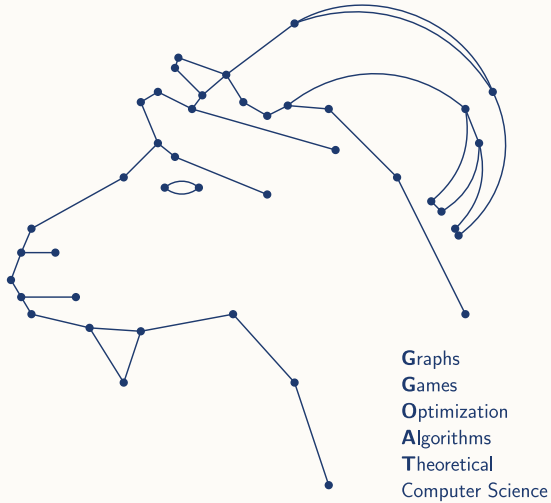
From Marcus–Tardos to optimal algorithms

Michal Opler

Czech Technical University in Prague

AofA 2026

Joint work with László Kozma (TU Dresden)



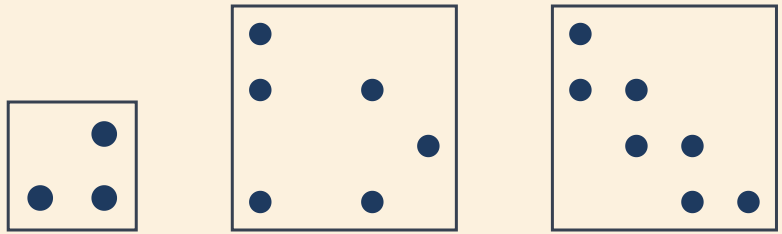
oplermic.pages.fit/talks/aofa26

Introduction

Extremal theory of binary matrices

Definition

- A 0-1 matrix M **contains** a matrix P if P can be obtained from M by deleting rows and columns and turning some 1-entries into 0-entries. Otherwise, M **avoids** P.

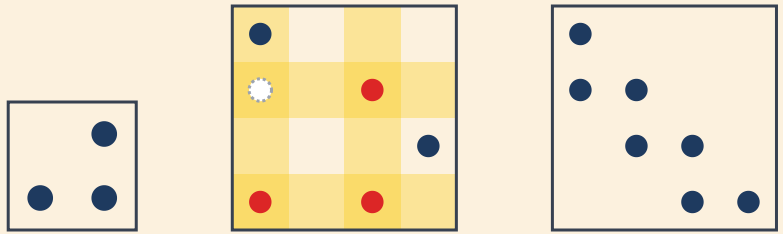


Pattern P M contains P M' avoids P

- $\mathrm{ex}(P, n)$ $\leftarrow$ the maximum number of 1-entries in a P-avoiding $n \times n$ 0-1 matrix.

Definition

- A 0-1 matrix M **contains** a matrix P if P can be obtained from M by deleting rows and columns and turning some 1-entries into 0-entries. Otherwise, M **avoids** P.



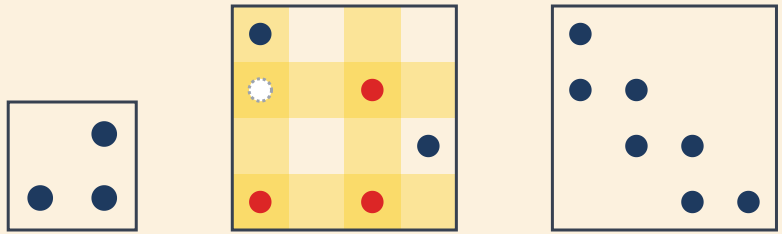
Pattern P M contains P M' avoids P

- $\mathrm{ex}(P, n)$ $\leftarrow$ the maximum number of 1-entries in a P-avoiding $n \times n$ 0-1 matrix.

Extremal theory of binary matrices

Definition

- A 0-1 matrix M **contains** a matrix P if P can be obtained from M by deleting rows and columns and turning some 1-entries into 0-entries. Otherwise, M **avoids** P.



Pattern P M contains P M' avoids P

- $\mathrm{ex}(P, n) \leftarrow$ the maximum number of 1-entries in a P-avoiding $n \times n$ 0-1 matrix.

- $\mathrm{ex}\left(\begin{pmatrix} \bullet & \bullet \\ \bullet & \bullet \end{pmatrix}, n\right) = \Theta(n^{3/2})$ *(Kővári et al., 1954)*
- $\mathrm{ex}\left(\begin{pmatrix} \bullet & \cdot & \bullet & \cdot \end{pmatrix}, n\right) = \Theta(n \cdot \alpha(n))$ *(Füredi & Hajnal, 1992)*

- for every $t \geq 2$, there is P_t with $\mathrm{ex}(P_t, n) = \Theta(n(\log n / \log\log n)^t)$ *(Pettie & Tardos, 2025)*

Marcus-Tardos theorem

- A **permutation** π is a sequence $\pi_1, \ldots, \pi_n$ of distinct values from $[n]$.
- For a permutation π, let P_π denote its permutation matrix.
- We write $\mathrm{ex}_\pi(n) := \mathrm{ex}(P_\pi, n)$ for short.

$\pi = 3142$ $\quad P_\pi =$ 

Theorem ***(Füredi-Hajnal conjecture** · Marcus, Tardos, 2004**)***

$\mathrm{ex}_\pi(n) \in \mathcal{O}_\pi(n)$ for every permutation π.

Moreover, $c_\pi := \lim_{n\to\infty} \frac{1}{n}\mathrm{ex}_\pi(n)$ exists and $\mathrm{ex}_\pi(n) \le c_\pi \cdot n$ for all n. *(Cibulka, 2009)*

Füredi–Hajnal limit of π

Marcus-Tardos theorem

- A **permutation** π is a sequence $\pi_1, \ldots, \pi_n$ of distinct values from $[n]$.
- For a permutation π, let P_π denote its permutation matrix.
- We write $\mathrm{ex}_\pi(n) := \mathrm{ex}(P_\pi, n)$ for short.

$\pi = 3142$ $\quad P_\pi =$

Theorem ***(Füredi-Hajnal conjecture** · Marcus, Tardos, 2004**)***

$\mathrm{ex}_\pi(n) \in \mathcal{O}_\pi(n)$ for every permutation π.

Moreover, $c_\pi := \lim_{n\to\infty} \frac{1}{n}\mathrm{ex}_\pi(n)$ exists and $\mathrm{ex}_\pi(n) \le c_\pi \cdot n$ for all n. *(Cibulka, 2009)*

Füredi–Hajnal limit of π

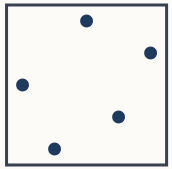
Corollary

Every $m \times n$ 0-1 matrix with strictly more than $c_\pi \cdot \max(m, n)$ 1-entries contains π.

Pattern-avoiding permutations

- A permutation σ **contains** a pattern π if it has a subsequence order-isomorphic to π.
- This coincides with containment on the matrices P_σ and P_π.
- Otherwise, σ **avoids** π.

$\pi = 132$



$\sigma = 31524$

Theorem ***(Stanley–Wilf conjecture*** · *Klazar, 2000* · *Marcus, Tardos, 2004)*

For every pattern π, the number of π-avoiding permutations of length n is $2^{\mathcal{O}_\pi(n)}$.

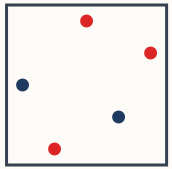
Moreover, the limit $s_\pi := \lim_{n\to\infty} \sqrt[n]{|\{\sigma \in \mathcal{S}_n \mid \sigma \text{ avoids } \pi\}|}$ exists *(Arratia, 1999)*.

Stanley–Wilf limit of π

Pattern-avoiding permutations

- A permutation σ **contains** a pattern π if it has a subsequence order-isomorphic to π.
- This coincides with containment on the matrices P_σ and P_π.
- Otherwise, σ **avoids** π.

$\pi = 132$



$\sigma = 31524$

Theorem ***(Stanley–Wilf conjecture*** · *Klazar, 2000* · *Marcus, Tardos, 2004)*

For every pattern π, the number of π-avoiding permutations of length n is $2^{\mathcal{O}_\pi(n)}$.

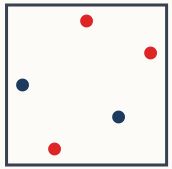
Moreover, the limit $s_\pi := \lim_{n\to\infty} \sqrt[n]{|\{\sigma \in \mathcal{S}_n \mid \sigma \text{ avoids } \pi\}|}$ exists *(Arratia, 1999)*.

Stanley–Wilf limit of π

Pattern-avoiding permutations

- A permutation σ **contains** a pattern π if it has a subsequence order-isomorphic to π.
- This coincides with containment on the matrices P_σ and P_π.
- Otherwise, σ **avoids** π.

