g(6) = g(0110_2) = 0110_2 = 6$
$g(7) = g(0111_2) = 0000_2 = 0$
$g(8) = g(1000_2) = 1000_2 = 8$
$g(9) = g(1001_2) = 1000_2 = 8$
$g(10) = g(1010_2) = 1010_2 = 10$
$g(11) = g(1011_2) = 1000_2 = 8$
$g(12) = g(1100_2) = 1100_2 = 12$

Now, let's do:

sum(3,8) *sum(1,10)* *sum(5,6)*

UPDATE: REVERSING G()

$$h(i) = \forall_j g(j) \leq i \leq j$$

Requirements:

- Should be fast ($O(1)$) to compute
- For any i, should take $O(\log i)$ iterations to loop over set
- Needs to correctly cover all cells in array that need to be updated

UPDATE: REVERSING G()

Bitwise ***OR***

$$h(j) = j \mid (j + 1)$$

Strengths / Notes:

- Sets the lowest 0 bit to a 1
- Intuitively, this reverses g() because g() removes ALL trailing 1's and turns them to 0's. h() puts those 1's back (one invocation of h() per digit)
- Is VERY fast to compute

Examples:

- $h(10) = h(1010_2) = 1011_2 = 11$
- $h(11) = h(1011_2) = 1111_2 = 15$
- $h(15) = h(1111_2) = 11111_2 = 31$

EXAMPLE ITERATION OF UPDATE(4, 2)

$h(j) = j \mid (j+1)$

** *Example: t9 does not need to be updated because:*

$g(9) = g(1001_2) = 1000_2 = 8 > 4$

$$T = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}\}$$

add 2 to t_4

$h(4) = h(0100_2) = 0101_2 = 5$

add 2 to t_5

$h(5) = h(0101_2) = 0111_2 = 7$

add 2 to t_7

$h(7) = h(0111_2) = 1111_2 = 15$

15 is out of bounds.

DONE!

```
def sum(int r):
    res = 0
    while (r >= 0):
        res += t[r]
        r = g(r) - 1
    return res

def increase(int i, int delta):
    for all j with g(j) <= i <= j:
        t[j] += delta
```

** ***Notice: We change one digit from 0 to 1 on each iteration, so all will be changed in $\Theta(\log n)$ time.***

CONCRETE EXAMPLE!

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$T = \{1, 3, 3, 10, 5, 11, 7, 36, 9, 19, 11, 42\}$$

g() values for reference:

$g(0) = g(0000_2) = 0000_2 = 0$
$g(1) = g(0001_2) = 0000_2 = 0$
$g(2) = g(0010_2) = 0010_2 = 2$
$g(3) = g(0011_2) = 0000_2 = 0$
$g(4) = g(0100_2) = 0100_2 = 4$
$g(5) = g(0101_2) = 0100_2 = 4$
$g(6) = g(0110_2) = 0110_2 = 6$
$g(7) = g(0111_2) = 0000_2 = 0$
$g(8) = g(1000_2) = 1000_2 = 8$
$g(9) = g(1001_2) = 1000_2 = 8$
$g(10) = g(1010_2) = 1010_2 = 10$
$g(11) = g(1011_2) = 1000_2 = 8$
$g(12) = g(1100_2) = 1100_2 = 12$

Now, let's do:

update(3,5) *update(0,10)*

FINAL UPDATE CODE

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

FENWICK TREE IMPLEMENTATION

Implementation: Full implementation for f() = sum() in one dimension.

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

What is the runtime of this constructor? Can we make it faster?

FENWICK TREE LINEAR TIME CONSTRUCTOR

FENWICK TREE IMPLEMENTATION

Implementation: Full implementation for f() = sum() in one dimension.

