Segment Trees, Rolling Hashes, and Bloom Filters

Finn Lidbetter

April 2017

Overview

1. Segment Trees

2. Rolling Hashes

3. Bloom Filters

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:
  - Sums

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:
  - Sums
  - Minimum and/or Maximum

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:
  - Sums
  - Minimum and/or Maximum
  - GCD, LCM

Segment Tree Overview

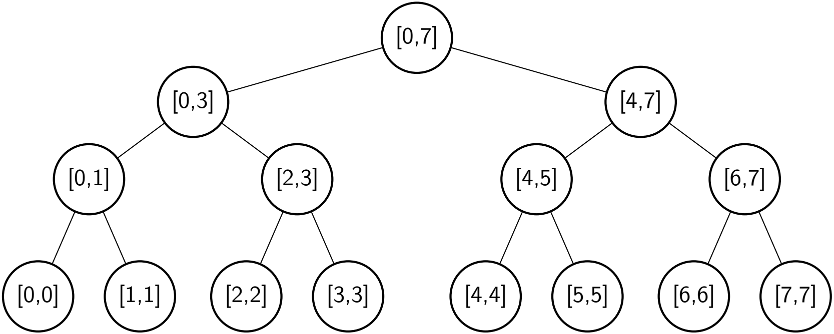
- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:
  - Sums
  - Minimum and/or Maximum
  - GCD, LCM
- Example updates:

Segment Tree Overview

- Powerful data structure for performing queries and updates on arrays
- AKA: Interval Tree, Binary Indexed Tree, Tournament Tree
- Very flexible: supports range updates and range queries
- Query function must be associative: $a + (b + c) = (a + b) + c$
- Update function must be associative and commutative: $a + b = b + a$
- Example queries:
  - Sums
  - Minimum and/or Maximum
  - GCD, LCM
- Example updates:
  - Increment/decrement all values in an interval

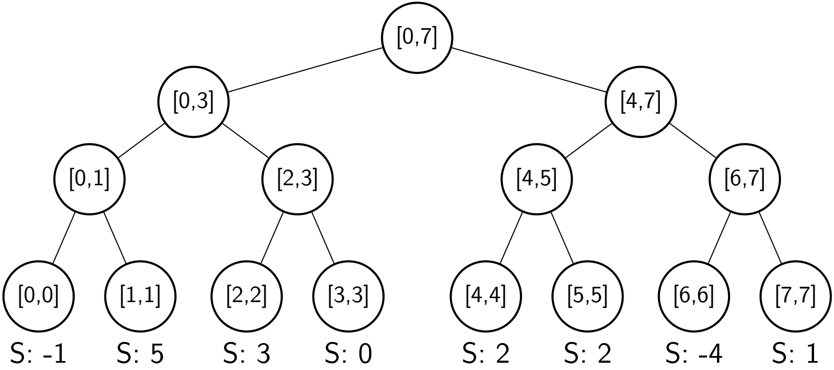
The Structure

- Binary tree where each node stores information about the interval spanned by its subtree.
- E.g. Store sums of elements in sub-arrays of [-1, 5, 3, 0, 2, 2, -4, 1]



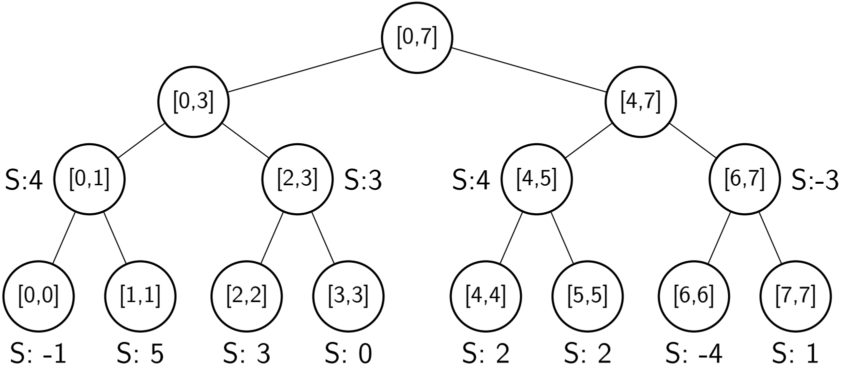
The Structure

- Binary tree where each node stores information about the interval spanned by its subtree.
- E.g. Store sums of elements in sub-arrays of [-1, 5, 3, 0, 2, 2, -4, 1]



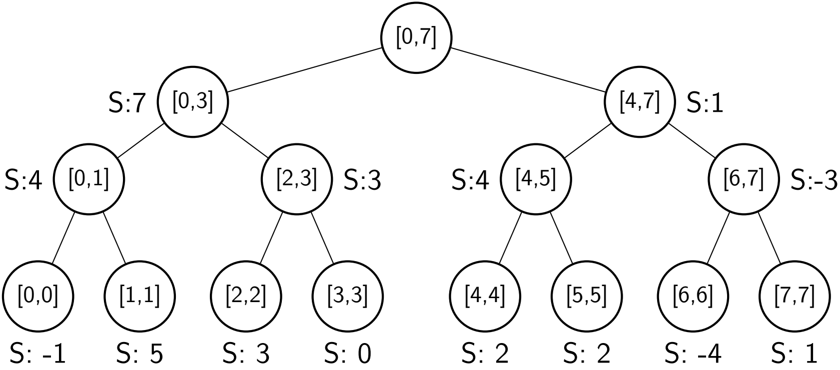
The Structure

- Binary tree where each node stores information about the interval spanned by its subtree.
- E.g. Store sums of elements in sub-arrays of [-1, 5, 3, 0, 2, 2, -4, 1]



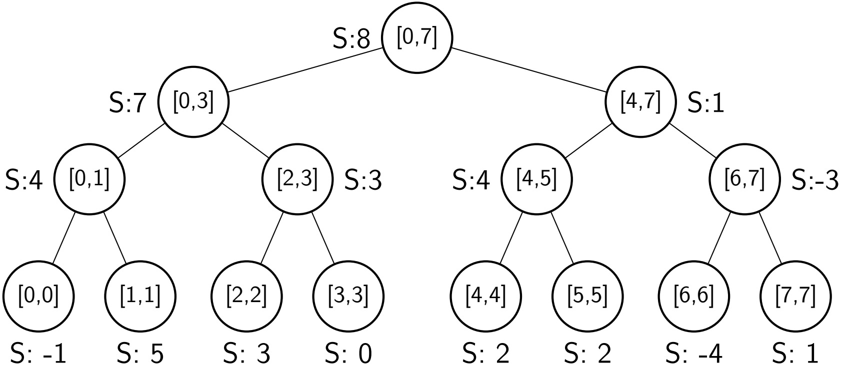
The Structure

- Binary tree where each node stores information about the interval spanned by its subtree.
- E.g. Store sums of elements in sub-arrays of [-1, 5, 3, 0, 2, 2, -4, 1]



The Structure

- Binary tree where each node stores information about the interval spanned by its subtree.
- E.g. Store sums of elements in sub-arrays of [-1, 5, 3, 0, 2, 2, -4, 1]

