Fitting a Step Function to a Point Set

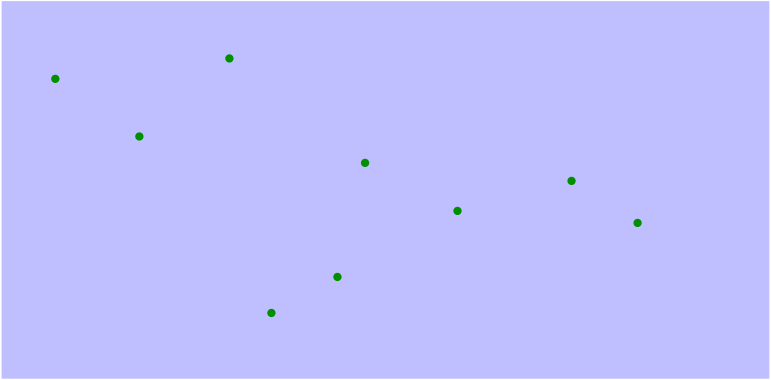
Hervé Fournier[1]   Antoine Vigneron[2]

[1]University of Versailles St-Quentin en Yvelines

[2]INRA Jouy-en-Josas

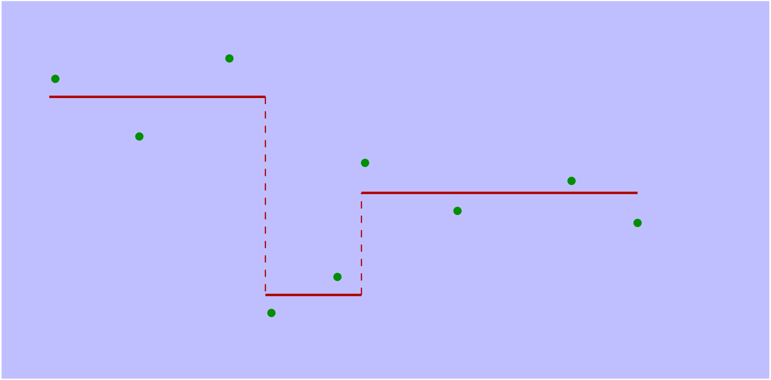
December 4, 2008

Problem statement



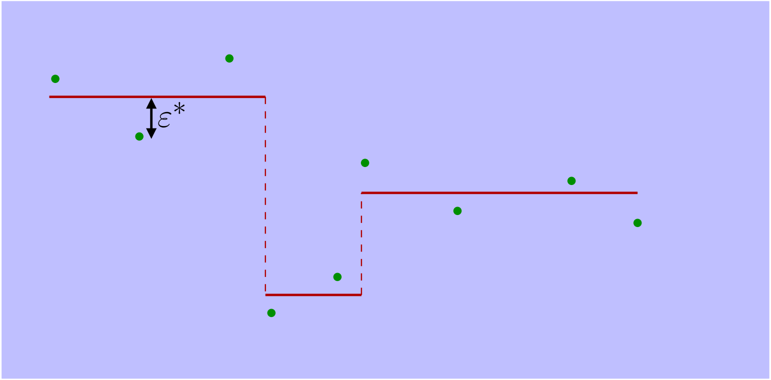
- INPUT: a set P of n points and an integer k.

Problem statement



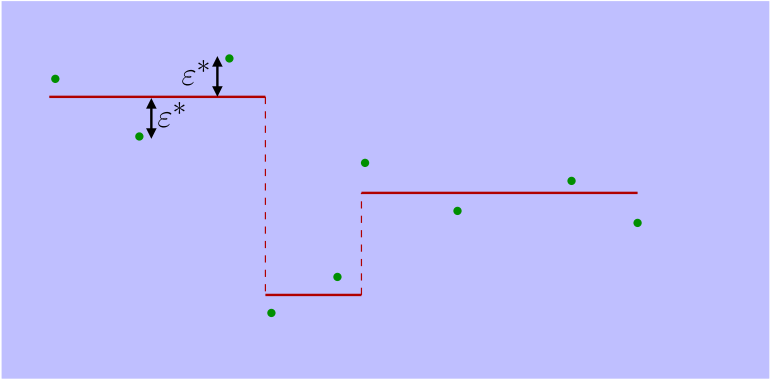
- INPUT: a set P of n points and an integer k.
- OUTPUT: a step function f^* with k steps that minimizes the maximum vertical distance $\varepsilon^* = d(P, f^*)$ between f and P.
- $d(f, P) = \max_i |f(x_i) - y_i|$ when $P = \{(x_1, y_1) \ldots (x_n, y_n)\}$

Problem statement



- INPUT: a set P of n points and an integer k.
- OUTPUT: a step function f^* with k steps that minimizes the maximum vertical distance $\varepsilon^* = d(P, f^*)$ between f and P.
- $d(f, P) = \max_i |f(x_i) - y_i|$ when $P = \{(x_1, y_1) \dots (x_n, y_n)\}$

Problem statement



- INPUT: a set P of n points and an integer k.
- OUTPUT: a step function f^* with k steps that minimizes the maximum vertical distance $\varepsilon^* = d(P, f^*)$ between f and P.
- $d(f, P) = \max_i |f(x_i) - y_i|$ when $P = \{(x_1, y_1) \dots (x_n, y_n)\}$

Our results

- An $O(n \log n)$ time algorithm.