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

This is $\Theta(n \log n)$

Can we make this $\Theta(n)$

FENWICK TREE IMPLEMENTATION

Key Observation

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> const &a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

Key Observation:

Each element a[i] contributes to bit[i] and everything bit[i | (i+1)]. So, if we push the change up one "level", we can then push it up again eventually when we each index i | (i+1)

FENWICK TREE FAST CONSTRUCTION

Recall that $i \mid (i+1)$ is the function $h(i)$

```
FenwickTree(vector<int> const &a) : FenwickTree(a.size()){
    for (int i = 0; i < n; i++) {
        bit[i] += a[i];
        int r = i | (i + 1);
        if (r < n) bit[r] += bit[i];
    }
}
```

$$T = \{t_0, t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}\}$$

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$$

When loop at i=8:

$r = i|(i+1) = 8|9 = 9$

$bit[9] \mathrel{+}= bit[8] = 36$

Consider t_9:

$9_{10} = 1001_2$

$t_9 = 36$ *//sum from 0 to 8*

When loop at i=9:

$bit[9] \mathrel{+}= a[9] = 36 + 9 = 45$

FENWICK TREE FOR MIN FUNCTION

FENWICK TREE IMPLEMENTATION

Implementation: Partial implementation with f() = min()

```
struct FenwickTreeMin {
    vector<int> bit;
    int n;
    const int INF = (int)1e9;

    FenwickTreeMin(int n) {
        this->n = n;
        bit.assign(n, INF);
    }

    FenwickTreeMin(vector<int> a) : FenwickTreeMin(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            update(i, a[i]);
    }

    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }

    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] = min(bit[idx], val);
    }
};
```

getmin() returns min from index 0 to r

Can you adapt to return min from a given left and right index? Why or why not?

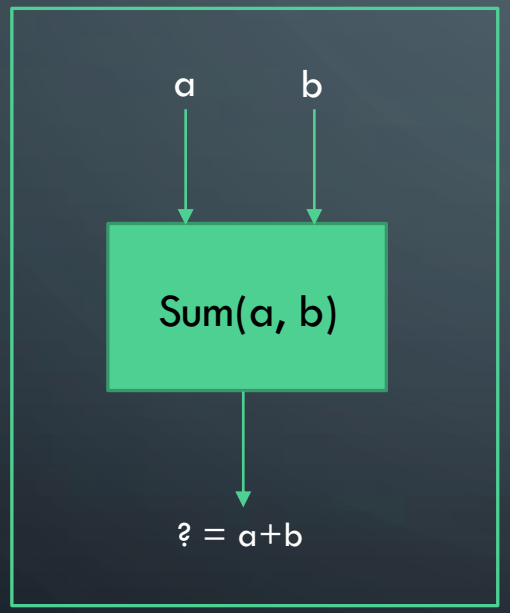
What about update(int idx, int val), does this method have any restrictions?

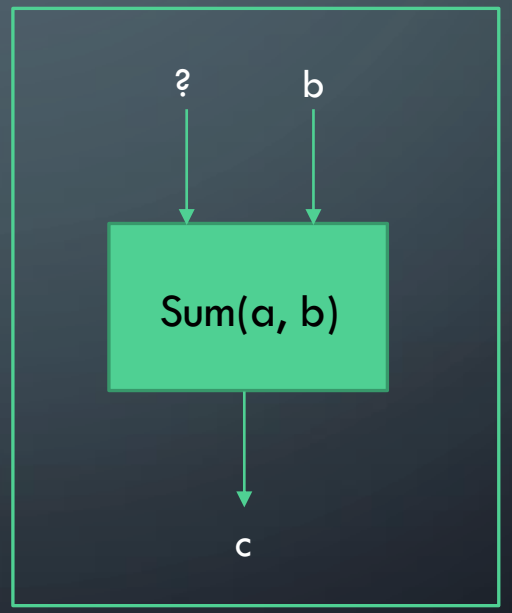
FUNCTION MUST BE INVERTIBLE

Invertible Functions

Fenwick tree supports arbitrary range and updates if the function of interest (e.g., sum) is invertible. An invertible function is one in which you can determine the inputs given the output (and in our case, all but one of the outputs)

"Normal", "Forward" use of a function like sum()



