CPSC 490 – Problem Solving in Computer Science

Lecture 12: SQRT Decomposition

Jason Chiu and Raunak Kumar

Based on slides by Paul Liu (2014), Kuba Karpierz and Bruno Vacherot (2015)

2017/02/01

University of British Columbia

Range Query

You are given an array A of n integers. How efficiently can you answer these queries?

Range Query

You are given an array A of n integers. How efficiently can you answer these queries?

- `sum(i, j) = A[i] + A[i+1] + ... + A[j]`
- `min(i, j) = min(A[i], ..., A[j])`
- Median
- Mode

Range Query

You are given an array A of n integers. How efficiently can you answer these queries?

- `sum(i, j) = A[i] + A[i+1] + ... + A[j]`
- `min(i, j) = min(A[i], ..., A[j])`
- Median
- Mode
- What if we allow updates?

Range Minimum Query with Updates

Given an array A of n integers, perform these queries efficiently:

- `update(i, x)`: set `A[i] = x`
- `min(i, j)`: return `min(A[i], A[i+1], ... , A[j])`

Potential Solutions

We could try the following:

- Option 1: Naively perform the 2 queries.
  - `update(i, x)` takes $O(1)$ time
  - `min(i, j)` takes $O(n)$ time.

Potential Solutions

We could try the following:

- Option 1: Naively perform the 2 queries.
    - `update(i, x)` takes $O(1)$ time
    - `min(i, j)` takes $O(n)$ time.
- Option 2: Precompute the minimum in $O(n^2)$ intervals.
    - `update(i, x)` takes $O(n)$ time, to change $O(n)$ intervals
    - `min(i, j)` takes $O(1)$ time

Potential Solutions

We could try the following:

- Option 1: Naively perform the 2 queries.
  - `update(i, x)` takes $O(1)$ time
  - `min(i, j)` takes $O(n)$ time.
- Option 2: Precompute the minimum in $O(n^2)$ intervals.
  - `update(i, x)` takes $O(n)$ time, to change $O(n)$ intervals
  - `min(i, j)` takes $O(1)$ time

Can we do better? Can we do a different type of precomputation that allows us to `update` and `min` fast?

Better Solution

Idea: Divide array into k blocks and precompute min of each block!

Better Solution

Idea: Divide array into k blocks and precompute min of each block!

- `update(i, x)`: Update the minimum in the relevant block in $O(n/k)$ time (since each block has n/k elements).
- `min(i, j)`: Lookup the relevant minimums from $O(k)$ blocks.

But how do we choose k?

Choosing k

Smaller k makes `min` query fast, larger k makes `update` query fast.

Choosing k

Smaller k makes `min` query fast, larger k makes `update` query fast.
So just balance the two!

$$k = \frac{n}{k} \quad \Leftrightarrow \quad k = \sqrt{n}$$

Divide the array into $\sqrt{n}$ blocks!

Choosing k

Smaller k makes `min` query fast, larger k makes `update` query fast.
So just balance the two!

$$k = \frac{n}{k} \quad \Leftrightarrow \quad k = \sqrt{n}$$

Divide the array into $\sqrt{n}$ blocks!

- If $\sqrt{n}$ is not an integer, just round down or up, doesn't matter.
- The last block may have size $< \sqrt{n}$, that's fine.

Square Root Decomposition

Useful technique for speeding up associative range queries.

- Divide the array into $\sqrt{n}$ blocks and precompute minimums for each block. $O(n)$.

Square Root Decomposition

Useful technique for speeding up associative range queries.

- Divide the array into $\sqrt{n}$ blocks and precompute minimums for each block. $O(n)$.
- `update(i, x)`: Go to the block $\lfloor i/\sqrt{n} \rfloor$. Set `A[i] = x` and update minimum in that block. $O(\sqrt{n})$.

Square Root Decomposition

Useful technique for speeding up associative range queries.