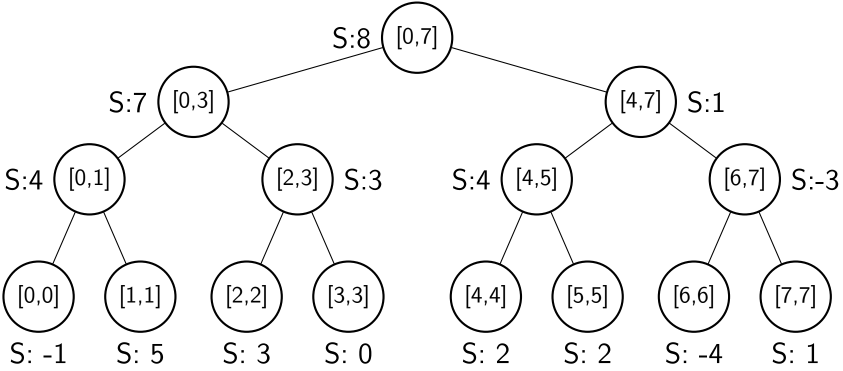


Querying

- Query is on some interval [a,b]
- 4 cases:
  - Node interval is a subset of the Query interval
    Return value stored in node
  - Query interval is a proper subset of Node interval
    Recurse on children
  - Query interval and Node interval are disjoint
    Return "0" (return value depends on query operation)
  - Query interval and Node interval are partially overlapping
    Recurse on children

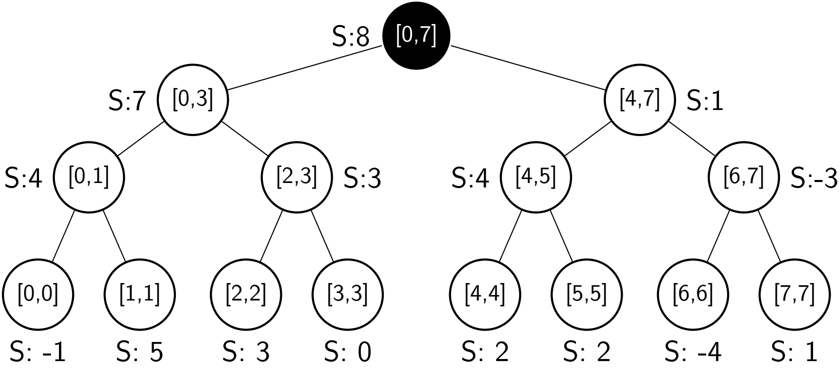
Querying Example

- Query the sum of the interval [1,3]



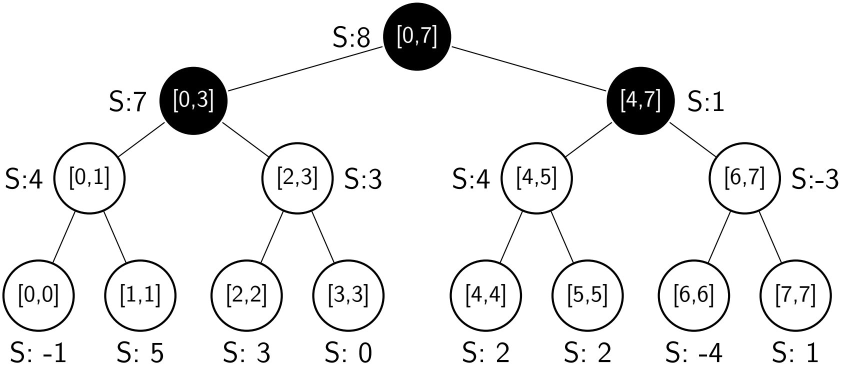
Querying Example

- Query the sum of the interval [1,3]



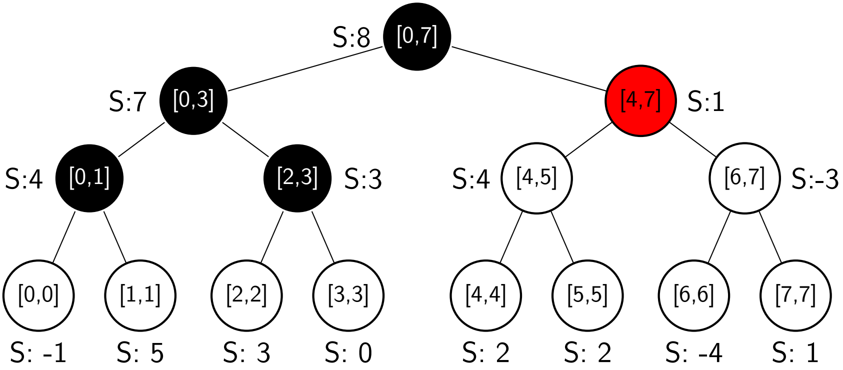
Querying Example

- Query the sum of the interval [1,3]



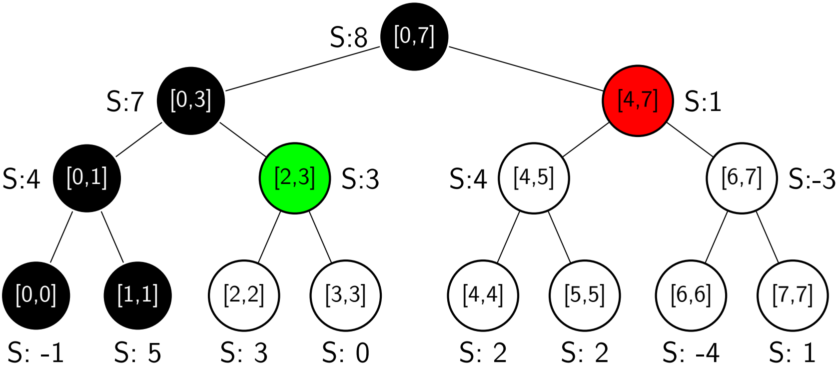
Querying Example

- Query the sum of the interval [1,3]



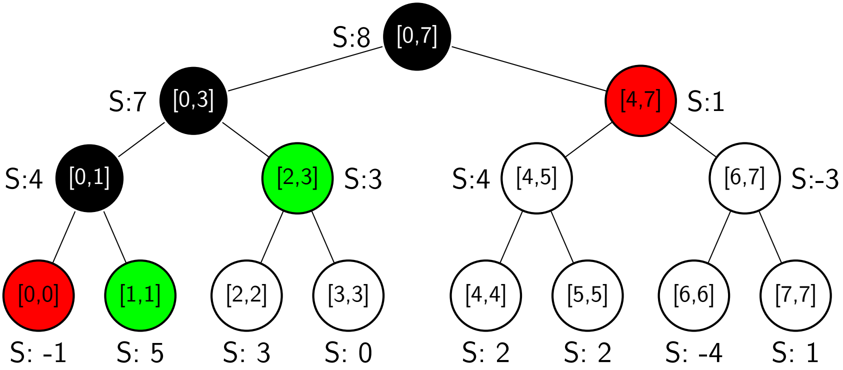
Querying Example

- Query the sum of the interval [1,3]



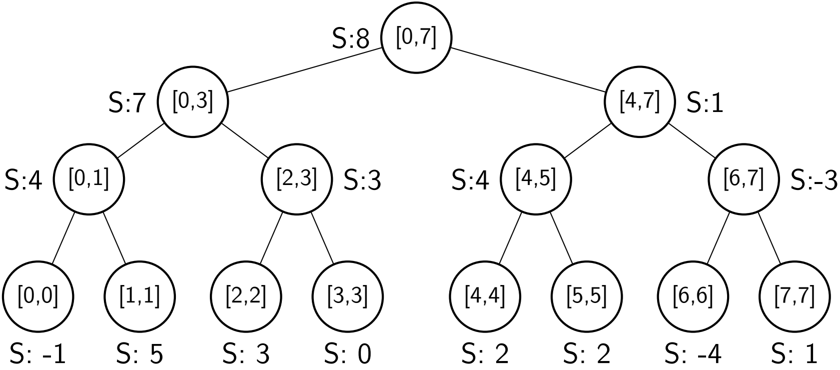
Querying Example

- Query the sum of the interval [1,3]