Our results

- An $O(n \log n)$ time algorithm.
  - based on Frederickson and Johnson's sorted matrix searching technique.

Our results

- An $O(n \log n)$ time algorithm.
    - based on Frederickson and Johnson's sorted matrix searching technique.
- An $O(n)$ time algorithm for sorted input.

Our results

- An $O(n \log n)$ time algorithm.
  - based on Frederickson and Johnson's sorted matrix searching technique.
- An $O(n)$ time algorithm for sorted input.
  - based on Frederickson's path partitioning algorithm, and the data structure by Gabow, Bentley, and Tarjan for range maxima.

Our results

- An $O(n \log n)$ time algorithm.
    - based on Frederickson and Johnson's sorted matrix searching technique.
- An $O(n)$ time algorithm for sorted input.
    - based on Frederickson's path partitioning algorithm, and the data structure by Gabow, Bentley, and Tarjan for range maxima.
- An $O(n \log^4 n)$ time algorithm for the weighted case.

Our results

- An $O(n \log n)$ time algorithm.
  - based on Frederickson and Johnson's sorted matrix searching technique.
- An $O(n)$ time algorithm for sorted input.
  - based on Frederickson's path partitioning algorithm, and the data structure by Gabow, Bentley, and Tarjan for range maxima.
- An $O(n \log^4 n)$ time algorithm for the weighted case.
- An $O(n \log n \cdot h^2)$ time algorithm when h outliers are allowed.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.
- Wang (2002): $O(n^2)$.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.
- Wang (2002): $O(n^2)$.
- Mayster and Lopez (2006): $O(\min(n^2, nk \log n))$.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.
- Wang (2002): $O(n^2)$.
- Mayster and Lopez (2006): $O(\min(n^2, nk \log n))$.
- Guha and Shim (2007): $O(n + k^2 \log n)$, and $O(n \log n + k^2 \log^6 n)$ for the weighted case.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.
- Wang (2002): $O(n^2)$.
- Mayster and Lopez (2006): $O(\min(n^2, nk \log n))$.
- Guha and Shim (2007): $O(n + k^2 \log n)$, and $O(n \log n + k^2 \log^6 n)$ for the weighted case.
- Mayster and Lopez (2008): $O(n^2)$ for the weighted case.

Previous work

- Diáz-Báñez and Mesa (2001): $O(n^2 \log n)$.
- Wang (2002): $O(n^2)$.
- Mayster and Lopez (2006): $O(\min(n^2, nk \log n))$.
- Guha and Shim (2007): $O(n + k^2 \log n)$, and $O(n \log n + k^2 \log^6 n)$ for the weighted case.
- Mayster and Lopez (2008): $O(n^2)$ for the weighted case.
- In databases, the problem is known as the problem of computing a *Maximum Error Histogram*.

Decision algorithm

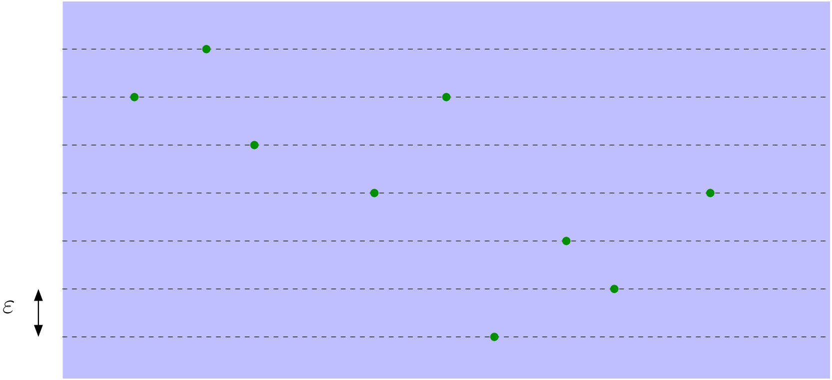
- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?

Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:

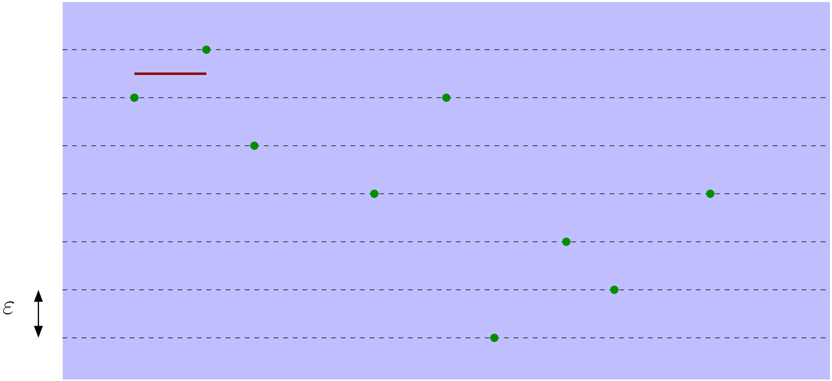
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



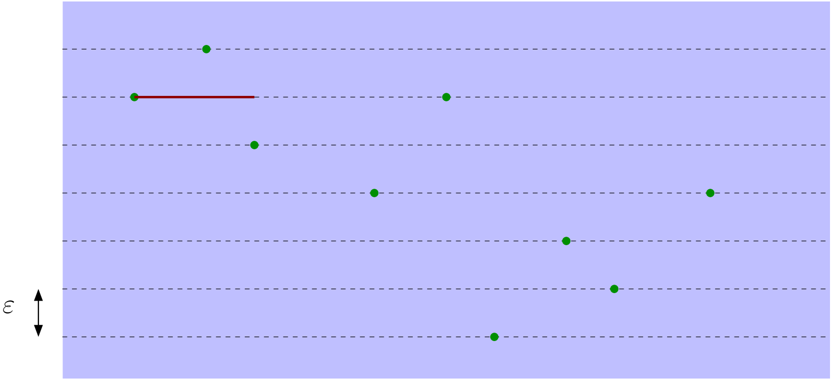
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



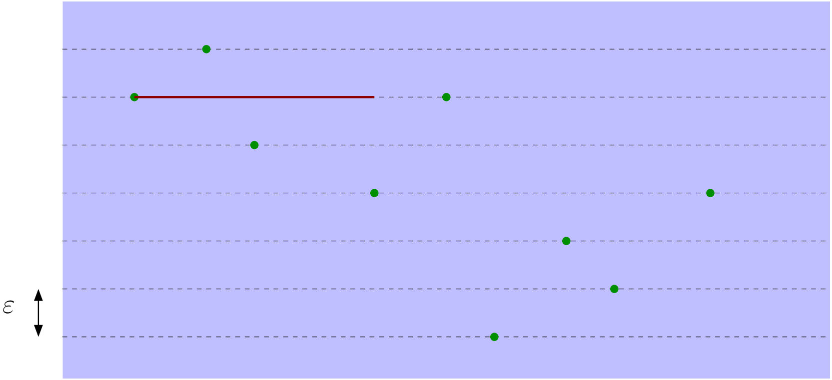
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



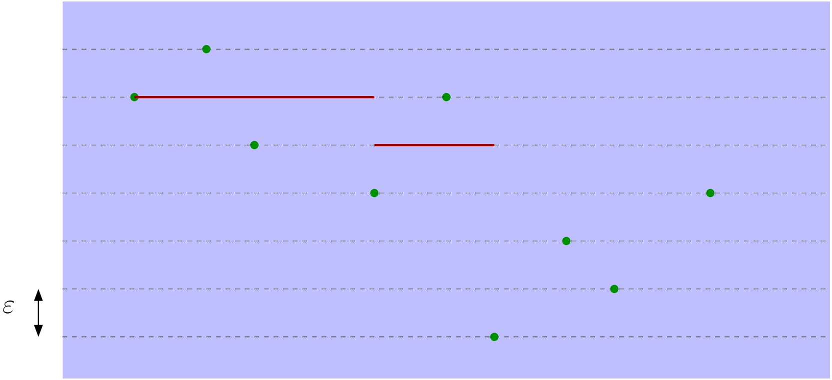
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



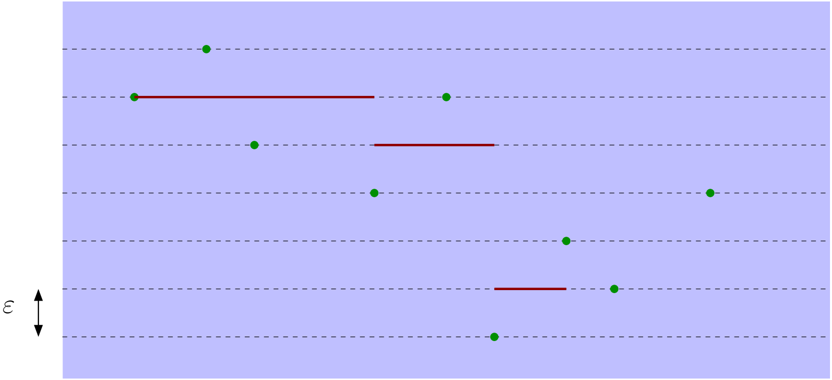
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