$\pi = 132$



$\sigma = 31524$

Theorem ***(Stanley–Wilf conjecture*** · *Klazar, 2000* · *Marcus, Tardos, 2004)*

For every pattern π, the number of π-avoiding permutations of length n is $2^{\mathcal{O}_\pi(n)}$.

Moreover, the limit $s_\pi := \lim_{n\to\infty} \sqrt[n]{|\{\sigma \in \mathcal{S}_n \mid \sigma \text{ avoids } \pi\}|}$ exists *(Arratia, 1999)*.

Stanley–Wilf limit of π

Theorem ***(Cibulka, 2009)***

For every pattern π, we have $c_\pi = \Omega(s_\pi^{4.5})$ and $s_\pi = \mathcal{O}(c_\pi^2)$, and therefore $\log s_\pi = \Theta(\log c_\pi)$.

Specific patterns

Example

- $(k \cdots 1)$-avoiding – union of $k-1$ increasing sequences: $c_\pi = 2(k-1)$ *(Cibulka, 2009)*, $s_\pi = (k-1)^2$
- 231-avoiding – stack-sortable *(Knuth, 1968)*: $s_\pi = 4$
- separable – built from direct/skew sums, avoids 2413 and 3142: $s_\pi = 3 + 2\sqrt{2}$ *(West, 1995)*
- 1324-avoiding: $10.271 \leq s_\pi \leq 13.5$ *(Bevan et al., 2020)*, conjectured $s_\pi \approx 11.60$ *(Conway & Guttmann, 2015)*

Specific patterns

Example

- $(k \cdots 1)$-avoiding – union of $k-1$ increasing sequences: $c_\pi = 2(k-1)$ *(Cibulka, 2009)*, $s_\pi = (k-1)^2$
- 231-avoiding – stack-sortable *(Knuth, 1968)*: $s_\pi = 4$
- separable – built from direct/skew sums, avoids 2413 and 3142: $s_\pi = 3 + 2\sqrt{2}$ *(West, 1995)*
- 1324-avoiding: $10.271 \leq s_\pi \leq 13.5$ *(Bevan et al., 2020)*, conjectured $s_\pi \approx 11.60$ *(Conway & Guttmann, 2015)*

The general behavior is exponential in the pattern length:

Theorem *(Fox, 2013)*

For every pattern π of length k, $c_\pi, s_\pi = 2^{\mathcal{O}(k)}$; and for almost all patterns π of length k, $s_\pi = 2^{\Omega((k/\log k)^{1/4})}$.

The asymptotics of Stanley-Wilf limits are poorly understood – e.g., when is s_π polynomial in length of π?

Beyond-worst case complexity

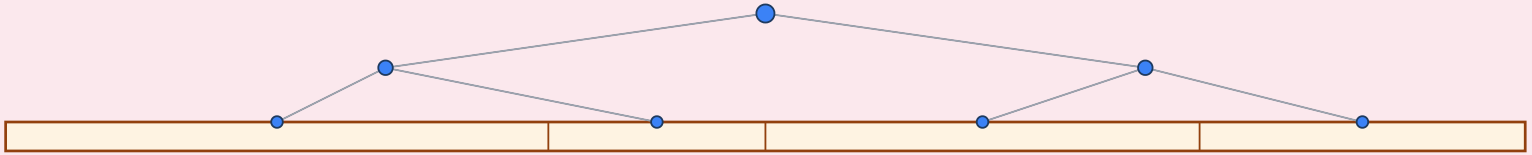
- **Worst-case analysis**: worst-case governed by the hardest instances, no additional guarantees on easier instances.
- **Average-case analysis**: expectation for given distribution on all instances, can still be very inefficient on easy instances.
- **Adaptivity**: we have a measure of difficulty of individual instances (usually accompanied by a lower bound) and the runtime should obey this measure.

Beyond-worst case complexity

- **Worst-case analysis**: worst-case governed by the hardest instances, no additional guarantees on easier instances.
- **Average-case analysis**: expectation for given distribution on all instances, can still be very inefficient on easy instances.
- **Adaptivity**: we have a measure of difficulty of individual instances (usually accompanied by a lower bound) and the runtime should obey this measure.

Example ***(Sorting)***

- $\Theta(n \log n)$ comparisons in the worst-case and there are many unrelated algorithms that match this bound.
- $\Theta(n \log n)$ comparisons even in expectation with uniform distribution.
- Possible measures of presortedness – number of inversions, number of runs, deletion distance to sorted sequence, …
- For example, permutation with k runs can be sorted in $\mathcal{O}(\log k \cdot n)$ time and there is a matching lower bound. Moreover, there are algorithms that also adapt to the entropy of run lengths like *PowerSort* used in Python *(Munro & Wild, 2018)*.



Sorting pattern-avoiding permutations

Lower bound

Theorem

Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons in the worst case to sort n elements.

Lower bound

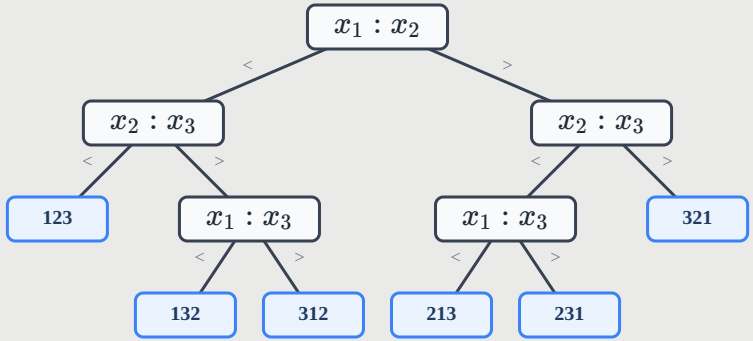
Theorem

Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons in the worst case to sort n elements.

Proof

Model the algorithm as a binary decision tree: each internal node is a comparison and each leaf is a permutation.

- There are $n!$ possible inputs, and the algorithm must reach a different leaf for each one $\rightarrow$ the tree has at least $n!$ leaves.
- A binary tree of depth d has at most 2^d leaves, so $2^d \geq n!$, i.e. $d \geq \log_2(n!) = \Omega(n \log n)$ by Stirling's approximation.
- depth of the tree = worst-case number of comparisons. ■



Decision tree of depth 3 for n=3.

Lower bound

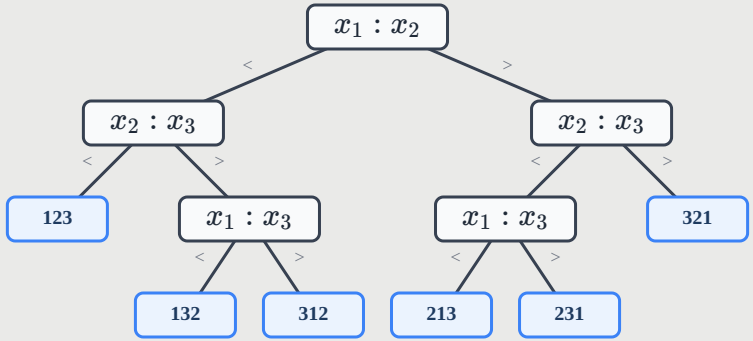
Theorem

Any deterministic comparison-based sorting algorithm must perform $\Omega(n \log n)$ comparisons in the worst case to sort n elements.

Proof

Model the algorithm as a binary decision tree: each internal node is a comparison and each leaf is a permutation.

- There are $n!$ possible inputs, and the algorithm must reach a different leaf for each one $\rightarrow$ the tree has at least $n!$ leaves.
- A binary tree of depth d has at most 2^d leaves, so $2^d \geq n!$, i.e. $d \geq \log_2(n!) = \Omega(n \log n)$ by Stirling's approximation.
- depth of the tree = worst-case number of comparisons. ■



Decision tree of depth 3 for n=3.

Applied to π-avoiding permutations, we get lower bound $\Omega\left(\log\left(s_\pi^n\right)\right) = \Omega(\log s_\pi \cdot n)$!

Previous results

Theorem (*Fredman, 1976*)

For any set Γ of permutations of length n, there exists a decision tree of depth $\log_2 |\Gamma| + 2n$ that sorts every input from Γ.

Pattern-specific approach:

- k-increasing in $\mathcal{O}(\log k \cdot n)$ time
- 231-avoiding in $\mathcal{O}(n)$ time *(Knuth, 1968)*
- 1234-, 1243- and 2143-avoiding in $\mathcal{O}(n)$ time *(Arthur, 2007)*
- 1324-, 1342-, 1423- and 1432-avoiding in $\mathcal{O}(n \log \log \log n)$ time *(Arthur, 2007)*
- no pattern-specific $o(n \log n)$ algorithm known for 2413-avoiding!

Universal approach:

- $n\, 2^{(\alpha(n))^{\mathcal{O}(|\pi|)}}$ time *(Chalermsook et al., 2015)*
- $n\, 2^{\mathcal{O}(\alpha(n)+|\pi|^2)}$ time *(Chalermsook et al., 2023)*
  where $\alpha(\cdot)$ is the *inverse-Ackermann function*

Previous results

Theorem (*Fredman, 1976*)

For any set Γ of permutations of length n, there exists a decision tree of depth $\log_2 |\Gamma| + 2n$ that sorts every input from Γ.