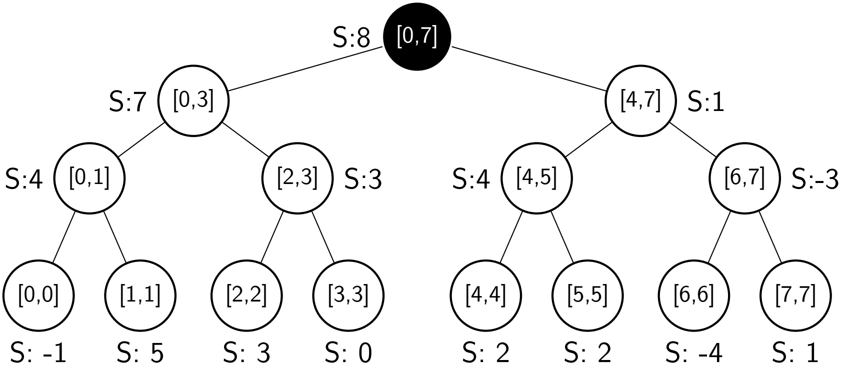
Querying Example

- Query the sum of the interval [3,6]



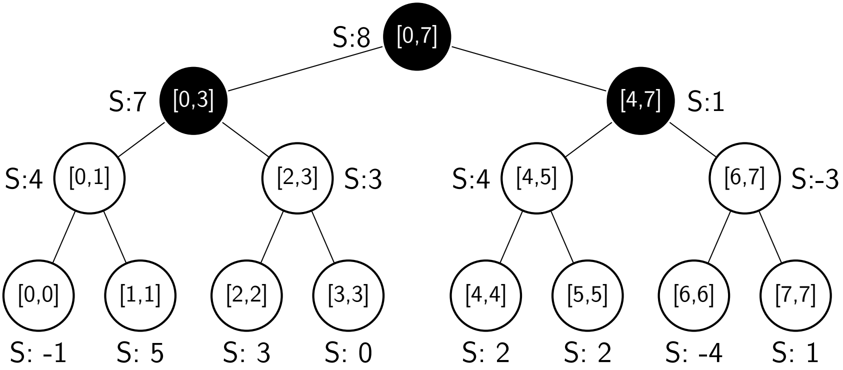
Querying Example

- Query the sum of the interval [3,6]



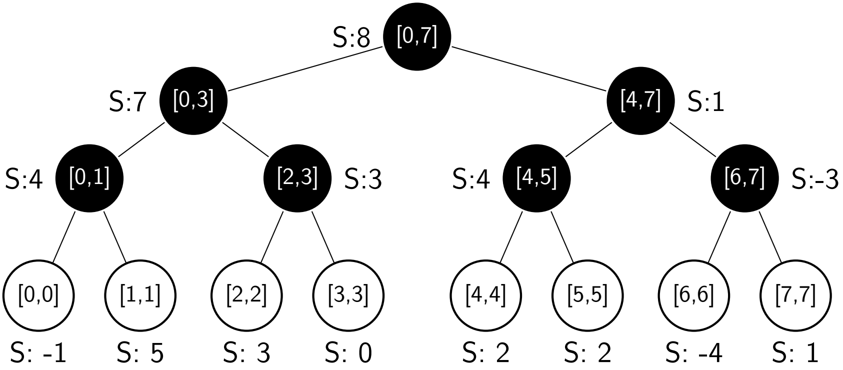
Querying Example

- Query the sum of the interval [3,6]



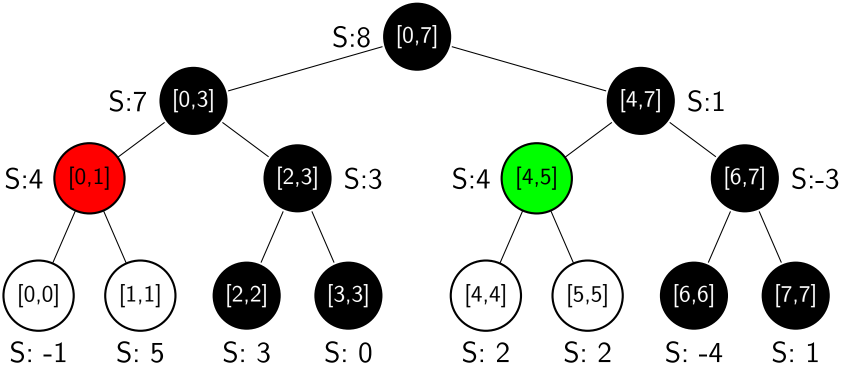
Querying Example

- Query the sum of the interval [3,6]



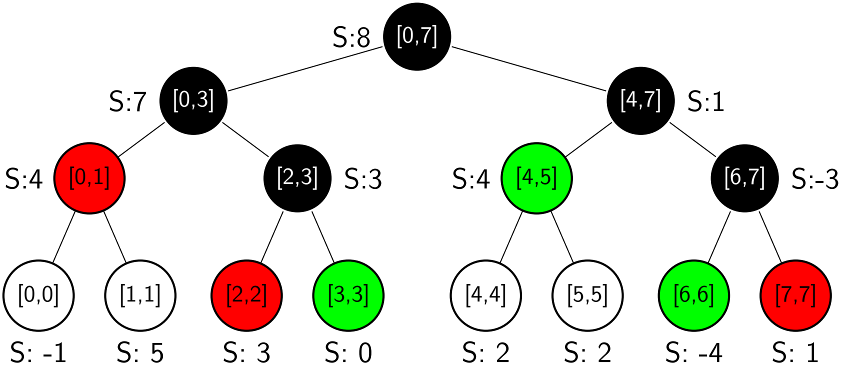
Querying Example

- Query the sum of the interval [3,6]



Querying Example

- Query the sum of the interval [3,6]



Updating the array

- Easy to update a single entry in $O(\log n)$ operations

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?
- $O(\log n)$ operations is still possible!

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?
- $O(\log n)$ operations is still possible!
- Have each node maintain a "delta" value for the subtree rooted at that node

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?
- $O(\log n)$ operations is still possible!
- Have each node maintain a "delta" value for the subtree rooted at that node
- Follow the same path as in query

Updating the array

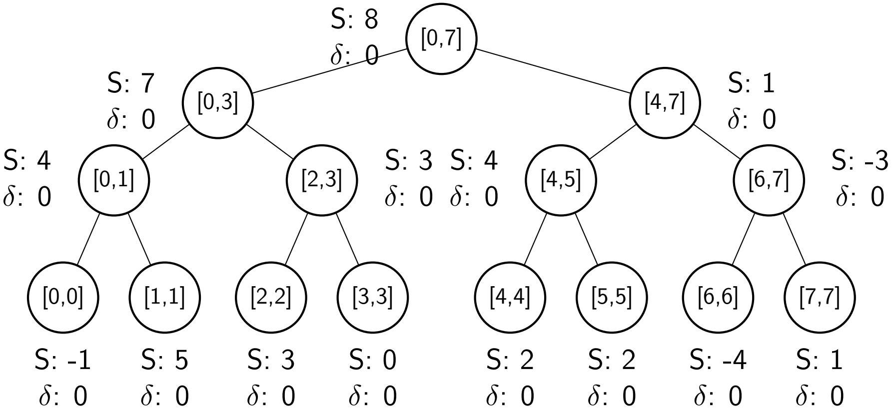
- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?
- $O(\log n)$ operations is still possible!
- Have each node maintain a "delta" value for the subtree rooted at that node
- Follow the same path as in query
- Only push the delta to a child node when the node is visited in another update or query

Updating the array

- Easy to update a single entry in $O(\log n)$ operations
  - Update value and push change up the path to the root
- But what about an update made to a range of values?
- $O(\log n)$ operations is still possible!
- Have each node maintain a "delta" value for the subtree rooted at that node
- Follow the same path as in query
- Only push the delta to a child node when the node is visited in another update or query
- This is called lazy propagation