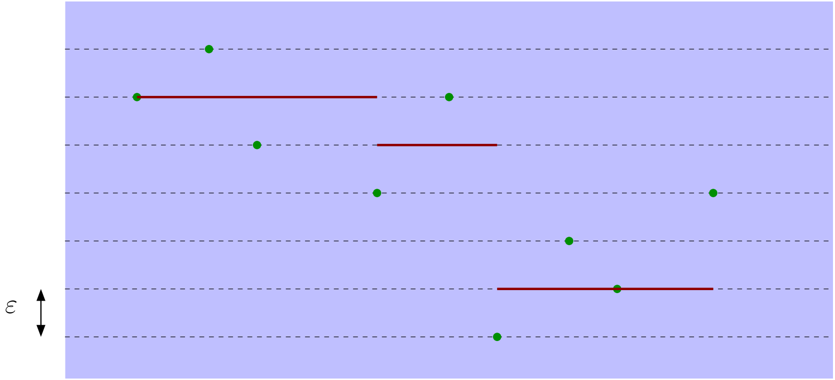
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



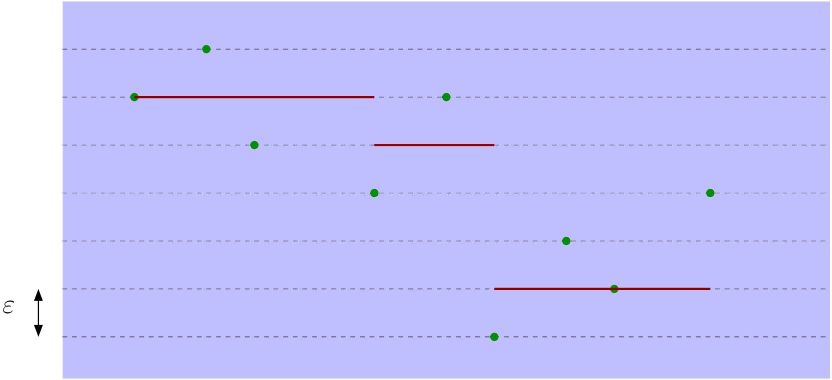
Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



Decision algorithm

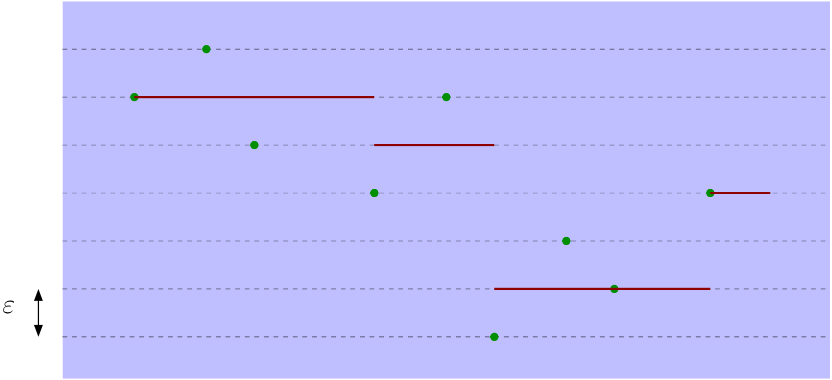
- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



- FALSE if $k = 3$

Decision algorithm

- Decision problem: given P, k, $\varepsilon > 0$, is there a k-step function f such that $d(P, f) \leqslant \varepsilon$?
- Simple $O(n)$ time greedy algorithm by Diáz-Báñez and Mesa:



- FALSE if $k = 3$
- TRUE if $k = 4$

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
  - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
    - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.
    - Perform binary search using the decision algorithm: $O(n \log n)$.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
  - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.
  - Perform binary search using the decision algorithm: $O(n \log n)$.
- Can be improved to $O(n \log n)$ using Frederickson and Johnson's sorted matrix searching technique.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
  - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.
  - Perform binary search using the decision algorithm: $O(n \log n)$.
- Can be improved to $O(n \log n)$ using Frederickson and Johnson's sorted matrix searching technique.
- Let $\tilde{y}_1 \leqslant \ldots \leqslant \tilde{y}_n$ denote the y-coordinates in sorted order.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
  - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.
  - Perform binary search using the decision algorithm: $O(n \log n)$.
- Can be improved to $O(n \log n)$ using Frederickson and Johnson's sorted matrix searching technique.
- Let $\tilde{y}_1 \leqslant \ldots \leqslant \tilde{y}_n$ denote the y-coordinates in sorted order.
- Let $M_{ij} = \frac{1}{2}\left(\tilde{y}_i - \tilde{y}_{n+1-j}\right)$.

