Square Root Decomposition

Arnar Bjarni Arnarson

September 24, 2025

School of Computer Science
Reykjavík University

Range queries

- We have an array A of size n.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\text{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\mathrm{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.

Range queries

- We have an array A of size n.
- Given i, j, we want to answer:
  - $\max(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\min(A[i], A[i+1], \ldots, A[j-1], A[j])$
  - $\operatorname{sum}(A[i], A[i+1], \ldots, A[j-1], A[j])$
- We want to answer these queries efficiently, or in other words, without looking through all elements.
- Sometimes we also want to update elements.

Bucketing

- Group values into buckets of size k and store result of each bucket

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
    - change the array element
    - recompute corresponding bucket

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to “jump” over intervals of size k

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k

Bucketing

- Group values into buckets of size k and store result of each bucket
- Updating is easy:
  - change the array element
  - recompute corresponding bucket
- Time complexity: $O(k)$
- Again we want to query over a range
  - When a bucket is contained in the range, use the stored sum for the bucket
  - This (sometimes) allows us to "jump" over intervals of size k
  - Only have to go inside at most two buckets (each end)
  - Have to consider at most n/k buckets and 2 buckets of size k
- Time complexity: $O(n/k + k)$

Choosing number of buckets

- Now we have a data structure that supports:
    - Querying in $O(n/k + k)$

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$

Choosing number of buckets

- Now we have a data structure that supports:
  - Updating in $O(k)$
  - Querying in $O(n/k + k)$
- What k to pick?
- Time complexity is minimized for $k = \sqrt{n}$:
  - Updating in $O(\sqrt{n})$
  - Querying in $O(n/\sqrt{n} + \sqrt{n}) = O(\sqrt{n})$
- Also known as square root decomposition, and is a very powerful technique

Example problem: Supercomputer

- https://open.kattis.com/problems/supercomputer

Offline and online

- Usually we get all queries in a batch.
- This means we can compute the answers for all of them collectively. This is known as an offline algorithm.
- Conversely, an online algorithm is interactive and answers each query as it is received.
- The order in which we compute the answer may differ from the input order.
- After computing, we make sure to output them in the correct order.
- We will draw inspiration from the sliding window method.

Another sliding window method

- Recall that when we move the right endpoint of a range, very little changes.
- Idea: sort the ranges by (l, r).
- Then use the same methodology as sliding window.
- Define operations $\mathrm{add}_{\mathrm{R}}, \mathrm{add}_{\mathrm{L}}, \mathrm{del}_{\mathrm{R}}, \mathrm{del}_{\mathrm{L}}$ that shift the active range.
- Now we expand and contract as needed using these operations.

One slight problem

- What is the worst case for this method?

One slight problem

- What is the worst case for this method?
- Consider ranges $(0, N-1), (1, 1), (2, N-1), (3, 3), (4, N-1), \ldots$

One slight problem

- What is the worst case for this method?
- Consider ranges $(0, N-1), (1, 1), (2, N-1), (3, 3), (4, N-1), \ldots$
- With Q queries, each taking N time, we get $\mathcal{O}(NQ)$.
- Bucketing by singular left endpoints is too rigid.
- We need more wiggle room.

Mo's algorithm

- Lets pick a value k which is the size of a bucket.
- Now sort by $(\lceil \frac{l}{k} \rceil, r)$.
- This puts ranges with similar values of l close to each other.
- Then internally orders by r in ascending order so we can build incrementally.
- Lets analyze the worst case here.

Analysis

- If everything is in the same bucket, then we need one linear pass.
- Even distribution requires more linear passes for right endpoint.

Analysis

- If everything is in the same bucket, then we need one linear pass.
- Even distribution requires more linear passes for right endpoint.
- For the left endpoint, worst case is to wiggle maximally.
- We have $b = \frac{N}{k}$ buckets.

Analysis

- If everything is in the same bucket, then we need one linear pass.
- Even distribution requires more linear passes for right endpoint.
- For the left endpoint, worst case is to wiggle maximally.
- We have $b = \frac{N}{k}$ buckets.
- Within each bucket, right endpoint will move by N steps to the right and N steps back when switching buckets.

Analysis

- If everything is in the same bucket, then we need one linear pass.
- Even distribution requires more linear passes for right endpoint.
- For the left endpoint, worst case is to wiggle maximally.
- We have $b = \frac{N}{k}$ buckets.
- Within each bucket, right endpoint will move by N steps to the right and N steps back when switching buckets.
- Even distribution gives $\frac{Q}{b}$ queries per bucket, each query moving left endpoint by k steps.

Analysis

- If everything is in the same bucket, then we need one linear pass.
- Even distribution requires more linear passes for right endpoint.
- For the left endpoint, worst case is to wiggle maximally.
- We have $b = \frac{N}{k}$ buckets.
- Within each bucket, right endpoint will move by N steps to the right and N steps back when switching buckets.
- Even distribution gives $\frac{Q}{b}$ queries per bucket, each query moving left endpoint by k steps.
- Time complexity is $\mathcal{O}\left(\frac{N}{k} \cdot (k \cdot \frac{Q}{b} + N)\right)$.

Choosing k

- Optimal choice depends on N and Q, but consider the common case of $Q \approx N$.
- Then all we need to do is find when $k^2 = N$.
- We saw this earlier, this is $k = \sqrt{N}$

Choosing k

- Optimal choice depends on N and Q, but consider the common case of $Q \approx N$.
- Then all we need to do is find when $k^2 = N$.
- We saw this earlier, this is $k = \sqrt{N}$
- Then our total operations would be $2 \cdot N\sqrt{N}$ for the right endpoint and $\sqrt{N}^3$ for the left endpoint.
- Total is $3 \cdot N\sqrt{N}$.
- But this can be reduced to $2 \cdot N\sqrt{N}$ by reducing how much we move the right endpoint.