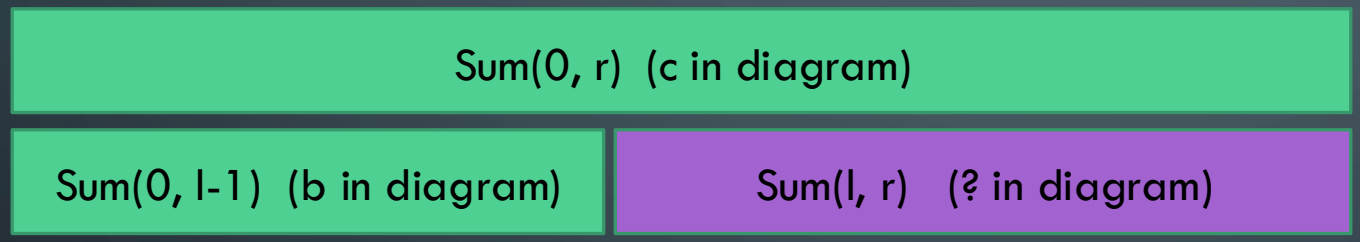


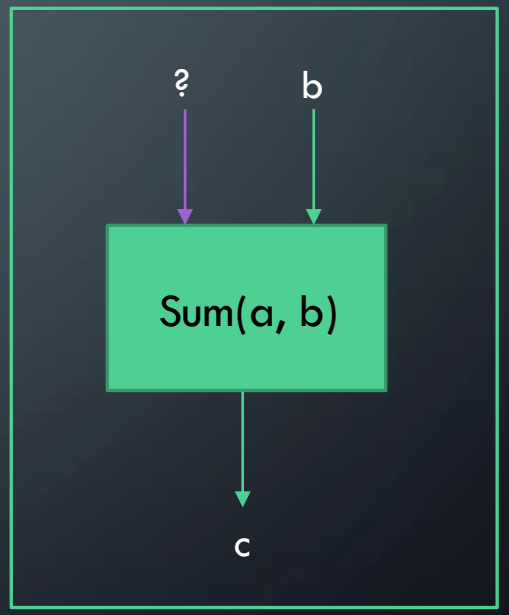
Given one input (b) and the output (c), can you compute the second original input?

FUNCTION MUST BE INVERTIBLE

Invertible Functions

Fenwick tree supports arbitrary range and updates if the function of interest (e.g., sum) is invertible. An invertible function is one in which you can determine the inputs given the output (and in our case, all but one of the outputs)