Optimization algorithm

- $\{y_1, \ldots, y_n\}$ denote the y-coordinates of points in P.
- Let $\varepsilon_{ij} = \frac{1}{2}(y_i - y_j)$.
- Then $\varepsilon^* = \varepsilon_{ij}$ for some i, j.
- Simple algorithm:
    - Sort the ε_{ij}'s: $O(n^2 \log n)$ time.
    - Perform binary search using the decision algorithm: $O(n \log n)$.
- Can be improved to $O(n \log n)$ using Frederickson and Johnson's sorted matrix searching technique.
- Let $\tilde{y}_1 \leqslant \ldots \leqslant \tilde{y}_n$ denote the y-coordinates in sorted order.
- Let $M_{ij} = \frac{1}{2}\left(\tilde{y}_i - \tilde{y}_{n+1-j}\right)$.
- M is a *sorted matrix*: $i \leqslant i'$ and $j \leqslant j'$ implies $M_{ij} \leqslant M_{i'j'}$.

Searching in a sorted matrix

$$\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{matrix}$$

- a sorted matrix M

Searching in a sorted matrix

$$\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{matrix}$$

- a sorted matrix M
- an increasing boolean function g: there exists x^* such that $g(x) = \mathit{FALSE}$ for all $x < x^*$, and $g(x) = \mathit{TRUE}$ for all $x \geqslant x^*$.

Searching in a sorted matrix

$$\begin{matrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{matrix}$$

- a sorted matrix M
- an increasing boolean function g: there exists x^* such that $g(x) = \mathit{FALSE}$ for all $x < x^*$, and $g(x) = \mathit{TRUE}$ for all $x \geqslant x^*$.
- Problem: search for x^* in M.

Searching in a sorted matrix

$$
\begin{array}{cccc|cccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}
$$

- smallest elements in submatrices: $\{1, 5, 6, 12\}$; median=6.

Searching in a sorted matrix

$$
\begin{array}{cccc|cccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}
$$

- smallest elements in submatrices: $\{1, 5, 6, 12\}$; median=6.
- largest elements in submatrices: $\{9, 15, 16, 23\}$; median=16.

Searching in a sorted matrix

$$\begin{array}{cccc|cccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}$$

- smallest elements in submatrices: $\{1, 5, 6, 12\}$; median=6.
- largest elements in submatrices: $\{9, 15, 16, 23\}$; median=16.
- we compute $g(6) = \mathit{FALSE}$ and $g(16) = \mathit{TRUE}$.

Searching in a sorted matrix

$$\begin{array}{cccc|cccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}$$

- smallest elements in submatrices: $\{1, 5, 6, 12\}$; median=6.
- largest elements in submatrices: $\{9, 15, 16, 23\}$; median=16.
- we compute $g(6) = \mathit{FALSE}$ and $g(16) = \mathit{TRUE}$.
- we did not make progress.

Searching in a sorted matrix

$$\begin{array}{cc|cc|cc|cc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
\hline
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}$$

- smallest elements: $\{1, 3, 5, 6, 7, 8, 10, 12, 13, 15, 19\}$; median$=8$.

Searching in a sorted matrix

$$
\begin{array}{cc|cc|cc|cc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
\hline
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}
$$

- smallest elements: $\{1, 3, 5, 6, 7, 8, 10, 12, 13, 15, 19\}$; median=8.
- largest elements: $\{3, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 23\}$; median=12.

Searching in a sorted matrix

$$\begin{array}{cc|cc|cc|cc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
\hline
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}$$

- smallest elements: $\{1, 3, 5, 6, 7, 8, 10, 12, 13, 15, 19\}$; median=8.
- largest elements: $\{3, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 23\}$; median=12.
- we compute $g(8) = \mathit{FALSE}$ and $g(12) = \mathit{TRUE}$.

Searching in a sorted matrix

$$
\begin{array}{cc|cc|cc|cc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
2 & 3 & 5 & 6 & 7 & 8 & 9 & 10 \\
\hline
3 & 6 & 7 & 8 & 10 & 11 & 12 & 13 \\
4 & 7 & 8 & 9 & 11 & 12 & 13 & 15 \\
\hline
6 & 9 & 10 & 11 & 12 & 14 & 15 & 17 \\
7 & 10 & 11 & 13 & 14 & 16 & 17 & 18 \\
\hline
8 & 12 & 13 & 14 & 15 & 18 & 19 & 20 \\
9 & 13 & 14 & 16 & 17 & 19 & 21 & 23
\end{array}
$$

- smallest elements: $\{1, 3, 5, 6, 7, 8, 10, 12, 13, 15, 19\}$; median=8.
- largest elements: $\{3, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 23\}$; median=12.
- we compute $g(8) = \mathit{FALSE}$ and $g(12) = \mathit{TRUE}$.
- we can discard 8 submatrices.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	13
		8	9	11	12	13	15
6	9	10	11	12	14		
7	10	11	13	14	16		
8	12						
9	13						