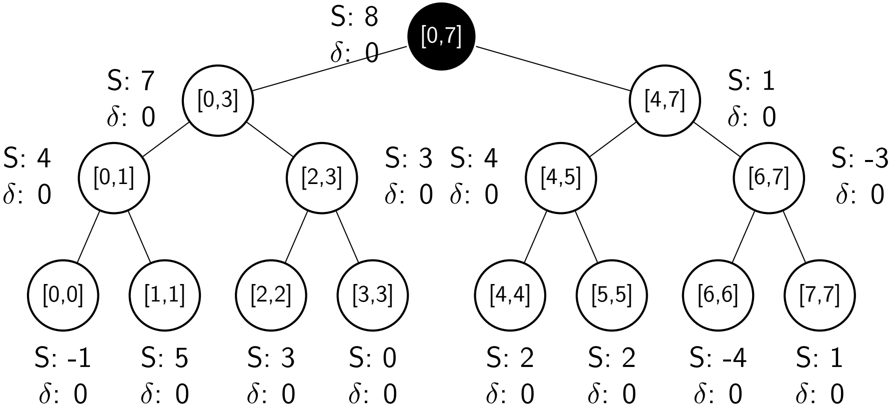
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



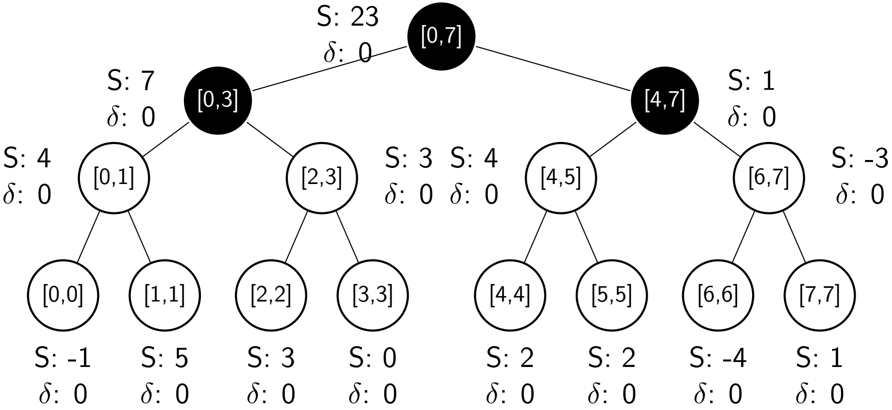
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



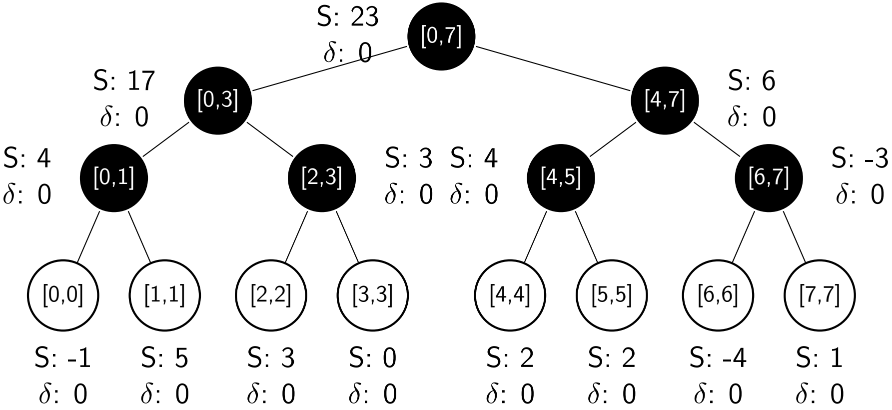
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



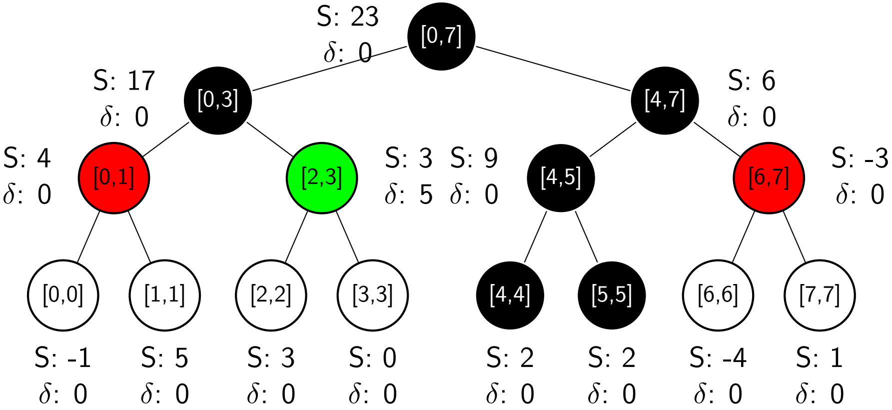
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



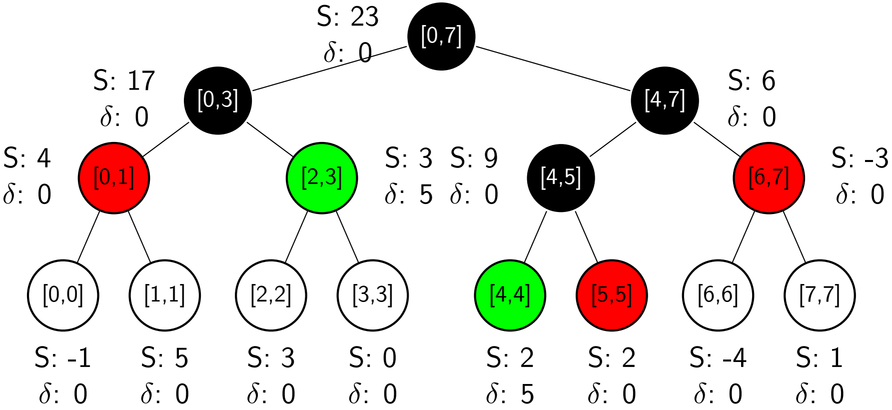
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



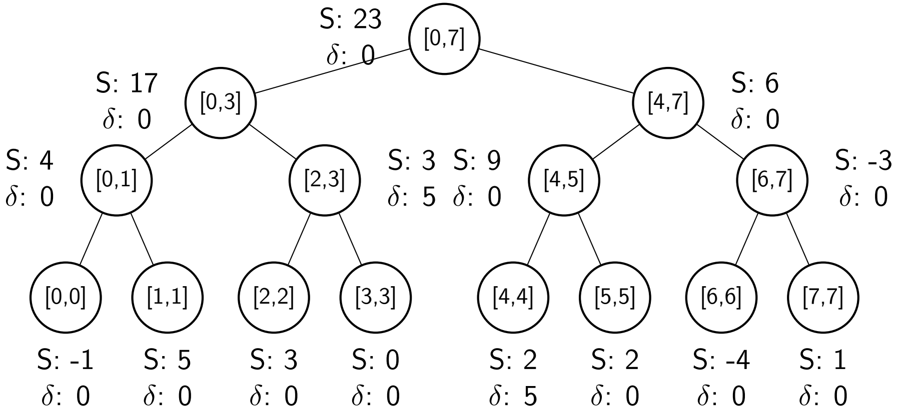
Lazy Propagation Example

- Increment every value in the range [2, 4] by 5.