- Divide the array into $\sqrt{n}$ blocks and precompute minimums for each block. $O(n)$.
- `update(i, x)`: Go to the block $\lfloor i/\sqrt{n} \rfloor$. Set `A[i] = x` and update minimum in that block. $O(\sqrt{n})$.
- `min(i, j)`: Determine the start and end blocks.
    - If start/end lie in the middle of a block, brute force those sections.
    - For all the blocks in the middle, lookup the precomputed answer.
    - We loop over at most $2\sqrt{n}$ elements and $\sqrt{n}$ blocks. $O(\sqrt{n})$.

SQRT Decomposition - Example

We can use the same idea to answer `sum(i,j)` queries!

SQRT Decomposition - Example

We can use the same idea to answer `sum(i,j)` queries!

What do we store in each block?

SQRT Decomposition - Example

We can use the same idea to answer `sum(i,j)` queries!

What do we store in each block? The sum for that block.

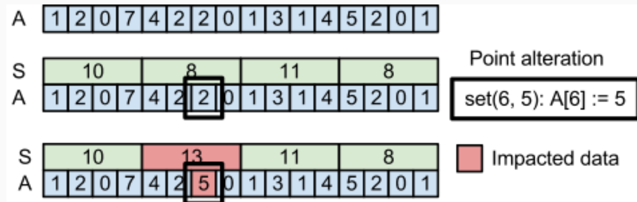
SQRT Decomposition - Example

`sum(2, 14)`:

Source: `http://www.infoarena.ro/blog/square-root-trick`

SQRT Decomposition - Example

`update(6, 5)`:



Source: `http://www.infoarena.ro/blog/square-root-trick`

Queries Without Updates

What about range query problems without updates?

Queries Without Updates

What about range query problems without updates?

If there are no updates, we can process queries in a different order, and return the results all at once!

In other words, we do queries offline!

Square Root Decomposition of Queries

Observation 1: if all queries have the same left end point, then we can sort by right end point and then do linear time sweep!

- Min queries: if we know `min(a, b)` and we want `min(a, c)`, just min with elements from index $b + 1$ to c.

Square Root Decomposition of Queries

Observation 1: if all queries have the same left end point, then we can sort by right end point and then do linear time sweep!

- Min queries: if we know `min(a, b)` and we want `min(a, c)`, just min with elements from index $b+1$ to c.

Observation 2: if left end points differ by a little, then moving the left end back and forth is not too expensive, so our strategy still works.

- Min queries (again): to get from `min(a, b)` to `min(c, d)`, keep a set elements from a to b, then add elements from $b+1$ to d, and add/delete elements between a and c.

Square Root Decomposition of Queries

Idea: Sort the queries by the $\sqrt{n}$ block their left endpoint is in, and then by their right endpoint.

Query $(a, b) < (c, d)$ iff:

- $\lfloor a/\sqrt{n} \rfloor < \lfloor c/\sqrt{n} \rfloor$ or
- $\lfloor a/\sqrt{n} \rfloor = \lfloor c/\sqrt{n} \rfloor$ and $b < d$.

Square Root Decomposition of Queries

After sorting the queries, we need an efficient way to get result for interval $[a_{t+1}, b_{t+1}]$ given result for $[a_t, b_t]$.

Square Root Decomposition of Queries

After sorting the queries, we need an efficient way to get result for interval $[a_{t+1}, b_{t+1}]$ given result for $[a_t, b_t]$.

For min queries, an balanced BST storing the set of elements in the current interval will do the job in $O(\log n)$ per element change.

Square Root Decomposition of Queries

Number of element changes

- Left end point moves $\leq \sqrt{n}$ times per query
- Right end point moves $\leq n$ times per block

$\Rightarrow O(q\sqrt{n} + n\sqrt{n})$ element changes in total

Square Root Decomposition of Queries

Number of element changes

- Left end point moves $\leq \sqrt{n}$ times per query
- Right end point moves $\leq n$ times per block

$\Rightarrow O(q\sqrt{n} + n\sqrt{n})$ element changes in total

Time complexity for min query

- Sorting queries take $O(q \log q)$
- For min query, each element change costs $O(\log n)$

$\Rightarrow O(q \log q + (n + q)\sqrt{n} \log n)$

Problem 1

Given array A of $n \leq 10,000$ ints, and an int $0 \leq k \leq 1,000,000$.