- smallest elements: $\{1, 3, 5, 6, 7, 8, 10, 12, 13, 15, 19\}$; median=8.
- largest elements: $\{3, 6, 7, 8, 9, 10, 12, 13, 15, 16, 18, 19, 23\}$; median=12.
- we compute $g(8) = \mathit{FALSE}$ and $g(12) = \mathit{TRUE}$.
- we can discard 8 submatrices.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	13
		8	9	11	12	13	15
6	9	10	11	12	14		
7	10	11	13	14	16		
8	12						
9	13						

- remaining elements $\{7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$; median=12.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	13
		8	9	11	12	13	15
6	9	10	11	12	14		
7	10	11	13	14	16		
8	12						
9	13						

- remaining elements $\{7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$; median=12.
- we compute $g(12) = \mathit{TRUE}$

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	13
		8	9	11	12	13	15
6	9	10	11	12	14		
7	10	11	13	14	16		
8	12						
9	13						

- remaining elements $\{7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$; median=12.
- we compute $g(12) = TRUE$
- we discard all elements > 12.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	
		8	9	11	12		
6	9	10	11	12			
7	10	11					
8	12						
9							

- repeat until one value is left.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	
		8	9	11	12		
6	9	10	11	12			
7	10	11					
8	12						
9							

- repeat until one value is left.
- $O(\log n)$ calls to the decision algorithm

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	
		8	9	11	12		
6	9	10	11	12			
7	10	11					
8	12						
9							

- repeat until one value is left.
- $O(\log n)$ calls to the decision algorithm
- $O(n \log n)$ time for the rest of the algorithm, assuming each matrix element can be accessed in $O(1)$ time.

Searching in a sorted matrix

						7	8
						9	10
		7	8	10	11	12	
		8	9	11	12		
6	9	10	11	12			
7	10	11					
8	12						
9							

- repeat until one value is left.
- $O(\log n)$ calls to the decision algorithm
- $O(n \log n)$ time for the rest of the algorithm, assuming each matrix element can be accessed in $O(1)$ time.
- Therefore, we can compute an optimal step function in $O(n \log n)$ time.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.
- Example: weights $\{3, 1, 4, 3, 2, 4, 1\}$, $k = 3$.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.
- Example: weights $\{3, 1, 4, 3, 2, 4, 1\}$, $k = 3$.
- Answer: $\{3, 1\}$, $\{4, 3\}$, $\{2, 4, 1\}$. Max weight=7.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.
- Example: weights $\{3, 1, 4, 3, 2, 4, 1\}$, $k = 3$.
- Answer: $\{3, 1\}$, $\{4, 3\}$, $\{2, 4, 1\}$. Max weight=7.
- The path partitioning problem can be solved in $O(n \log n)$ time by sorted matrix searching.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.
- Example: weights $\{3, 1, 4, 3, 2, 4, 1\}$, $k = 3$.
- Answer: $\{3, 1\}$, $\{4, 3\}$, $\{2, 4, 1\}$. Max weight=7.
- The path partitioning problem can be solved in $O(n \log n)$ time by sorted matrix searching.
- Frederickson found an optimal $O(n)$ time algorithm for path partitioning.

Path partitioning

- INPUT: a path P with n weighted nodes, and an integer k
- OUTPUT: a partition of P into k subpaths that minimizes the maximum weight of the subpaths.
- Example: weights $\{3, 1, 4, 3, 2, 4, 1\}$, $k = 3$.
- Answer: $\{3, 1\}$, $\{4, 3\}$, $\{2, 4, 1\}$. Max weight=7.
- The path partitioning problem can be solved in $O(n \log n)$ time by sorted matrix searching.
- Frederickson found an optimal $O(n)$ time algorithm for path partitioning.
- This algorithm works in the following, more general case:

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \to \mathbb{R}^+$ with $\theta(e) = 0$

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \rightarrow \mathbb{R}^+$ with $\theta(e) = 0$
- $\textrm{MIN-MAX PARTITION}(\theta)$ problem:

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \rightarrow \mathbb{R}^+$ with $\theta(e) = 0$
- $\text{MIN-MAX PARTITION}(\theta)$ problem:
  - Given $w \in \Sigma^*$ and $k > 0$, compute a factorization $w = w_1 w_2 \dots w_k$ minimizing

$$\max_{i \in \{1,\dots,k\}} \theta(w_i)$$

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \to \mathbb{R}^+$ with $\theta(e) = 0$
- $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ problem:
  - Given $w \in \Sigma^*$ and $k > 0$, compute a factorization $w = w_1 w_2 \ldots w_k$ minimizing

$$\max_{i \in \{1,\ldots,k\}} \theta(w_i)$$

- Frederickson's problem is obtained with:

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \to \mathbb{R}^+$ with $\theta(e) = 0$
- MIN-MAX PARTITION(θ) problem:
  - Given $w \in \Sigma^*$ and $k > 0$, compute a factorization $w = w_1 w_2 \ldots w_k$ minimizing

$$\max_{i \in \{1,\ldots,k\}} \theta(w_i)$$

- Frederickson's problem is obtained with:
  - $\Sigma = \mathbb{R}^+$

General framework

- Σ an alphabet (e.g. $\Sigma = \{a, b\}$ or $\Sigma = \mathbb{R}$)
- θ: $\Sigma^* \to \mathbb{R}^+$ with $\theta(e) = 0$
- $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ problem:
  - Given $w \in \Sigma^*$ and $k > 0$, compute a factorization $w = w_1 w_2 \ldots w_k$ minimizing
$$\max_{i \in \{1,\ldots,k\}} \theta(w_i)$$
- Frederickson's problem is obtained with:
  - $\Sigma = \mathbb{R}^+$
  - $\theta(a_1 \ldots a_p) = a_1 + \ldots + a_p$

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \rightarrow \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \rightarrow \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \to \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \ldots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \ldots a_j)$ in time $\kappa(n)$.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \to \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \dots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \dots a_j)$ in time $\kappa(n)$.
- Then $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ can be solved in time $O(\pi(n) + n\kappa(n))$.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \rightarrow \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \dots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \dots a_j)$ in time $\kappa(n)$.
- Then $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ can be solved in time $O(\pi(n) + n\kappa(n))$.
- Example: path partitioning.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \rightarrow \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \dots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \dots a_j)$ in time $\kappa(n)$.
- Then $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ can be solved in time $O(\pi(n) + n\kappa(n))$.
- Example: path partitioning.
  - Preprocessing: compute $S_i = \sum_{\ell=1}^{i} \omega_\ell$ for all i.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \to \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \dots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \dots a_j)$ in time $\kappa(n)$.
- Then $\text{MIN-MAX PARTITION}(\theta)$ can be solved in time $O(\pi(n) + n\kappa(n))$.
- Example: path partitioning.
  - Preprocessing: compute $S_i = \sum_{\ell=1}^{i} \omega_\ell$ for all i.
  - Query: $(i, j) \mapsto \theta(\omega_i, \dots, \omega_j) = \sum_{\ell=i}^{j} \omega_\ell = S_j - S_{i-1}$.

General framework

- Let Σ be an alphabet, and $\theta : \Sigma^* \to \mathbb{R}^+$ be a mapping such that $\theta(e) = 0$. Suppose that θ has the following properties:
  - (i) θ is non-decreasing, that is, $\theta(v) \leqslant \theta(uvw)$ for all $u, v, w \in \Sigma^*$.
  - (ii) We can preprocess $a_1 \dots a_n \in \Sigma^n$ in time $\pi(n)$ so that, given any query (i, j), we can compute $\theta(a_i \dots a_j)$ in time $\kappa(n)$.
- Then $\mathrm{MIN\text{-}MAX\ PARTITION}(\theta)$ can be solved in time $O(\pi(n) + n\kappa(n))$.
- Example: path partitioning.
  - Preprocessing: compute $S_i = \sum_{\ell=1}^{i} \omega_\ell$ for all i.
  - Query: $(i, j) \mapsto \theta(\omega_i, \dots, \omega_j) = \sum_{\ell=i}^{j} \omega_\ell = S_j - S_{i-1}$.
  - $\pi(n) = O(n)$ and $\kappa(n) = O(1)$, so running time $O(n)$.

Linear-time algorithm for fitting a step function

- Application of previous theorem with:

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$
- $\theta(a_1 \ldots a_p) = \frac{1}{2}\left(\max(a_1, \ldots, a_p) - \min(a_1, \ldots, a_p)\right)$

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$
- $\theta(a_1 \dots a_p) = \frac{1}{2}\left(\max(a_1, \dots, a_p) - \min(a_1, \dots, a_p)\right)$
- Range maxima problem:

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$
- $\theta(a_1 \dots a_p) = \frac{1}{2}\left(\max(a_1, \dots, a_p) - \min(a_1, \dots, a_p)\right)$
- Range maxima problem:
  - Preprocess a sequence of numbers $(y_1, \dots, y_n)$ to allow efficient answer to query
$$(i, j) \mapsto \max(y_i, \dots, y_j)$$

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$
- $\theta(a_1 \dots a_p) = \frac{1}{2}(\max(a_1, \dots, a_p) - \min(a_1, \dots, a_p))$
- Range maxima problem:
  - Preprocess a sequence of numbers $(y_1, \dots, y_n)$ to allow efficient answer to query
    $$(i, j) \mapsto \max(y_i, \dots, y_j)$$
  - Gabow, Bentley, and Tarjan (1984): $O(1)$ query time and $O(n)$ time preprocessing.