Pattern-specific approach:

- k-increasing in $\mathcal{O}(\log k \cdot n)$ time
- 231-avoiding in $\mathcal{O}(n)$ time *(Knuth, 1968)*
- 1234-, 1243- and 2143-avoiding in $\mathcal{O}(n)$ time *(Arthur, 2007)*
- 1324-, 1342-, 1423- and 1432-avoiding in $\mathcal{O}(n \log \log \log n)$ time *(Arthur, 2007)*
- no pattern-specific $o(n \log n)$ algorithm known for 2413-avoiding!

Universal approach:

- $n\, 2^{(\alpha(n))^{\mathcal{O}(|\pi|)}}$ time *(Chalermsook et al., 2015)*
- $n\, 2^{\mathcal{O}(\alpha(n)+|\pi|^2)}$ time *(Chalermsook et al., 2023)*
  where $\alpha(\cdot)$ is the *inverse-Ackermann function*

Recent breakthrough

$\mathcal{O}(c_\pi^2 \cdot n)$ time assuming **Dynamic Optimality Conjecture** (Berendsohn, **Kozma** & **O.**, 2024).

Optimal Sorting

Theorem ***(O., 2024)***

There is a comparison-based algorithm that sorts π-avoiding sequences of length n in $\mathcal{O}((\log s_\pi + 1) \cdot n)$ time even if π is a priori unknown.

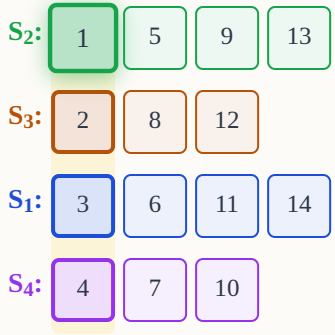
Matches the information-theoretic lower bound!

In this talk, we mostly assume that π is known and the input is a permutation (no repeated values).

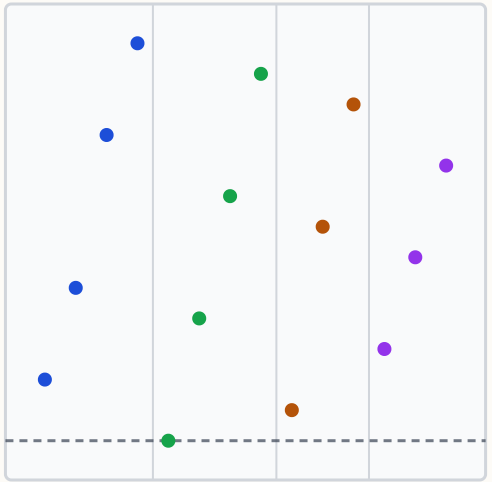
(I) Efficient Merging

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total



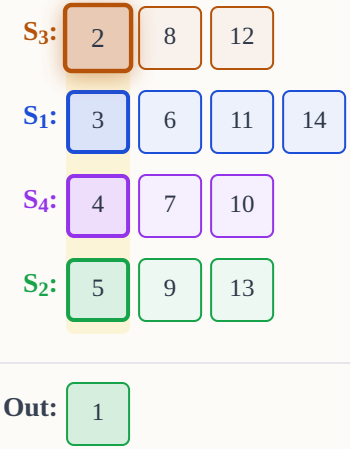
Out:

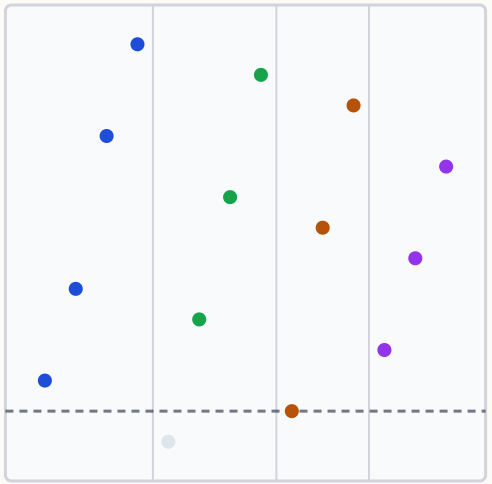


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

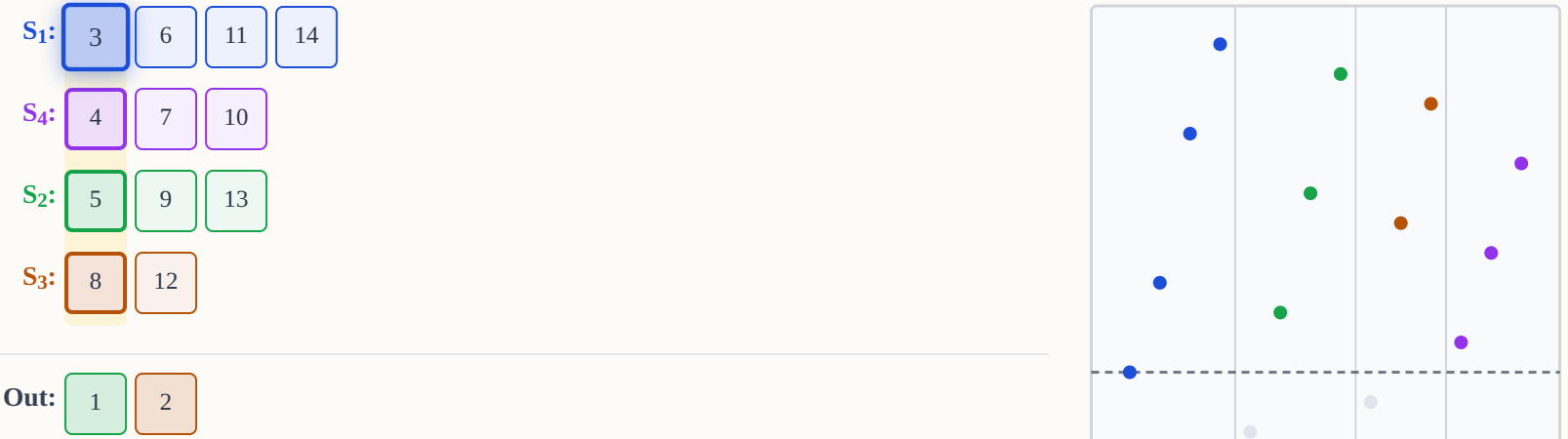




Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

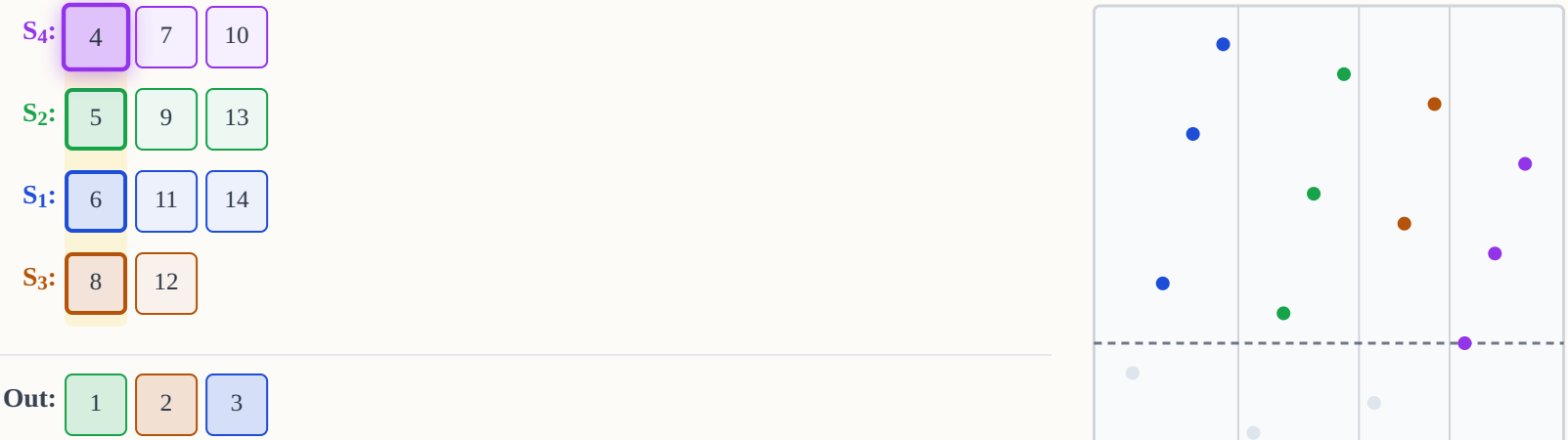
- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total



Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

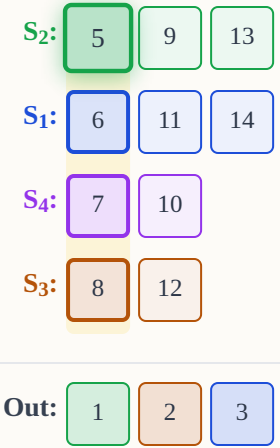
- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

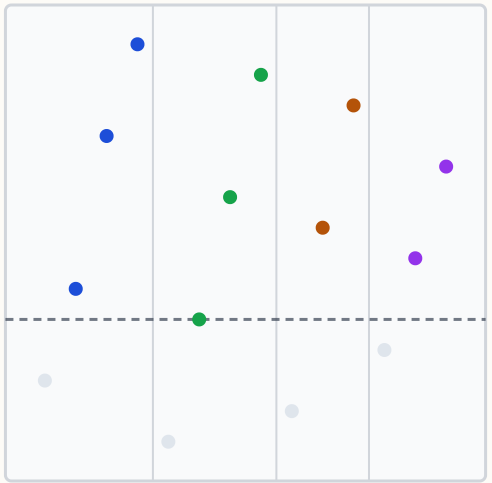


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

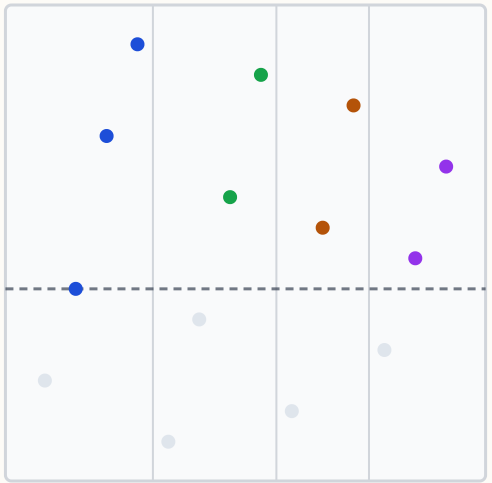




Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

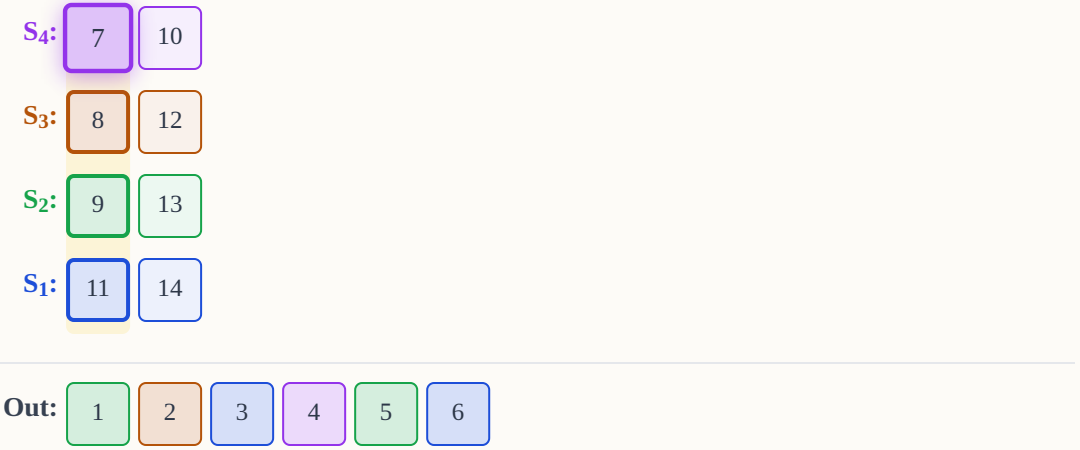
- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