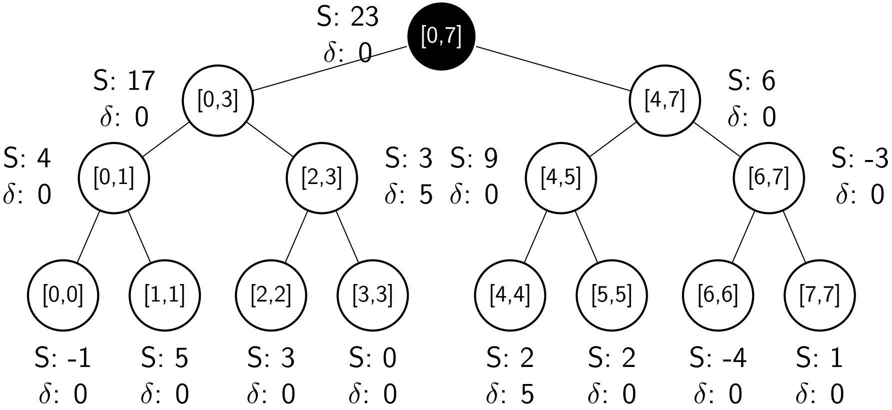
Lazy Propagation Example

- Now query [3,7]

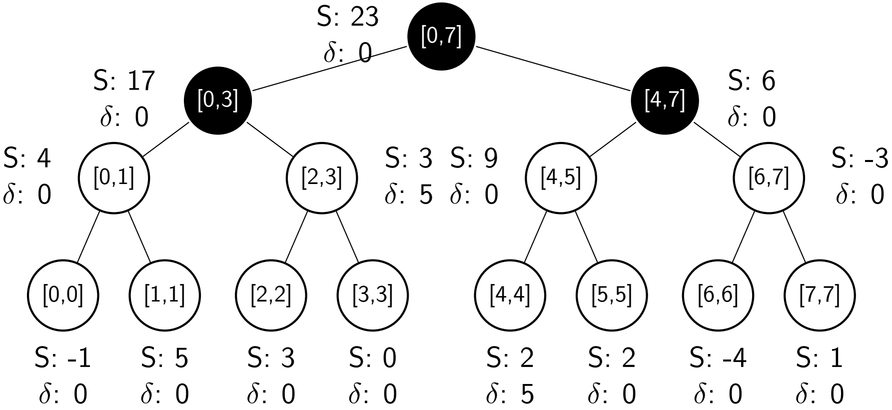


Lazy Propagation Example

- Now query [3,7]

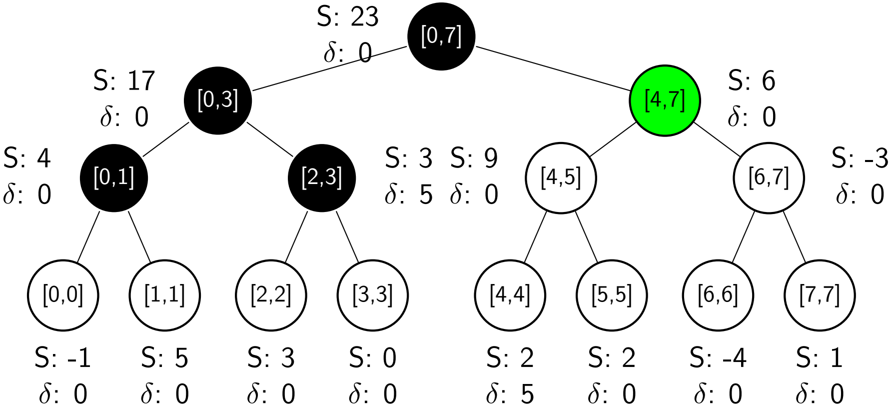


- Now query [3,7]


S: 23
δ: 0
[0,7]
S: 17
δ: 0
[0,3]
[4,7]
S: 6
δ: 0
S: 4
δ: 0
[0,1]
[2,3]
S: 3
δ: 5
S: 9
δ: 0
[4,5]
[6,7]
S: -3
δ: 0
[0,0]
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
[6,6]
[7,7]
S: -1
δ: 0
S: 5
δ: 0
S: 3
δ: 0
S: 0
δ: 0
S: 2
δ: 5
S: 2
δ: 0
S: -4
δ: 0
S: 1
δ: 0

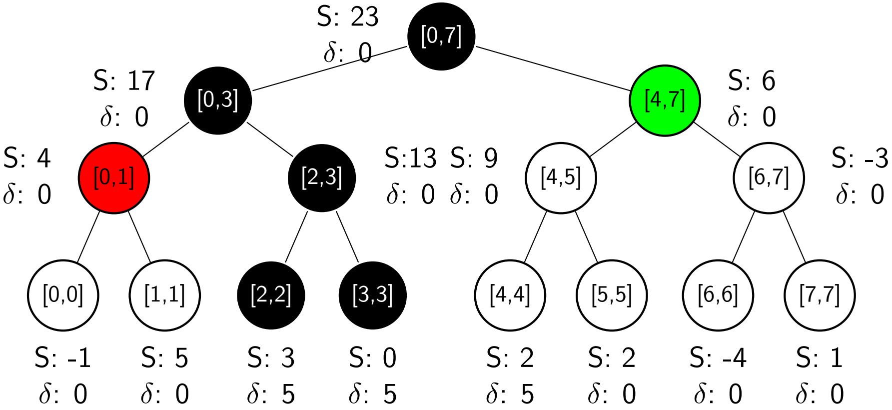
Lazy Propagation Example

- Now query [3,7]



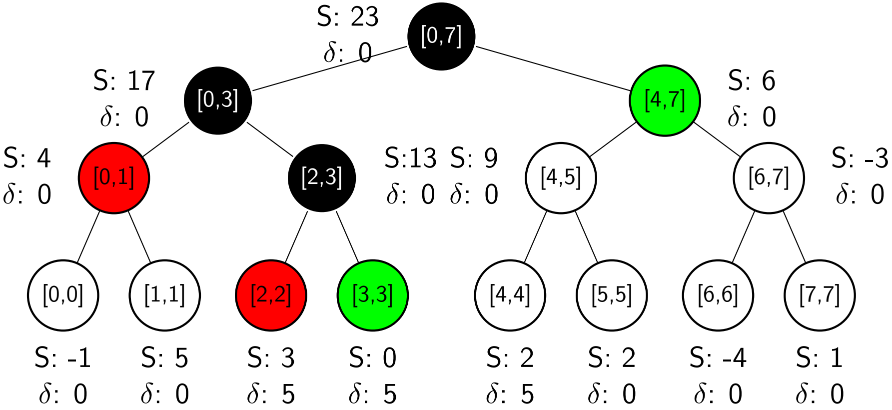
Lazy Propagation Example

- Now query [3,7]



Lazy Propagation Example

- Now query [3,7]



Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated
- We don't care about the rest of the points that aren't the endpoint of a query/update

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated
- We don't care about the rest of the points that aren't the endpoint of a query/update
- Read in and store all queries and updates to determine the important points, then perform queries and updates in the order given

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated
- We don't care about the rest of the points that aren't the endpoint of a query/update
- Read in and store all queries and updates to determine the important points, then perform queries and updates in the order given
- Example problems:

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated
- We don't care about the rest of the points that aren't the endpoint of a query/update
- Read in and store all queries and updates to determine the important points, then perform queries and updates in the order given
- Example problems:
  - Streaming Statistics: https://open.kattis.com/problems/streamstats

Coordinate Compression

- Sometimes the range of values that need to be covered is too large to have a node for every single value
- But there might be a manageable limit on the number of values queried/updated
- We don't care about the rest of the points that aren't the endpoint of a query/update
- Read in and store all queries and updates to determine the important points, then perform queries and updates in the order given
- Example problems:
  - Streaming Statistics: https://open.kattis.com/problems/streamstats
  - Worst Weather Ever https://open.kattis.com/problems/worstweather

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

General idea:

- Process the array of values left to right

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

General idea:

- Process the array of values left to right
- Use a Segment Tree to add the cost of all subarrays ending at the current index simultaneously

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

General idea:

- Process the array of values left to right
- Use a Segment Tree to add the cost of all subarrays ending at the current index simultaneously
- Min, max, and length values will be updated for a range of values along the way

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

General idea:

- Process the array of values left to right
- Use a Segment Tree to add the cost of all subarrays ending at the current index simultaneously
- Min, max, and length values will be updated for a range of values along the way
- Queries will be on an interval starting at 0 and ending at the current index

Example problem: Norma

Norma

Define the cost of an array to be the product of its length, its min value, and its max value. Given an array of 500,000 values in the range $[0, 10^8]$ find the sum (modulo 1 billion) of the costs of all contiguous subarrays of the original.