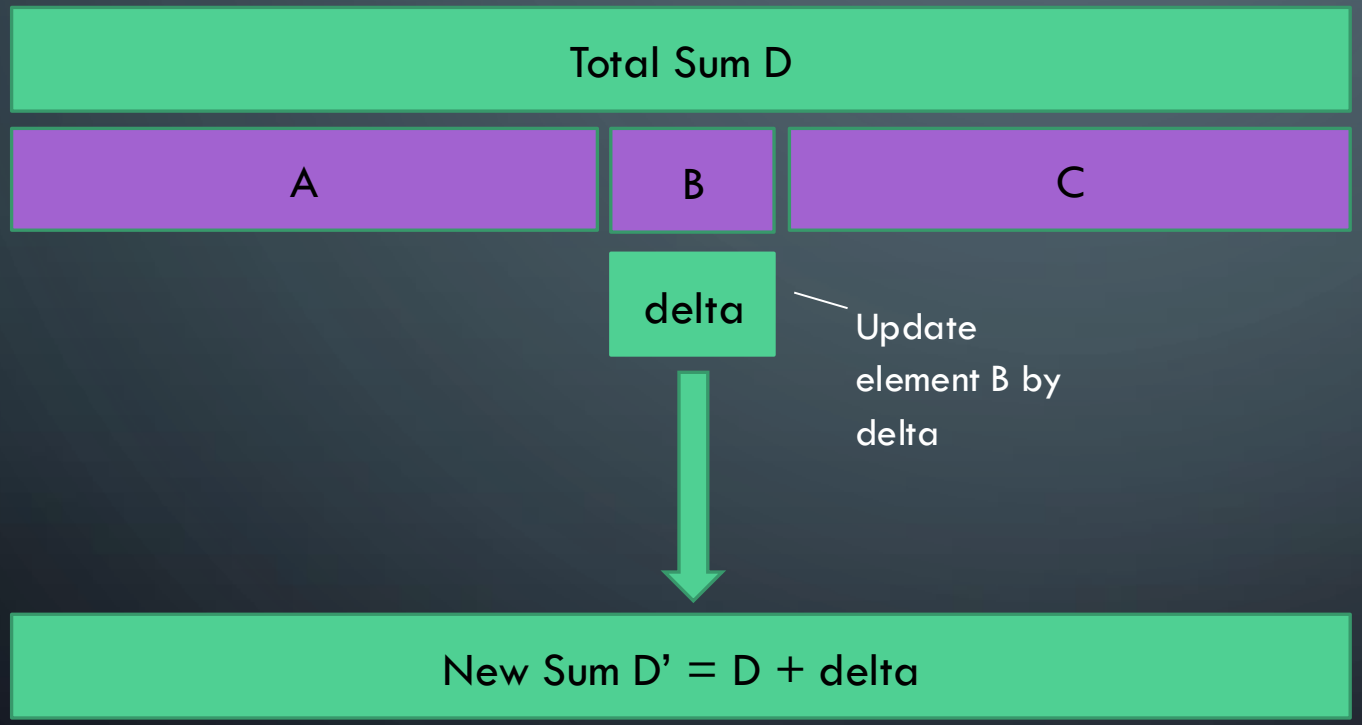


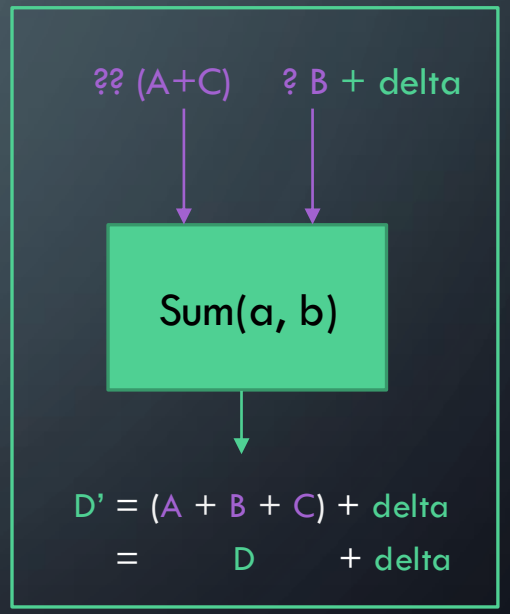
For Sum(), this is fine, but consider min(a,b) and you'll notice that it cannot be inverted unless c is not equal to b

FUNCTION MUST BE INVERTIBLE

Invertible Functions

A similar relationship occurs with updates





ONE MORE OPERATION:
SUMS THAT EXCEED GIVEN THRESHOLD

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

Example:

Consider thresholdSearch(14)

This call should return index 3 because:

$Sum[0,3] = 15 \geq 14$ and 3 is the smallest given this constraint.

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2

T =	3	10	1	15	5	9	3	28	1	3

Here is the T array (Fenwick Tree). How might we search this efficiently to implement thresholdSearch(int t)?

Option 1: Binary Search

```
thresholdSearch(int t):
    l = 0; r=T.size()-1
    while(l<r):
        mid = Floor(T.size() / 2)
        val = Sum(mid)              //fenwick tree sum query
        if(val < threshold) then l=mid+1
        else r = mid

    return l                //l should equal r here
```

What is the runtime of this?

log(n) binary search with a log(n) call to Sum() $= \Theta(\log^2 n)$

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

Notice that some indices store full prefixes (g() function takes them to 0:

0 $g(0) = g(0000_2) = 0000_2 = 0$

1 $g(1) = g(0001_2) = 0000_2 = 0$

3 $g(3) = g(0011_2) = 0000_2 = 0$

7 $g(7) = g(0111_2) = 0000_2 = 0$

This is great for a first check because we quickly check an entire prefix sum and know if the answer index is to the right or to the left (feels like binary search) without calling the sum() function

First index we check should be highest value such that g() = 0:

$$\boldsymbol{idx = (1 \ll \lfloor \log_2(T.size) \rfloor) - 1}$$

e.g.,: $(1 \ll \lfloor \log_2 10 \rfloor) - 1$
$= (1 \ll 3) - 1$
$= 1000_2 - 1$
$= 0111_2$

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

Some variables we need / will use:

sum = 0	*Will update as we go and build off this*
idx $= 000_2$	*Search values i such that g(i) = idx*
mask $= 111_2$	*Initial value as per previous slide*

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum $= 0$
idx $= 000_2$
mask $= 111_2$

Consider thresholdSearch(22): Answer should be index 5

1) Check idx+mask $= 000_2 + 111_2 = 111_2 = 7_{10}$ *sum+T[7]* $= 28 > 22$ ***Go Left (mask >> 1)***