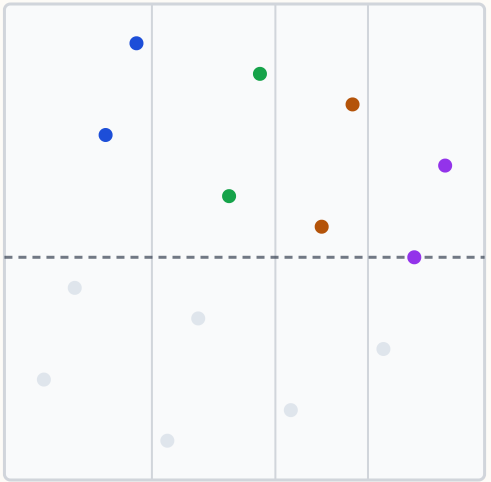


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

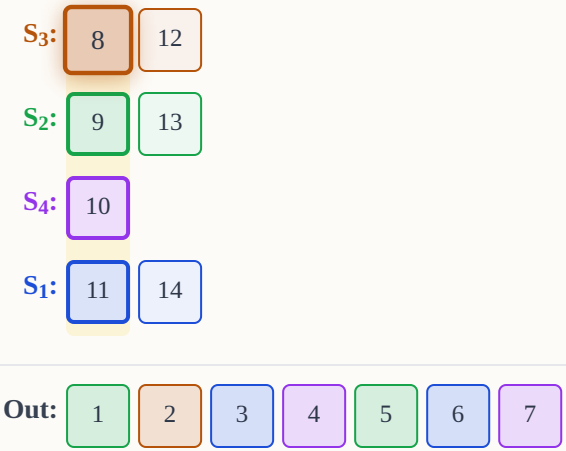


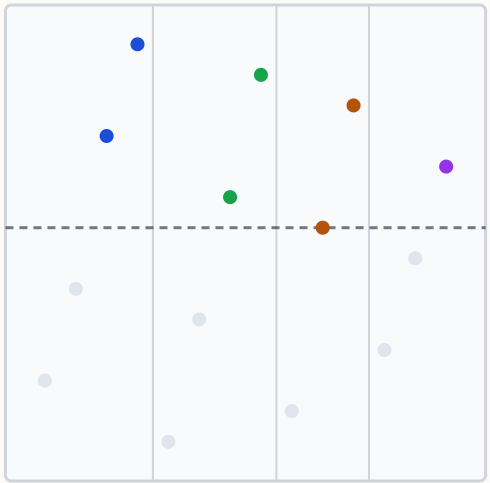


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

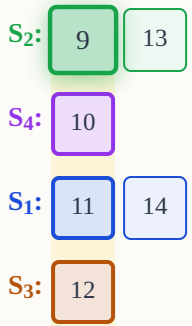


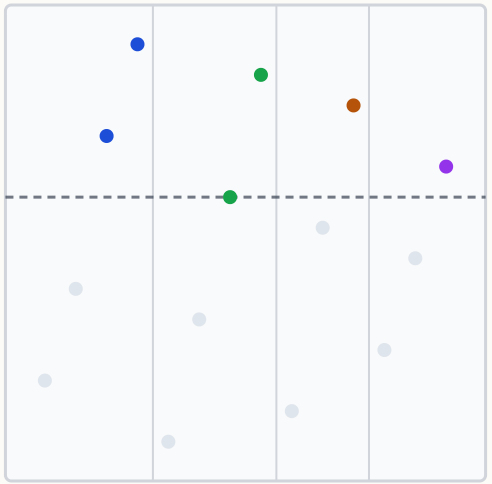


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

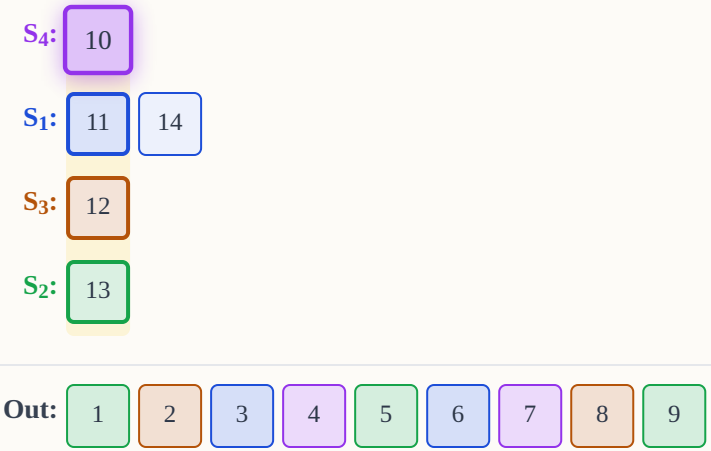


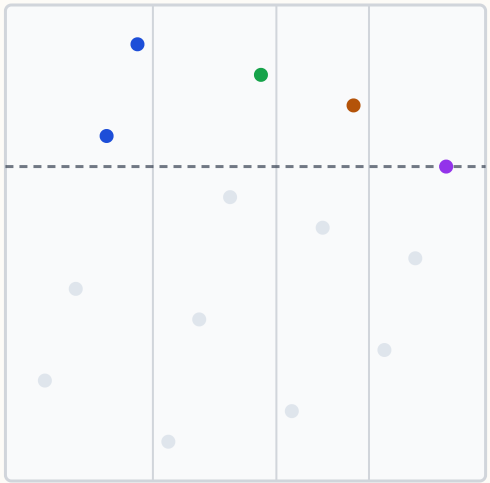


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

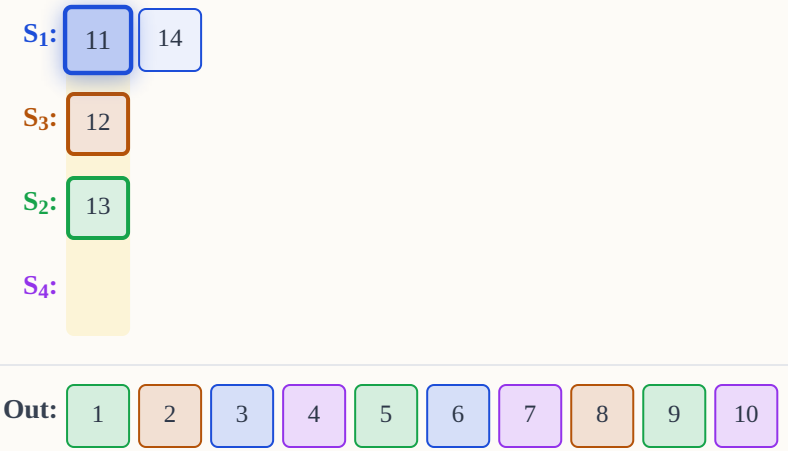


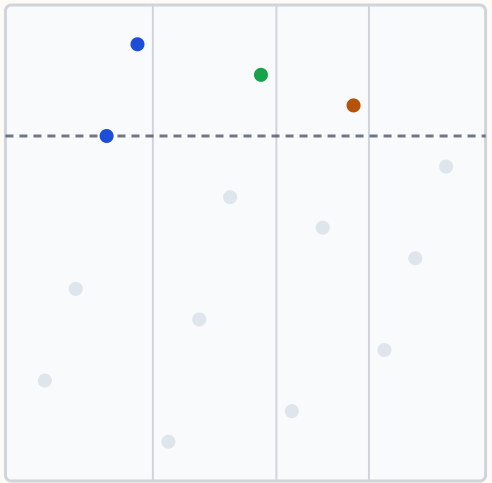


Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

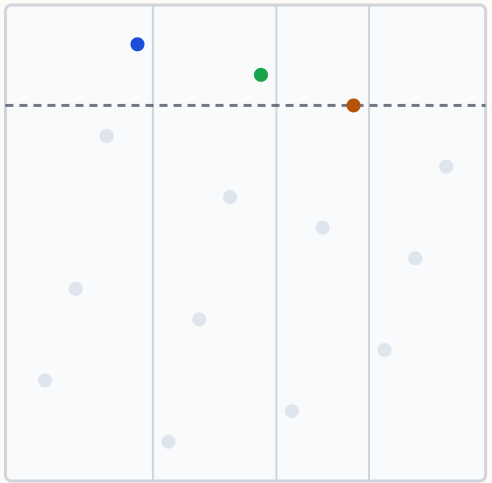




Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

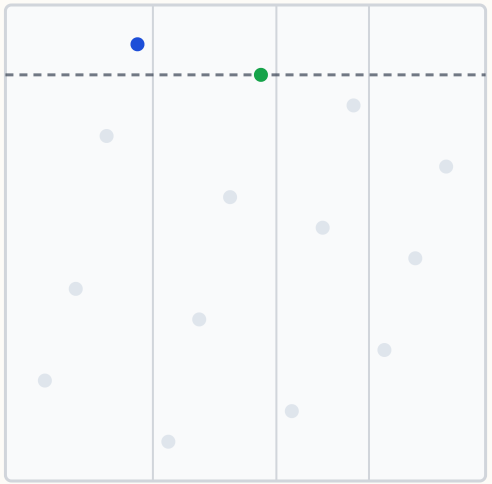
- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total



Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

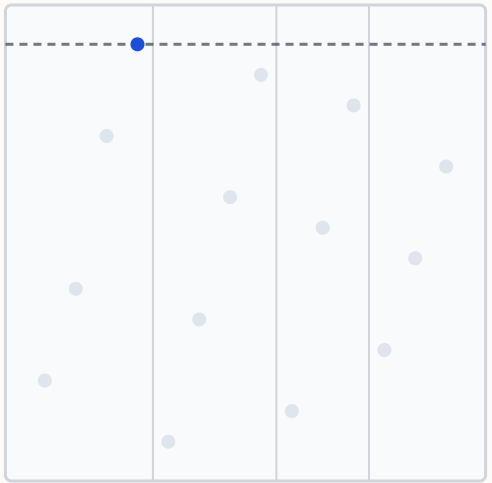
- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total



Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total



Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Multiway merge

- Merge d sorted sequences (combined length n) — maintain a min-heap on their current heads
- Always extract the global minimum, advance that sequence's pointer: $\mathcal{O}(n \log d)$ total

Time complexity: $T(n) = d \cdot T(n/d) + \Theta(n \log d)$ which solves to $\Theta(n \log n)$.

Pattern-avoiding merge

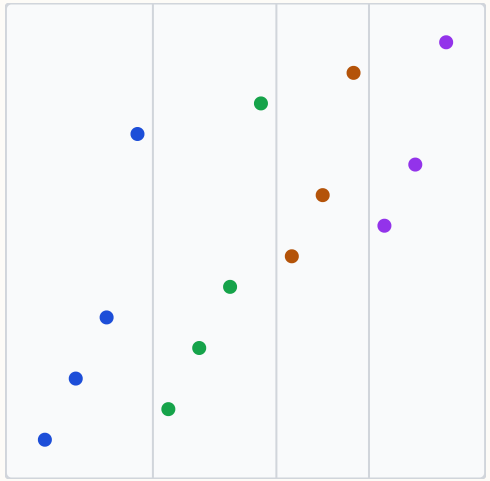
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

Claim: The procedure terminates in $R \leq m$ rounds.

Proof: Matrix with m columns (one per sequence) and R rows (one per round); each round except for the last one involved exactly d sequences.

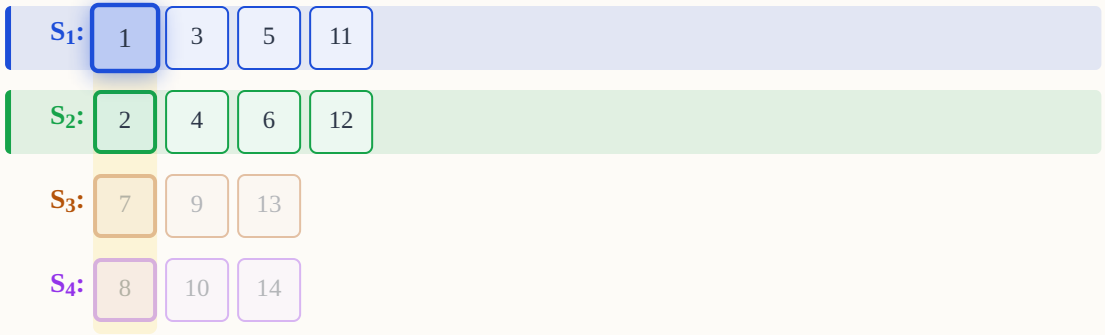
If $R > m$, then $\#\text{1-entries} \geq d \cdot (R-1) > c_\pi \cdot \max(m, R)$

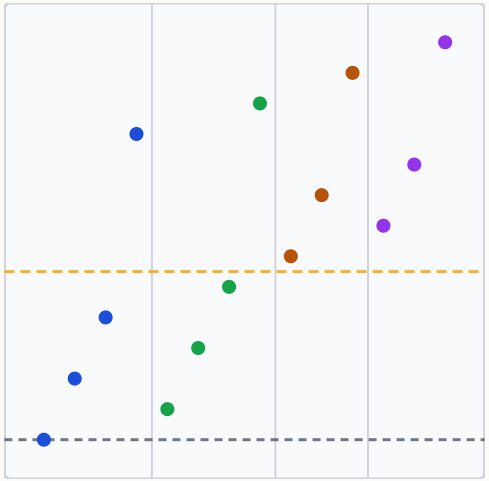
$\rightarrow$ matrix (and hence input) contains π. ■



Pattern-avoiding merge

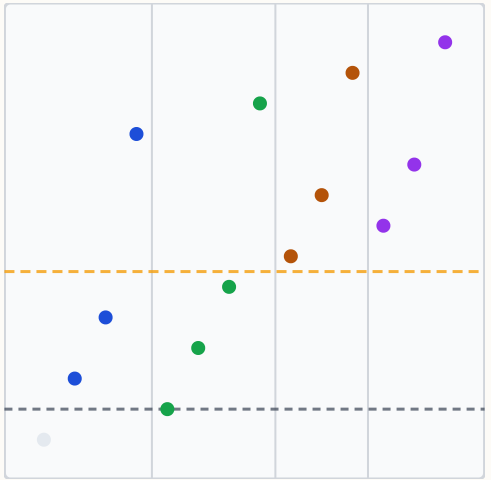
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round