General idea:

- Process the array of values left to right
- Use a Segment Tree to add the cost of all subarrays ending at the current index simultaneously
- Min, max, and length values will be updated for a range of values along the way
- Queries will be on an interval starting at 0 and ending at the current index
- A query of a leaf node with array index i corresponds to getting the cost of the subarray from i to the current ending index

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range
- Updating the min:

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range
- Updating the min:
    - There will always be exactly 1 contiguous interval that needs updating

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range
- Updating the min:
  - There will always be exactly 1 contiguous interval that needs updating
  - Do preprocessing of the array to determine intervals that will need updating when a new value is added

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range
- Updating the min:
  - There will always be exactly 1 contiguous interval that needs updating
  - Do preprocessing of the array to determine intervals that will need updating when a new value is added
- Updating the max is similar

Example problem: Norma

To use a segment tree, we then need to know that the necessary updates and queries can be performed efficiently

- From the previous ending index we know the mins, maxes, and lengths of all subarrays ending at the previous ending index
- Update of length is easy: add one to everything in range
- Updating the min:
  - There will always be exactly 1 contiguous interval that needs updating
  - Do preprocessing of the array to determine intervals that will need updating when a new value is added
- Updating the max is similar

This makes sense for the leaf nodes, but what about the interior nodes?

- Ultimately each node will need to store a sum of min*max*length for the interval that it covers. We will see that nodes will store sums of maxes, sums of mins, and sums of lengths, and 3 other variables too.

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.
- So this sum will become $\Sigma_{i=lo}^{hi} m_{new} \cdot M_i \cdot L_i = m_{new} \Sigma_{i=lo}^{hi} M_i L_i$

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.
- So this sum will become $\Sigma_{i=lo}^{hi} m_{new} \cdot M_i \cdot L_i = m_{new} \Sigma_{i=lo}^{hi} M_i L_i$
- So if we knew $\Sigma_{i=lo}^{hi} M_i L_i$ we could update easily $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.
- So this sum will become $\Sigma_{i=lo}^{hi} m_{new} \cdot M_i \cdot L_i = m_{new} \Sigma_{i=lo}^{hi} M_i L_i$
- So if we knew $\Sigma_{i=lo}^{hi} M_i L_i$ we could update easily $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Similar arguments for max and length show that we would need to also store $\Sigma_{i=lo}^{hi} m_i \cdot L_i$ and $\Sigma_{i=lo}^{hi} m_i \cdot M_i$

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.
- So this sum will become $\Sigma_{i=lo}^{hi} m_{new} \cdot M_i \cdot L_i = m_{new} \Sigma_{i=lo}^{hi} M_i L_i$
- So if we knew $\Sigma_{i=lo}^{hi} M_i L_i$ we could update easily $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Similar arguments for max and length show that we would need to also store $\Sigma_{i=lo}^{hi} m_i \cdot L_i$ and $\Sigma_{i=lo}^{hi} m_i \cdot M_i$
- How to update these new sums?

Example problem: Norma

- If the min changes for some interval, how does this change the sum of the products of min, max, and length?
- Sum of products is $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Every node in new interval is getting the same new minimum.
- So this sum will become $\Sigma_{i=lo}^{hi} m_{new} \cdot M_i \cdot L_i = m_{new} \Sigma_{i=lo}^{hi} M_i L_i$
- So if we knew $\Sigma_{i=lo}^{hi} M_i L_i$ we could update easily $\Sigma_{i=lo}^{hi} m_i \cdot M_i \cdot L_i$
- Similar arguments for max and length show that we would need to also store $\Sigma_{i=lo}^{hi} m_i \cdot L_i$ and $\Sigma_{i=lo}^{hi} m_i \cdot M_i$
- How to update these new sums?
- Similar to the argument above, tracking Σm_i, ΣM_i, and ΣL_i for each node will allow this and these new sums can be updated easily based on the length of the interval.

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node
- `smML` - the sum of the products of min, max, and length covered by the node

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node
- `smML` - the sum of the products of min, max, and length covered by the node

Updates need to be able to handle:

- Increment lengths of subarrays by one for all subarrays in interval

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node
- `smML` - the sum of the products of min, max, and length covered by the node

Updates need to be able to handle:

- Increment lengths of subarrays by one for all subarrays in interval
- Set a new minimum for all subarrays in some interval

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node
- `smML` - the sum of the products of min, max, and length covered by the node

Updates need to be able to handle:

- Increment lengths of subarrays by one for all subarrays in interval
- Set a new minimum for all subarrays in some interval
- Set a new maximum for all subarrays in some interval

Example problem: Norma

So each node will store:

- `len` - the length of the interval covered by the node
- `sm` - the sum of the minimums covered by the node
- `sM` - the sum of the maximums covered by the node
- `sL` - the sum of the lengths covered by the node
- `smM` - the sum of the products of min and max covered by the node
- `smL` - the sum of the products of min and length covered by the node
- `sML` - the sum of the products of max and length covered by the node
- `smML` - the sum of the products of min, max, and length covered by the node

Updates need to be able to handle:

- Increment lengths of subarrays by one for all subarrays in interval
- Set a new minimum for all subarrays in some interval
- Set a new maximum for all subarrays in some interval

Queries need to get sum of the products of min, max, and length

Other example problems

- https://open.kattis.com/problems/palindromes
- https://open.kattis.com/problems/unrealestate
- https://open.kattis.com/problems/nonboringsequences
- Anything solvable with a Fenwick tree can be solved using a Segment Tree, but the code is more complicated and there is a little more overhead

Hashing

- A hash function is a mapping from any input value to an element of a finite set of values

Hashing

- A hash function is a mapping from any input value to an element of a finite set of values
- If the size of the set of possible inputs is greater than the set of possible hash values there will be collisions

Hashing

- A hash function is a mapping from any input value to an element of a finite set of values
- If the size of the set of possible inputs is greater than the set of possible hash values there will be collisions
- A *rolling* hash is a hash that can be updated easily if the input value is modified in some small way

Hashing

- A hash function is a mapping from any input value to an element of a finite set of values
- If the size of the set of possible inputs is greater than the set of possible hash values there will be collisions
- A *rolling* hash is a hash that can be updated easily if the input value is modified in some small way
- E.g. Given the hash of String *abac*, find the hash of *abacd*, or *bac*

Hashing

- A hash function is a mapping from any input value to an element of a finite set of values
- If the size of the set of possible inputs is greater than the set of possible hash values there will be collisions
- A *rolling* hash is a hash that can be updated easily if the input value is modified in some small way
- E.g. Given the hash of String *abac*, find the hash of *abacd*, or *bac*
- Implementation of the hash function is dependent on what constitutes "equality" of input values

Frequency Hash

- Two strings should be considered equal if they have the same number of each character

Frequency Hash

- Two strings should be considered equal if they have the same number of each character
- E.g. $abaa = baaa$ and $abccd = dcbac = ccadb$

Frequency Hash

- Two strings should be considered equal if they have the same number of each character
- E.g. $abaa = baaa$ and $abccd = dcbac = ccadb$
- So if we are to hash these strings we want them to map to the same value

Frequency Hash

- Two strings should be considered equal if they have the same number of each character
- E.g. $abaa = baaa$ and $abccd = dcbac = ccadb$
- So if we are to hash these strings we want them to map to the same value

The Fundamental Theorem of Arithmetic

Every integer n with $n > 1$ can be written uniquely as a product of powers of primes: $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where each p_i is a prime number, $p_1 < p_2 < \cdots < p_k$, and each e_i is a positive integer.