Linear-time algorithm for fitting a step function

- Application of previous theorem with:
- $\Sigma = \mathbb{R}$
- $\theta(a_1 \ldots a_p) = \frac{1}{2}\left(\max(a_1, \ldots, a_p) - \min(a_1, \ldots, a_p)\right)$
- Range maxima problem:
  - Preprocess a sequence of numbers $(y_1, \ldots, y_n)$ to allow efficient answer to query

    $$(i, j) \mapsto \max(y_i, \ldots, y_j)$$

  - Gabow, Bentley, and Tarjan (1984): $O(1)$ query time and $O(n)$ time preprocessing.
- Conclusion: the sorted case can be solved in $O(n)$ time.

Weighted version

- Given a set of points in the plane $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ with positive weights $\mu_1, \ldots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that

  $$\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$$

  is minimized.

Weighted version

- Given a set of points in the plane $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ with positive weights $\mu_1, \ldots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that

  $$\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$$

  is minimized.

Weighted version

- Given a set of points in the plane $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ with positive weights $\mu_1, \ldots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that

  $$\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$$

  is minimized.
- Data structure by Guha and Shim: preprocessing $\pi(n) = O(n \log n)$ and query time $\kappa(n) = O(\log^4 n)$.

Weighted version

- Given a set of points in the plane $\{(x_1, y_1), \dots, (x_n, y_n)\}$ with positive weights $\mu_1, \dots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that
$$\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$$
is minimized.
- Data structure by Guha and Shim: preprocessing $\pi(n) = O(n \log n)$ and query time $\kappa(n) = O(\log^4 n)$.
- So we obtain an $O(n \log^4 n)$ time algorithm.

Weighted version

- Given a set of points in the plane $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ with positive weights $\mu_1, \ldots, \mu_n$ and an integer $k > 0$, compute a k-step function f such that
$$\max_{1 \leqslant i \leqslant n} \mu_i |f(x_i) - y_i|$$
is minimized.
- Data structure by Guha and Shim: preprocessing $\pi(n) = O(n \log n)$ and query time $\kappa(n) = O(\log^4 n)$.
- So we obtain an $O(n \log^4 n)$ time algorithm.
- Guha and Shim gave an $O(n \log n + k^2 \log^6 n)$ time algorithm.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \dots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.
- using this information, we do some preprocessing on each subinterval that is not associated with any remaining matrix element.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.
- using this information, we do some preprocessing on each subinterval that is not associated with any remaining matrix element.
- it gives us an $O(n \log \log n / \log n)$ time decision algorithm.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.
- using this information, we do some preprocessing on each subinterval that is not associated with any remaining matrix element.
- it gives us an $O(n \log \log n / \log n)$ time decision algorithm.
- using it to search the whole matrix, we get an $O(n \log \log n)$ optimization algorithm.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \dots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.
- using this information, we do some preprocessing on each subinterval that is not associated with any remaining matrix element.
- it gives us an $O(n \log \log n / \log n)$ time decision algorithm.
- using it to search the whole matrix, we get an $O(n \log \log n)$ optimization algorithm.
- It can be improved to $O(n \log^* n)$ by cutting the factors recursively.

Frederickson's algorithm (sketch)

- Partition $a_1 a_2 \ldots a_n$ into n/r factors of length r, for $r = \lceil \log n \rceil$.
- construct a collection of submatrices, each submatrix corresponding to one factor.
- apply the matrix searching technique on this collection of submatrices, until $O(n/r^2)$ elements remain.
- using this information, we do some preprocessing on each subinterval that is not associated with any remaining matrix element.
- it gives us an $O(n \log \log n / \log n)$ time decision algorithm.
- using it to search the whole matrix, we get an $O(n \log \log n)$ optimization algorithm.
- It can be improved to $O(n \log^* n)$ by cutting the factors recursively.
- It can be further improved to $O(n)$, using careful counting arguments.

Conclusion

- Our algorithms for the unweighted cases (sorted and unsorted) are optimal.

Conclusion

- Our algorithms for the unweighted cases (sorted and unsorted) are optimal.
- In the weighted case, there may be room for improvement.

Conclusion

- Our algorithms for the unweighted cases (sorted and unsorted) are optimal.
- In the weighted case, there may be room for improvement.
- Are there other applications of the general formulation of Frederickson's linear time algorithm?

Conclusion

- Our algorithms for the unweighted cases (sorted and unsorted) are optimal.
- In the weighted case, there may be room for improvement.
- Are there other applications of the general formulation of Frederickson's linear time algorithm?
    - Is there a simple randomized algorithm?

Conclusion

- Our algorithms for the unweighted cases (sorted and unsorted) are optimal.
- In the weighted case, there may be room for improvement.
- Are there other applications of the general formulation of Frederickson's linear time algorithm?
  - Is there a simple randomized algorithm?
- L_1-problem: minimize the sum of the vertical distances between P and f.