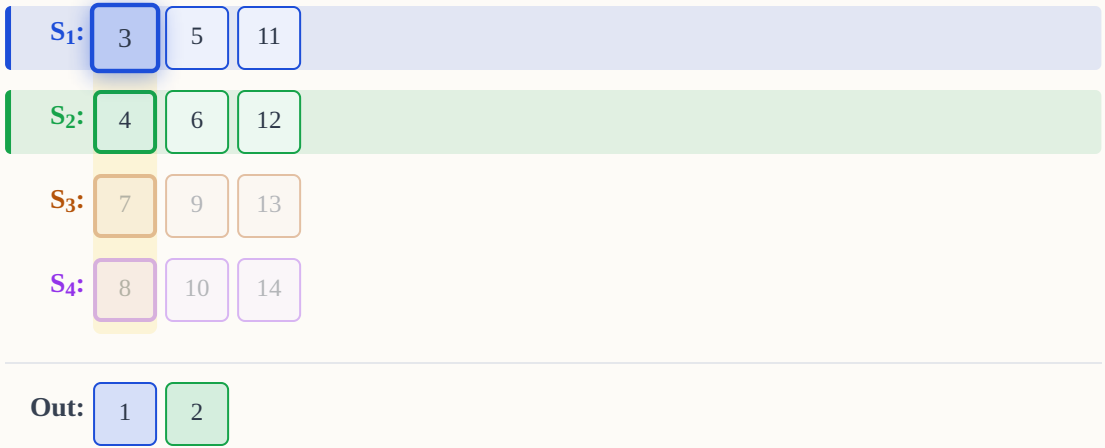
Pattern-avoiding merge

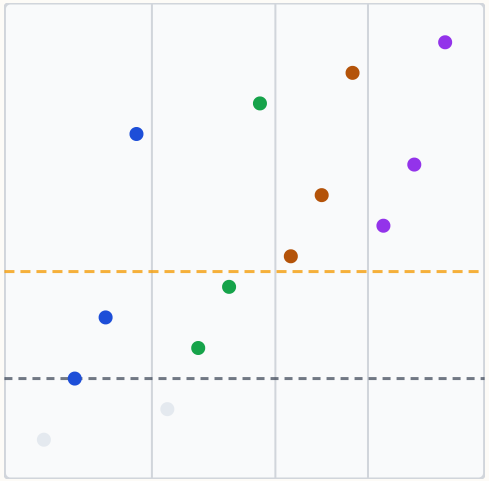
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round



Pattern-avoiding merge

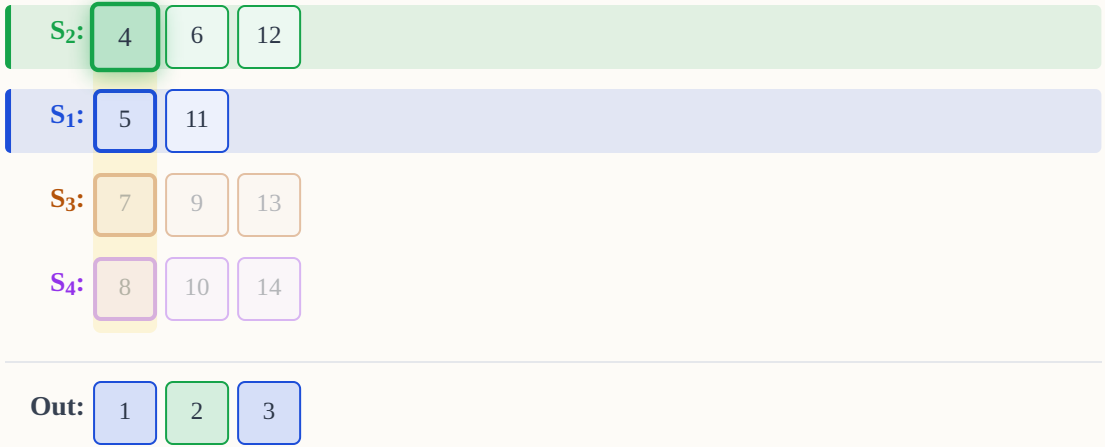
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

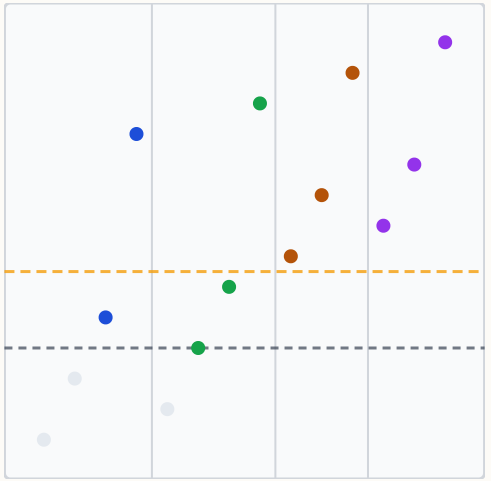




Pattern-avoiding merge

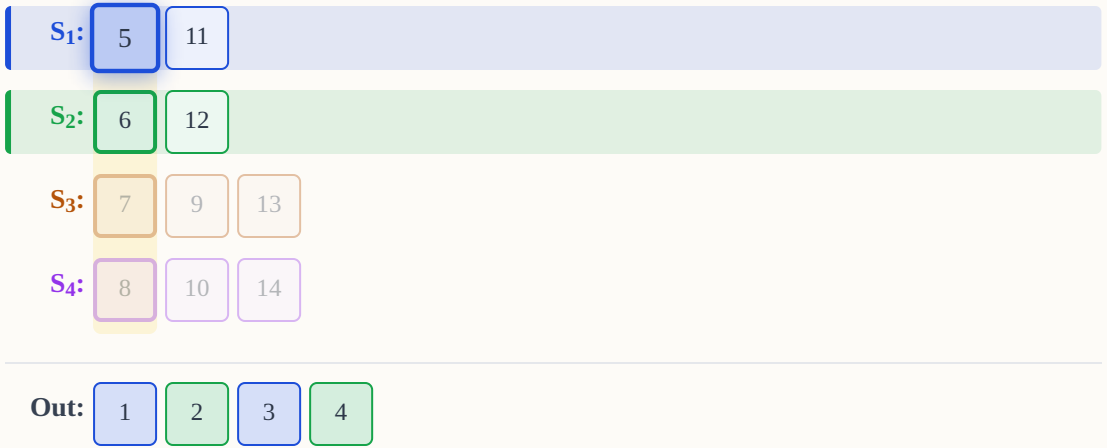
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

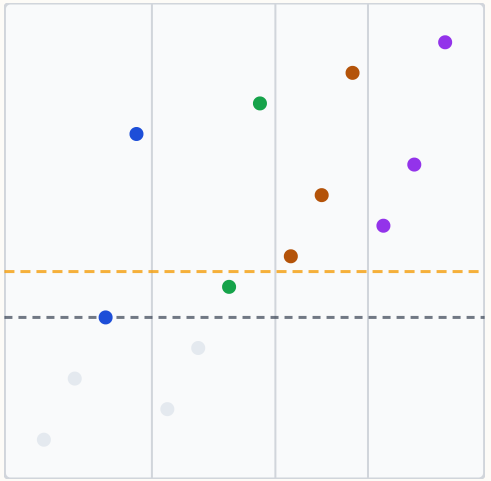




Pattern-avoiding merge

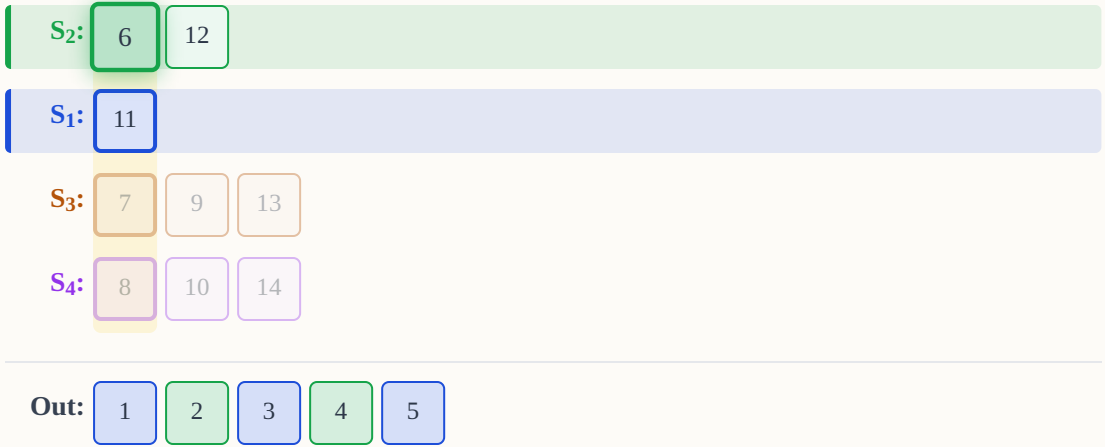
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