Answer $q \leq 10000$ queries of the following form:

For query (a, b), return the number of pairs of indices i, j such that

- $a \leq i < j \leq b$
- $A[i] \oplus A[i+1] \oplus \cdots \oplus A[j] = k$.

where $\oplus$ is the bit-wise XOR operator.

Hint: bit-wise XOR of integers = addition in vector space of {0, 1} and addition = subtraction when you only have two numbers 0 and 1

Problem 1 - Solution

No updates $\Rightarrow$ let's use square root decomposition for offline query

From answer to query (a, b), how do we get answer for $(a, b + 1)$?

Problem 1 - Solution

First, let's calculate a prefix xor array:

$$P[0] = 0, P[i] = P[i-1] \oplus A[i]$$

Problem 1 - Solution

First, let's calculate a prefix xor array:

$$P[0] = 0, P[i] = P[i-1] \oplus A[i]$$

Note that because subtraction is same as addition

$$A[i] \oplus \cdots \oplus A[j] = P[j] \oplus P[i-1]$$

Problem 1 - Solution

First, let's calculate a prefix xor array:

$$P[0] = 0, P[i] = P[i-1] \oplus A[i]$$

Note that because subtraction is same as addition

$$A[i] \oplus \cdots \oplus A[j] = P[j] \oplus P[i-1]$$

$\Rightarrow$ Query (a, b) is counting the number of pairs i, j such that

$$a - 1 \leq i < j \leq b \qquad \text{and} \qquad P[i] \oplus P[j] = k$$

Problem 1 - Solution

First, let's calculate a prefix xor array:

$$P[0] = 0, P[i] = P[i-1] \oplus A[i]$$

Note that because subtraction is same as addition

$$A[i] \oplus \cdots \oplus A[j] = P[j] \oplus P[i-1]$$

$\Rightarrow$ Query (a, b) is counting the number of pairs i, j such that

$$a - 1 \leq i < j \leq b \qquad \text{and} \qquad P[i] \oplus P[j] = k$$

Also, note that $P[i] \oplus P[j] = k \Leftrightarrow P[j] = k \oplus P[i]$.

Problem 1 - Solution

Maintain a count array (or hash map) *cnt*.

Problem 1 - Solution

Maintain a count array (or hash map) *cnt*.

Add an element $A[b+1]$ to the right end of interval

- Look at how many $k \oplus P[b+1]$ we have in $P[a-1], \ldots, P[b]$
- $\Rightarrow$ add $cnt[k \oplus P[b+1]]$ to result and increment $cnt[P[b+1]]$

Problem 1 - Solution

Maintain a count array (or hash map) *cnt*.

Add an element $A[b+1]$ to the right end of interval

- Look at how many $k \oplus P[b+1]$ we have in $P[a-1], \ldots, P[b]$
- $\Rightarrow$ add $cnt[k \oplus P[b+1]]$ to result and increment $cnt[P[b+1]]$

Remove an element $A[a]$ from left end of interval

- Look at how many $k \oplus P[a-1]$ we have in $P[a], \ldots, P[b]$
- $\Rightarrow$ subtract $cnt[k \oplus P[a-1]]$ from result and decrement $cnt[P[a-1]]$, beware when $P[a-1] = k \oplus P[a-1]$

Addition on the left end is handled similarly.

Problem 2

You are given an array A with $n \leq 10,000$ integers.

Answer $q \leq 10,000$ queries of the form *mode(l,r)*.

No updates.

Problem 2 - Solution

No updates $\Rightarrow$ let's use square root decomposition for offline query

But how to get mode of $[a_{t+1}, b_{t+1}]$ given mode of $[a_t, b_t]$?

Problem 2 - Solution

Keep track of two things:

- The running count of each element.
- An ordered set of (element, count), sorted by count so that we can obtain the mode fast.

Problem 2 - Solution

Keep track of two things:

- The running count of each element.
- An ordered set of (element, count), sorted by count so that we can obtain the mode fast.

If we have the above information for an interval $[a, b]$, then to go to e.g. $[a, b+1]$ we just need to insert 1 element:

- Erase (new element, count) from the set
- Increment count of the new element
- Insert (new element, new count) back to the set

Deletion is pretty similar

Next Class

Segment Tree