- Have each character correspond to a prime number, and let its frequency be the exponent for its associated prime

Frequency Hash

- Two strings should be considered equal if they have the same number of each character
- E.g. $abaa = baaa$ and $abccd = dcbac = ccadb$
- So if we are to hash these strings we want them to map to the same value

The Fundamental Theorem of Arithmetic

Every integer n with $n > 1$ can be written uniquely as a product of powers of primes: $n = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$, where each p_i is a prime number, $p_1 < p_2 < \cdots < p_k$, and each e_i is a positive integer.

- Have each character correspond to a prime number, and let its frequency be the exponent for its associated prime
- Commutativity of multiplication and the Fundamental Theorem of Arithmetic ensure that Strings with matching character frequencies map to the same hash value

Frequency Hash

a	b	c	d	e	f	g	h	i	j	k	l	m
2	3	5	7	11	13	17	19	23	29	31	37	41

n	o	p	q	r	s	t	u	v	w	x	y	z
43	47	53	59	61	67	71	73	79	83	89	97	101

- For hash function H from Strings to integers:
- $H(abcc) = 2 \cdot 3 \cdot 5 \cdot 5 = 2^1 3^1 5^2 = 150$
- $H(cbac) = 5 \cdot 3 \cdot 2 \cdot 5 = 2^1 3^1 5^2 = 150$

- Easy to implement a rolling hash with this representation

Frequency Hash

a	b	c	d	e	f	g	h	i	j	k	l	m
2	3	5	7	11	13	17	19	23	29	31	37	41

n	o	p	q	r	s	t	u	v	w	x	y	z
43	47	53	59	61	67	71	73	79	83	89	97	101

- For hash function H from Strings to integers:
- $H(abcc) = 2 \cdot 3 \cdot 5 \cdot 5 = 2^1 3^1 5^2 = 150$
- $H(cbac) = 5 \cdot 3 \cdot 2 \cdot 5 = 2^1 3^1 5^2 = 150$

- Easy to implement a rolling hash with this representation
- Add a new character by multiplying by a prime

Frequency Hash

a	b	c	d	e	f	g	h	i	j	k	l	m
2	3	5	7	11	13	17	19	23	29	31	37	41

n	o	p	q	r	s	t	u	v	w	x	y	z
43	47	53	59	61	67	71	73	79	83	89	97	101

- For hash function H from Strings to integers:
- $H(abcc) = 2 \cdot 3 \cdot 5 \cdot 5 = 2^1 3^1 5^2 = 150$
- $H(cbac) = 5 \cdot 3 \cdot 2 \cdot 5 = 2^1 3^1 5^2 = 150$

- Easy to implement a rolling hash with this representation
- Add a new character by multiplying by a prime
- Remove a character by dividing by a prime

Positional Hash

- Two strings should be considered equal if every character matches at every position

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$
- So use base 27: $a \to 1, b \to 2, \ldots, z \to 26$

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$
- So use base 27: $a \rightarrow 1, b \rightarrow 2, \ldots, z \rightarrow 26$
- Examples:

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$
- So use base 27: $a \to 1, b \to 2, \ldots, z \to 26$
- Examples:
  - $H(adcc) = 1 \cdot 27^3 + 4 \cdot 27^2 + 3 \cdot 27^1 + 3 \cdot 27^0 = 22683$

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$
- So use base 27: $a \to 1, b \to 2, \ldots, z \to 26$
- Examples:
  - $H(adcc) = 1 \cdot 27^3 + 4 \cdot 27^2 + 3 \cdot 27^1 + 3 \cdot 27^0 = 22683$
  - $H(a) = 1 \cdot 27^0 = 1$

Positional Hash

- Two strings should be considered equal if every character matches at every position
- E.g. $abc = abc$ and $abc \neq acb$ and $a \neq aa$
- So we want a unique hash value for each string
- Use base 26?
- This almost works, but if a maps to 0 then we would get $a = aa = aaaaaaaaaa$
- So use base 27: $a \to 1, b \to 2, \ldots, z \to 26$
- Examples:
  - $H(adcc) = 1 \cdot 27^3 + 4 \cdot 27^2 + 3 \cdot 27^1 + 3 \cdot 27^0 = 22683$
  - $H(a) = 1 \cdot 27^0 = 1$
  - $H(aaa) = 1 \cdot 27^2 + 1 \cdot 27^1 + 1 \cdot 27^0 = 757$

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently
- E.g. Given the hash of *abcfas* compute the hash of $(abcfas)^n$

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently
- E.g. Given the hash of *abcfas* compute the hash of $(abcfas)^n$
- $H((abcfas)^2) = 27^6 \cdot H(abcfas) + H(abcfas)$

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently
- E.g. Given the hash of *abcfas* compute the hash of $(abcfas)^n$
- $H((abcfas)^2) = 27^6 \cdot H(abcfas) + H(abcfas)$
- $H((abcfas)^4) = 27^{12} \cdot H((abcfas)^2) + H((abcfas)^2)$

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently
- E.g. Given the hash of *abcfas* compute the hash of $(abcfas)^n$
- $H((abcfas)^2) = 27^6 \cdot H(abcfas) + H(abcfas)$
- $H((abcfas)^4) = 27^{12} \cdot H((abcfas)^2) + H((abcfas)^2)$
- Use square and multiply algorithm idea to compute $H((abcfas)^n)$ for any n

Updating a positional hash

Appending a character (on the right)

- Multiply current hash by the base (27)
- Add the value of the new character

Removing a character (from the left)

- Subtract the character's value multiplied by the base to the exponent of the number of characters minus 1

- Can easily do more sophisticated updates efficiently
- E.g. Given the hash of *abcfas* compute the hash of $(abcfas)^n$
- $H((abcfas)^2) = 27^6 \cdot H(abcfas) + H(abcfas)$
- $H((abcfas)^4) = 27^{12} \cdot H((abcfas)^2) + H((abcfas)^2)$
- Use square and multiply algorithm idea to compute $H((abcfas)^n)$ for any n
- Example problem: Power Strings - open.kattis.com/problems/powerstrings

Collisions

- Both of these hashing strategies give extremely large hash values, even if the strings are relatively short

Collisions

- Both of these hashing strategies give extremely large hash values, even if the strings are relatively short
- Using BigInteger will be too slow for most applications so we will use some large prime modulus

Collisions

- Both of these hashing strategies give extremely large hash values, even if the strings are relatively short
- Using BigInteger will be too slow for most applications so we will use some large prime modulus
- This can lead to collisions, particularly if there is a large number of values that need to be hashed

Collisions

- Both of these hashing strategies give extremely large hash values, even if the strings are relatively short
- Using BigInteger will be too slow for most applications so we will use some large prime modulus
- This can lead to collisions, particularly if there is a large number of values that need to be hashed

Idea:

- Use multiple hash functions that use different moduli!

Collisions

- Both of these hashing strategies give extremely large hash values, even if the strings are relatively short
- Using BigInteger will be too slow for most applications so we will use some large prime modulus
- This can lead to collisions, particularly if there is a large number of values that need to be hashed

Idea:

- Use multiple hash functions that use different moduli!
- Chinese Remainder Theorem tells us that we will only get collisions for one in every $lcm(p_1, p_2, \ldots, p_k) = p_1 p_2 \cdots p_k$ values

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n+m)$ (dependent on collisions)

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n + m)$ (dependent on collisions)
- So KMP is generally preferable for matching a single string to a pattern of text

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n+m)$ (dependent on collisions)
- So KMP is generally preferable for matching a single string to a pattern of text
- But suppose you want to find a match in the text for one of multiple patterns...