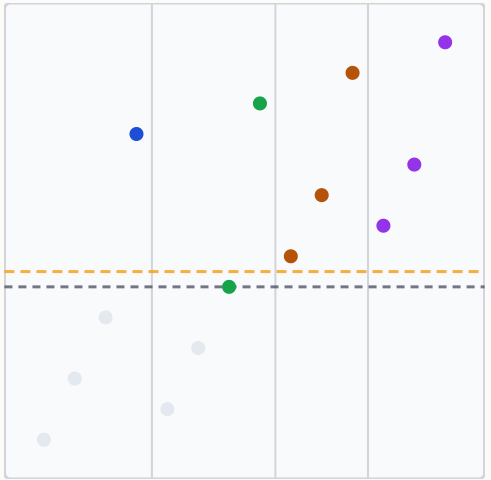




Pattern-avoiding merge

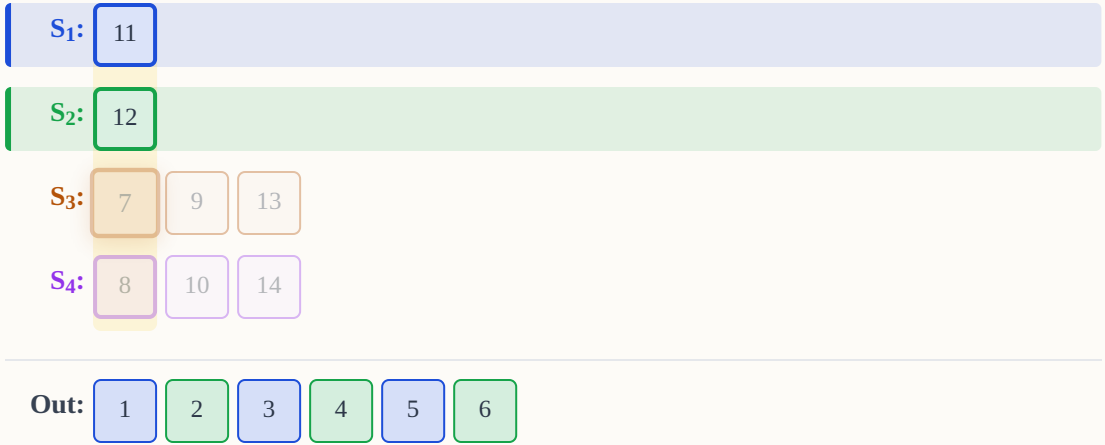
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

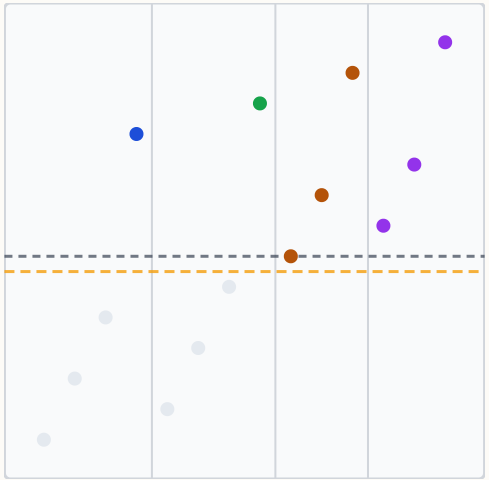




Pattern-avoiding merge

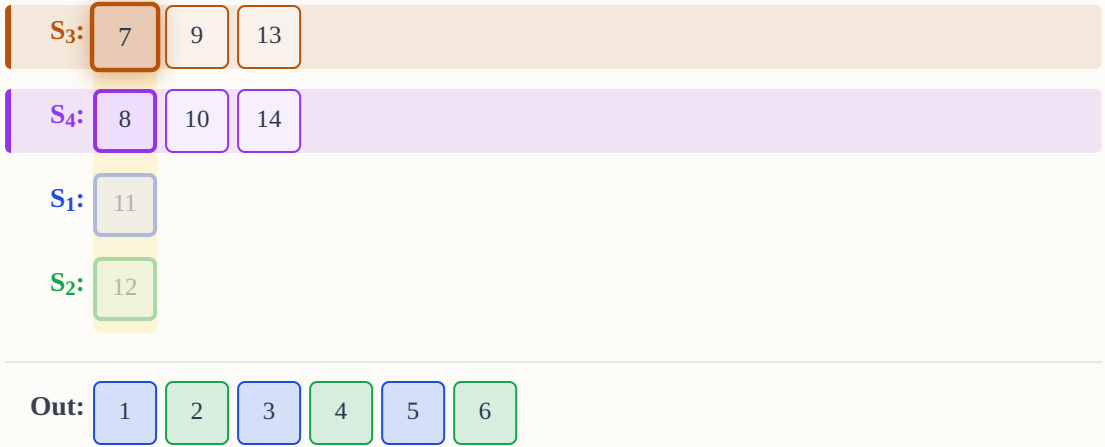
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

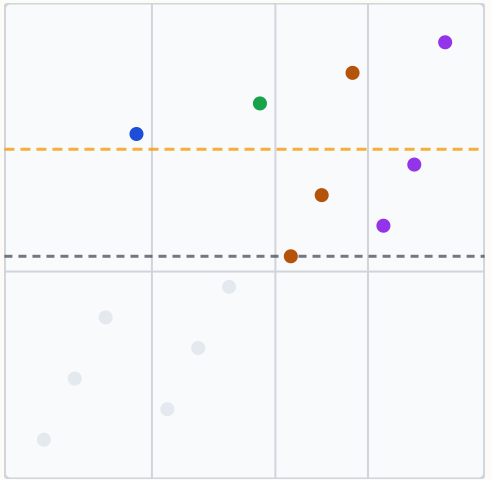




Pattern-avoiding merge

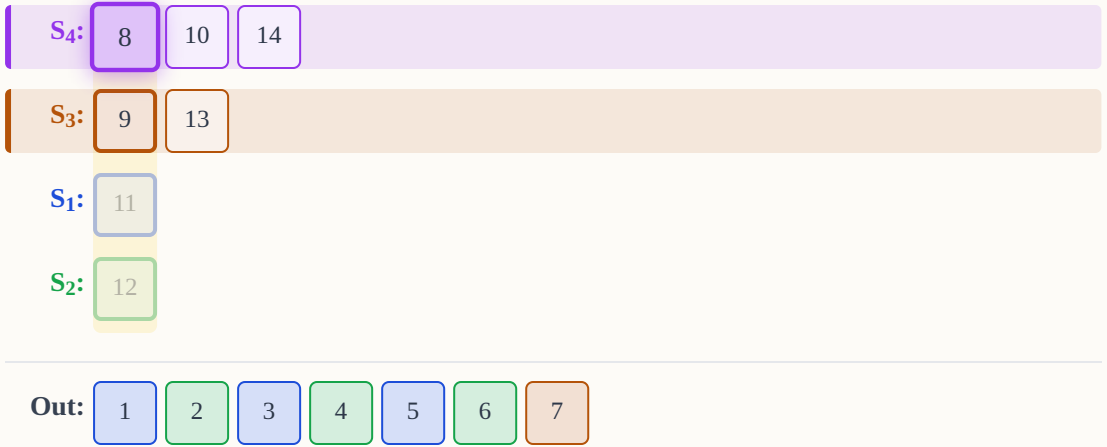
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

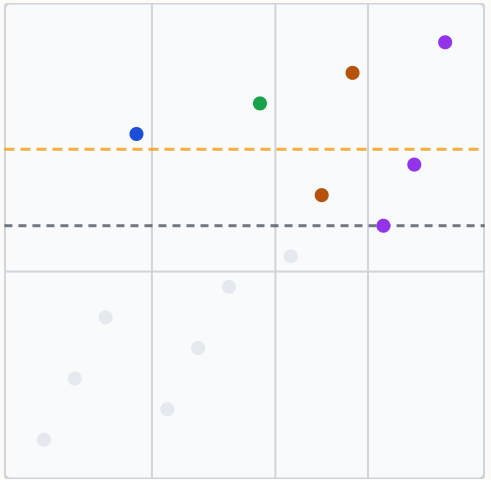




Pattern-avoiding merge

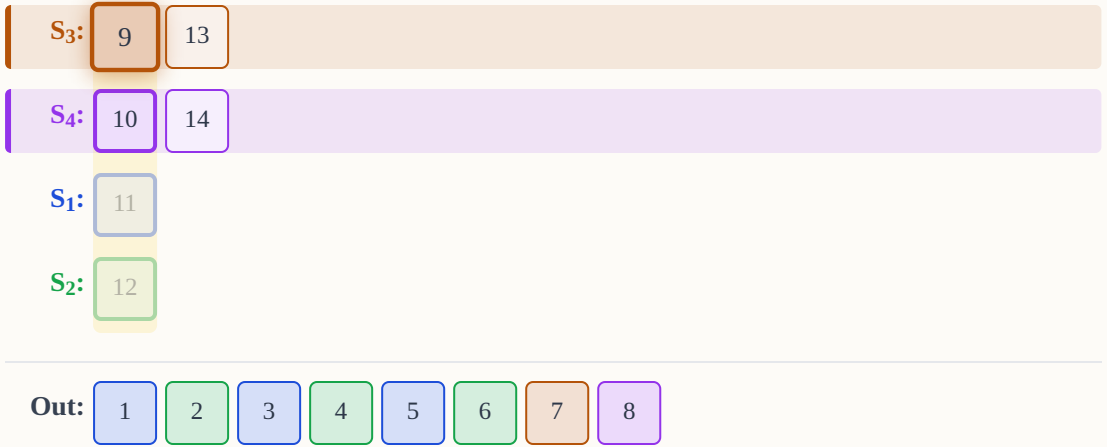
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