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum = 0
idx $= 000_2$
mask $= \mathbf{11_2}$

Consider thresholdSearch(22): Answer should be index 5

1) Check idx+mask $= 000_2 + 111_2 = 111_2 = 7_{10}$ — *sum+T[7]* $= 28 > 22$ — *Go Left (mask >> 1)*

2) Check idx+mask $= 000_2 + 11_2 = 11_2 = 3_{10}$ — *sum+T[3] = 15 < 22* — ***Go Right*** *(idx+=mask+1; mask>>1; sum+=T[3])*

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum = 15
idx = 100_2
mask = 1_2

Consider thresholdSearch(22): Answer should be index 5

1) Check idx+mask $= 000_2 + 111_2 = 111_2 = 7_{10}$ *sum+T[7]* $= 28 > 22$ *Go Left (mask >> 1)*

2) Check idx+mask $= 000_2 + 11_2 = 11_2 = 3_{10}$ *sum+T[3]* $= 15 < 22$ *Go Right (idx+=mask+1; mask>>1; sum+=T[3])*

Now we know answer index is between 3 and 7, we also know prefix sum to index 3 is 15!
IDEA: Search across indices between 3 and 7 whose g() function maps to $3 + 1 = 4 = 100_2$

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum = 15
idx = 100_2
mask = 1_2

Consider thresholdSearch(22): Answer should be index 5

1) Check idx+mask $= 000_2 + 111_2 = 111_2 = 7_{10}$ — *sum+T[7] = 28 > 22* — *Go Left (mask >> 1)*

2) Check idx+mask $= 000_2 + 11_2 = 11_2 = 3_{10}$ — *sum+T[3] = 15 < 22* — *Go Right (idx+=mask+1; mask>>1; sum+=T[3])*

3) Check idx+mask $= 100_2 + 001_2 = 101_2 = 5_{10}$ — sum+T[5] = 15+9 = 24 > 22 — **Go Left (mask >> 1)**

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum = 15
idx = 100_2
mask = 0_2

Consider thresholdSearch(22): Answer should be index 5

1) Check idx+mask $= 000_2 + 111_2 = 111_2 = 7_{10}$	*sum+T[7]* $= 28 > 22$	*Go Left (mask >> 1)*
2) Check idx+mask $= 000_2 + 11_2 = 11_2 = 3_{10}$	*sum+T[3] = 15 < 22*	*Go Right (idx+=(mask+1); mask>>1; sum+=T[3])*
3) Check idx+mask $= 100_2 + 001_2 = 101_2 = 5_{10}$	sum+T[5] = 15+9 = 24 > 22	Go Left (mask >> 1)
4) Check idx+mask $= 100_2 + 000_2 = 100_2 = 4_{10}$	sum+T[4] = 15+5 = 20 < 22	**Go Right (idx+=(mask+1); mask>>1; Sum+=T[4])**

THRESHOLD SEARCH

Consider the operation:

thresholdSearch(int t): Given a threshold value t, return the smallest index i such that $Sum[0, i] \geq t$

A =	3	7	1	4	5	4	3	1	1	2
T =	3	10	1	15	5	9	3	28	1	3

sum = 15
idx = 101_2
mask = 0_2

Some notes to consider for this:

1. ***idx is your answer at the end, but notice that there is no answer (no index works) if you never "go left". Make sure to keep track of that***
2. ***Think through your termination conditions to ensure you don't have out of bounds or off by one errors. Algorithm terminates when the mask hits 0 (make sure to check with mask=0 once before terminating)***
3. ***Runtime is clearly $\Theta(\log n)$ because mask reduces by one bit each time and we terminate after it hits 0***

CONCLUSION

CONCLUSIONS

Strengths:

- Works great for simple f(), but complicated for some functions (min over l,r is not trivial)
- Logarithmic time complexity for both queries and updates
- Once understood conceptually, very easy implementation
- Easy to extend to 2D arrays (see reading for details)

Weaknesses:

- Some functions aren't natural to implement because you can't build off smaller solutions (e.g., min…which section of the array is the min actually in!?)

Other cool things:

- There is a variation that indexes from 1 in the array (see reading if you want to use this version)
- 2D summation Fenwick tree is not too difficult to do (check out reading for details)
- Can also support range updates and index queries with some fancy tricks…