INF280: Competitive programming

Advanced datastructure algorithms

Louis Jachiet

Sliding windows

Sliding window techniques to improve efficiency

Typical examples using a list $i_1, \ldots, i_N$ and an integer K

- find $max_j(i_j + \cdots + i_{j+K})$

Sliding window techniques to improve efficiency

Typical examples using a list $i_1, \ldots, i_N$ and an integer K

- find $max_j(i_j + \cdots + i_{j+K})$
- find $max_j\,(max\,(i_j, \ldots, i_{j+K}) - min\,(i_j, \ldots, i_{j+K}))$

Sliding window techniques to improve efficiency

Typical examples using a list $i_1, \dots, i_N$ and an integer K

- find $max_j(i_j + \dots + i_{j+K})$
- find $max_j\,(max\,(i_j, \dots, i_{j+K}) - min\,(i_j, \dots, i_{j+K}))$
- find $max_{j,\ell}\,(\ell - j \ \mid\ max(i_j, \dots, i_\ell) - min(i_j, \dots, i_\ell) < K)$

Sliding window techniques to improve efficiency

Typical examples using a list $i_1, \dots, i_N$ and an integer K

- find $max_j(i_j + \dots + i_{j+K})$
- find $max_j\,(max\,(i_j, \dots, i_{j+K}) - min\,(i_j, \dots, i_{j+K}))$
- find $max_{j,\ell}\,(\ell - j \;\mid\; max(i_j, \dots, i_\ell) - min(i_j, \dots, i_\ell) < K)$

Sliding window techniques to improve efficiency

Typical examples using a list $i_1, \ldots, i_N$ and an integer K

- find $max_j(i_j + \cdots + i_{j+K})$
- find $max_j\,(max\,(i_j, \ldots, i_{j+K}) - min\,(i_j, \ldots, i_{j+K}))$
- find $max_{j,\ell}\,(\ell - j \ \mid\ max(i_j, \ldots, i_\ell) - min(i_j, \ldots, i_\ell) < K)$

Naive algorithm

Two (or three!) nested loops, recomputing from scratch for each position j.

Sliding window idea

Optimize away nested loops!

Example: fixed width (e.g. maintaining sum of K elements)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window idea

Optimize away nested loops!

Example: fixed width (e.g. maintaining sum of K elements)

-17

+89

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window idea

Optimize away nested loops!

Example: fixed width (e.g. maintaining sum of K elements)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Sliding window idea

Optimize away nested loops!

Example: fixed width (e.g. maintaining sum of K elements)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

Example: variable width (e.g. biggest total less than 100)

17	37	42	5	23	89	45	71	43	2	45	74	28	44	98

Sliding window techniques to improve efficiency

General idea

Maintain a double ended queue where:

- you add on the right to grow
- remove on the left to shrink
- maintain some computation over the window content

Works with monotone criteria for windows!

Sliding window techniques to improve efficiency

General idea

Maintain a double ended queue where:

- you add on the right to grow
- remove on the left to shrink
- maintain some computation over the window content

Works with monotone criteria for windows!

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 5, 23

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 5, 23, 89

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

Advanced technique for sliding window

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 5, 23, 45

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 5, 23, 45, 71

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 23, 43

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

Advanced technique for sliding window

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 2

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

Advanced technique for sliding window

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 **elements)**

Candidate mins: 2, 35

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

Candidate mins: 2, 35,74

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

Advanced technique for sliding window

The deque trick to maintain min and max

Maintain the (ordered) list of elements that might become min/max.

Example: maintaining min of 5 elements)

17	37	42	5	23	89	45	71	43	2	35	74	28	44	98

Updates

- Add on the right: remove everything bigger
- Remove on the left: remove when min

Also works with a set... with a $O(log(n))$ penalty.

Prefix sums

Structure to compute sums in $O(1)$

Input

A list of elements $v_1 ... v_n$ over a group

Query

Compute $q(i,j) = \sum_l i \leq \ell < j v_\ell$

Structure to compute sums in $O(1)$

Input

A list of elements $v_1 ... v_n$ over a group

Query

Compute $q(i,j) = \sum_l i \leq \ell < j v_\ell$

Solution

Precompute $T[i] = \sum_{\ell < i} v_i$, $q(i,j) = T[j] - T[i]$

Structure to compute sums in $O(1)$

Input

A list of elements $v_1 ... v_n$ over a group

Query

Compute $q(i,j) = \sum_l i \leq \ell < j v_\ell$

Solution

Precompute $T[i] = \sum_{\ell<i} v_i$, $q(i,j) = T[j] - T[i]$

Works in d dimensions!

Segment trees

Segment trees

Segment trees works with keys and values

- Keys needs to be ordered and are typically integers,
- values can be from any semigroup (e.g. $+$ or min over floats).

Segment trees

Segment trees works with keys and values

- Keys needs to be ordered and are typically integers,
- values can be from any semigroup (e.g. $+$ or min over floats).

A powerful structure that supports the following queries:

- Add a key k with a value v
- Add v to the value of all keys in the range $[l; r]$
- Replace with v to the value of all keys in the range $[l; r]$
- Get the sum of all values for keys in the range $[l; r]$

Segment trees, an example

Snow problem

We have a 1D representation of the snow height and the following queries:

- maximal snow height for a span $[l; r]$
- add k centimeters of snow on the span $[l; r]$
- replace the height of snow with k for the span $[l; r]$ (snow grooming)

Segment trees, an example

Snow problem

We have a 1D representation of the snow height and the following queries:

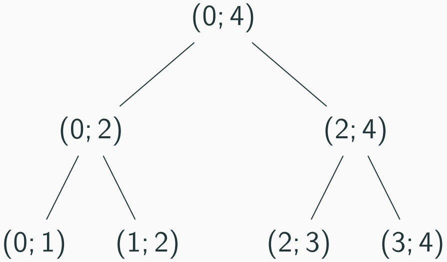
- maximal snow height for a span $[l; r]$
- add k centimeters of snow on the span $[l; r]$
- replace the height of snow with k for the span $[l; r]$ (snow grooming)

A weird semigroup:

- Add a key k with a value v
- Add v to the value of all keys in the range $[l; r]$
- Get the sum of all values for keys in the range $[l; r]$

Segment trees, the idea

Maintain a binary search tree on the keys

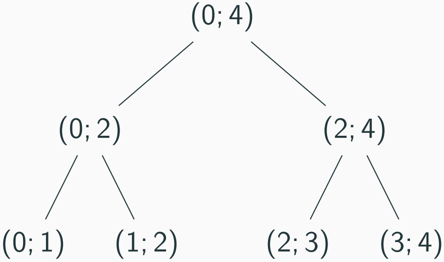


Maintain lots of information on all nodes

- the span the current node is concerned with
- the sum/max of all values from values below in the tree
- whether the values have been replaced *(lazy)*
- whether we need to add something to the children values *(lazy)*

Segment trees, the idea

Maintain a binary search tree on the keys



To update or query :

- start from the root to the leaves
- propagate any lazy value
- restore sum/max going back up

Maintain lots of information on all nodes

- the span the current node is concerned with
- the sum/max of all values from values below in the tree
- whether the values have been replaced *(lazy)*
- whether we need to add something to the children values *(lazy)*