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n+m)$ (dependent on collisions)
- So KMP is generally preferable for matching a single string to a pattern of text
- But suppose you want to find a match in the text for one of multiple patterns...
- Can store the hash of the multiple patterns in a set and then perform a rolling hash on the text and check if the set contains any of the hash values

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n+m)$ (dependent on collisions)
- So KMP is generally preferable for matching a single string to a pattern of text
- But suppose you want to find a match in the text for one of multiple patterns...
- Can store the hash of the multiple patterns in a set and then perform a rolling hash on the text and check if the set contains any of the hash values
- This depends on the number of hashes stored in the set. Using a HashSet to store the values can become too slow if $\sim$100,000+ hashes are being stored...

Applications and Complexity

- The rolling hash method is used in the Rabin-Karp algorithm for string matching
- To find a pattern of length m in text of length n the algorithm has probabilistic complexity $O(n+m)$ (dependent on collisions)
- So KMP is generally preferable for matching a single string to a pattern of text
- But suppose you want to find a match in the text for one of multiple patterns...
- Can store the hash of the multiple patterns in a set and then perform a rolling hash on the text and check if the set contains any of the hash values
- This depends on the number of hashes stored in the set. Using a HashSet to store the values can become too slow if $\sim$100,000+ hashes are being stored... But there's a solution to this!

Bloom Filters!

Bloom Filters

- Probabilistic set data structure

Bloom Filters

- Probabilistic set data structure
- Queries can have false positives but no false negatives

Bloom Filters

- Probabilistic set data structure
- Queries can have false positives but no false negatives
  - A query for something that is in the set will return true - that the element is in the set

Bloom Filters

- Probabilistic set data structure
- Queries can have false positives but no false negatives
  - A query for something that is in the set will return true - that the element is in the set
  - A query for something that is not in the set may return true or false - that the element is, or is not in the set

Bloom Filters

- Probabilistic set data structure
- Queries can have false positives but no false negatives
  - A query for something that is in the set will return true - that the element is in the set
  - A query for something that is not in the set may return true or false - that the element is, or is not in the set
- We have some control over the probability, so we can make it such that the probability of a false positive is smaller than the probability of a hardware failure

Bloom Filters

- Probabilistic set data structure
- Queries can have false positives but no false negatives
  - A query for something that is in the set will return true - that the element is in the set
  - A query for something that is not in the set may return true or false - that the element is, or is not in the set
- We have some control over the probability, so we can make it such that the probability of a false positive is smaller than the probability of a hardware failure
- Used together with hash functions

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

For example:
Suppose we have 2 positional hashing functions (as described above), H_1 and H_2, with moduli 5 and 7 respectively.
We would have 2 BitSets, S_1 and S_2, initialised to $S_1 = 00000$ and $S_2 = 0000000$.

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

For example:
Suppose we have 2 positional hashing functions (as described above), H_1 and H_2, with moduli 5 and 7 respectively.
We would have 2 BitSets, S_1 and S_2, initialised to $S_1 = 00000$ and $S_2 = 0000000$.

String	$H_1(\cdot)$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

For example:
Suppose we have 2 positional hashing functions (as described above), H_1 and H_2, with moduli 5 and 7 respectively.
We would have 2 BitSets, S_1 and S_2, initialised to $S_1 = 00000$ and $S_2 = 0000000$.

String	$H_1(\cdot)$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

For example:
Suppose we have 2 positional hashing functions (as described above), H_1 and H_2, with moduli 5 and 7 respectively.
We would have 2 BitSets, S_1 and S_2, initialised to $S_1 = 00000$ and $S_2 = 0000000$.

String	$H_1()$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001
ac	$30 \bmod 5 = 0$	$30 \bmod 7 = 2$	11000	0010001

Bloom Filters - Insertion

- The set is represented as one or more BitSets of size equal to the modulus of the associated hash function
- A bit in the BitSet is set if there is some input that hashes to that value

For example:
Suppose we have 2 positional hashing functions (as described above), H_1 and H_2, with moduli 5 and 7 respectively.
We would have 2 BitSets, S_1 and S_2, initialised to $S_1 = 00000$ and $S_2 = 0000000$.

String	$H_1()$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001
ac	$30 \bmod 5 = 0$	$30 \bmod 7 = 2$	11000	0010001
a	$1 \bmod 5 = 1$	$1 \bmod 7 = 1$	11000	0110001

Bloom Filters - Querying

String	$H_1()$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001
ac	$30 \bmod 5 = 0$	$30 \bmod 7 = 2$	11000	0010001
a	$1 \bmod 5 = 1$	$1 \bmod 7 = 1$	11000	0110001

- Querying each of f, ac, and a will return true as the appropriate bits are set in S_1 and S_2

Bloom Filters - Querying

String	$H_1()$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001
ac	$30 \bmod 5 = 0$	$30 \bmod 7 = 2$	11000	0010001
a	$1 \bmod 5 = 1$	$1 \bmod 7 = 1$	11000	0110001

- Querying each of f, ac, and a will return true as the appropriate bits are set in S_1 and S_2
- Querying b will return false. $H_2(b) = 2$ and this bit is set in S_2, but $H_1(b) = 2$ and this bit is not set in S_1

Bloom Filters - Querying

String	$H_1(\cdot)$	$H_2(\cdot)$	S_1	S_2
-	-	-	00000	0000000
f	$6 \bmod 5 = 1$	$6 \bmod 7 = 6$	01000	0000001
ac	$30 \bmod 5 = 0$	$30 \bmod 7 = 2$	11000	0010001
a	$1 \bmod 5 = 1$	$1 \bmod 7 = 1$	11000	0110001

- Querying each of f, ac, and a will return true as the appropriate bits are set in S_1 and S_2
- Querying b will return false. $H_2(b) = 2$ and this bit is set in S_2, but $H_1(b) = 2$ and this bit is not set in S_1
- Querying p will return true. $H_1(p) = 16 \bmod 5 = 1$ and $H_2(p) = 16 \bmod 7 = 2$ and these bits are set in S_1 and S_2 respectively

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$
- Use more hashing functions and BitSets

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$
- Use more hashing functions and BitSets
- Optimal number of primes is determined by the number of values to be stored in the set and the sizes of the sets

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$
- Use more hashing functions and BitSets
- Optimal number of primes is determined by the number of values to be stored in the set and the sizes of the sets
- http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$
- Use more hashing functions and BitSets
- Optimal number of primes is determined by the number of values to be stored in the set and the sizes of the sets
- http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html
- The optimal number of BitSets is $\frac{m}{n} \ln 2$ (m is the size of the moduli, n is the number of values being stored), get as close to this as the time limit will allow

Bloom Filters - Reducing collisions

- Use larger moduli - e.g. $10^9 + 7, 10^9 + 9$
- Use more hashing functions and BitSets
- Optimal number of primes is determined by the number of values to be stored in the set and the sizes of the sets
- http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html
- The optimal number of BitSets is $\frac{m}{n} \ln 2$ (m is the size of the moduli, n is the number of values being stored), get as close to this as the time limit will allow
- Using 6 BitSets of size $\sim 10^9$, and storing 10 million values, querying 10 million values has a low probability of having 1 or more false positives.

Example problem and Sources

Requires rolling hashes and Bloom filters:
maps17.kattis.com/problems/maps17.kingofspades

Videos on segment trees and hashing and bloom filters

- Algorithms Live! - Segment Trees
- Algorithms Live! - Hashing and Bloom Filters