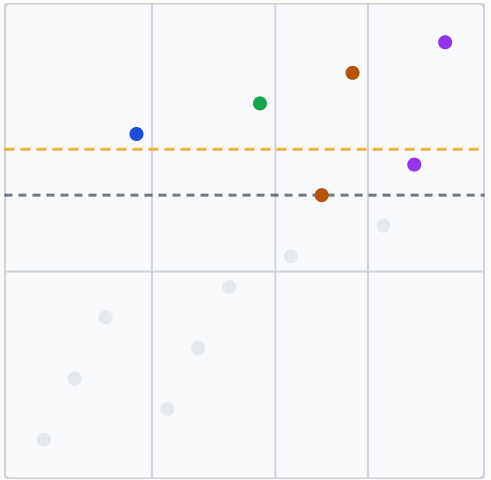




Pattern-avoiding merge

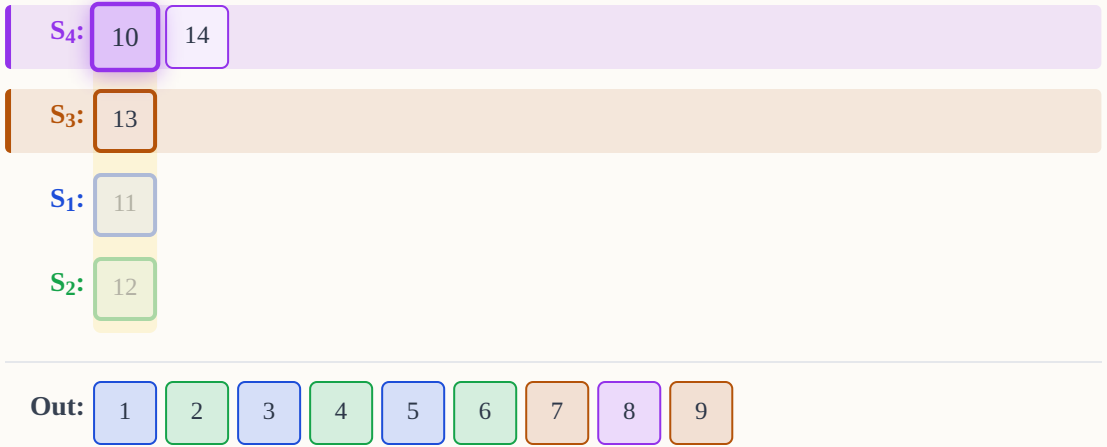
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

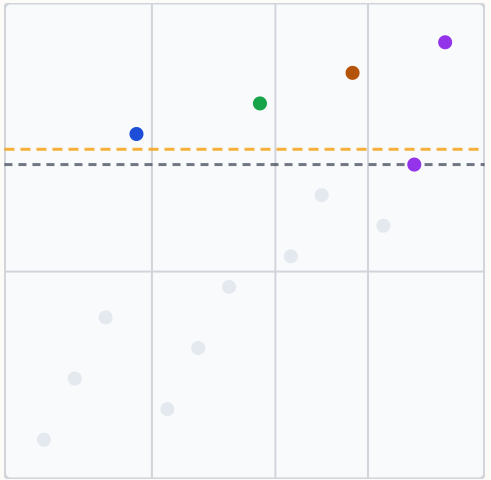




Pattern-avoiding merge

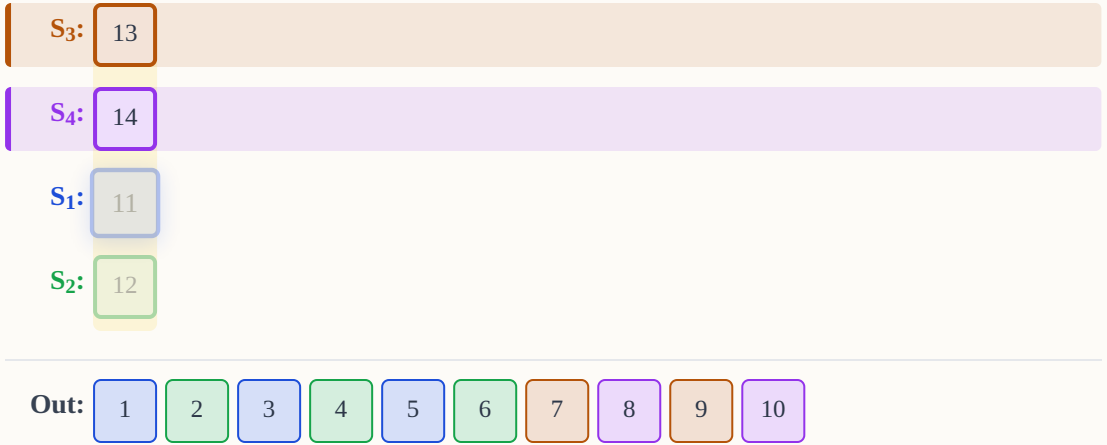
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

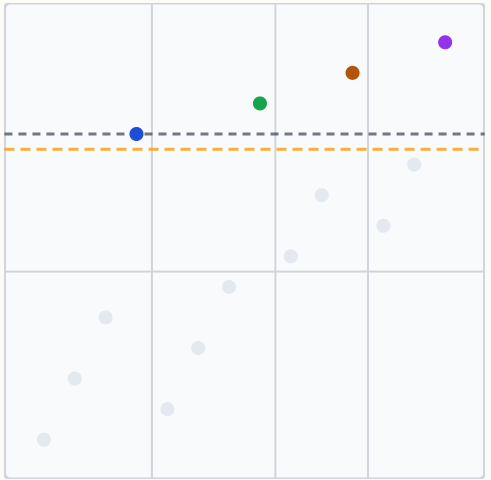




Pattern-avoiding merge

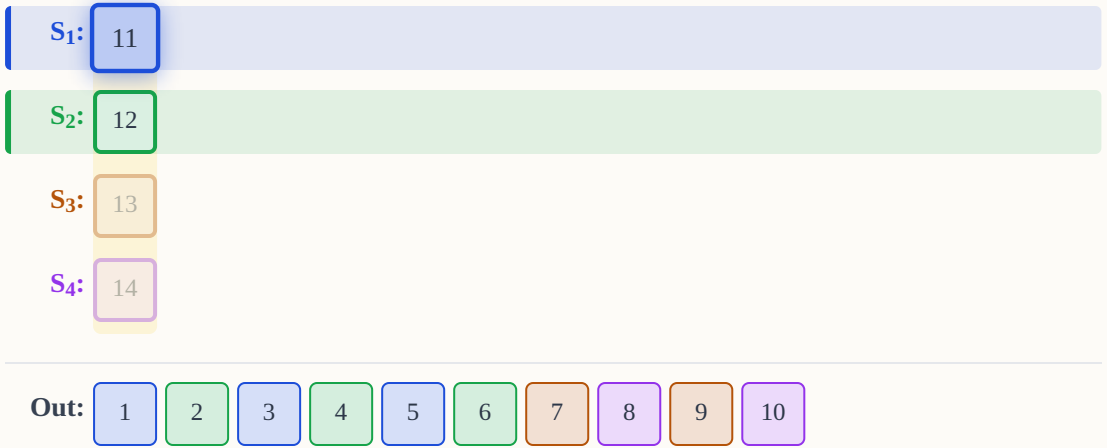
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

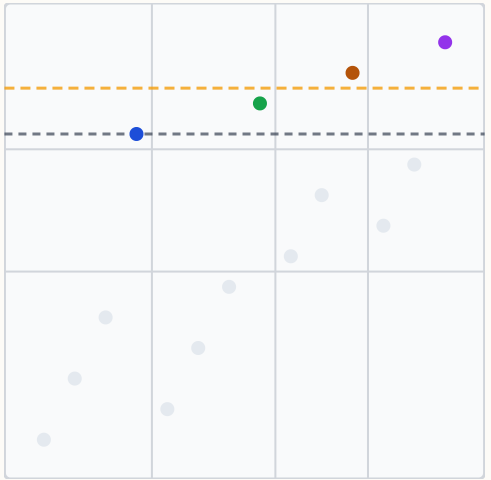




Pattern-avoiding merge

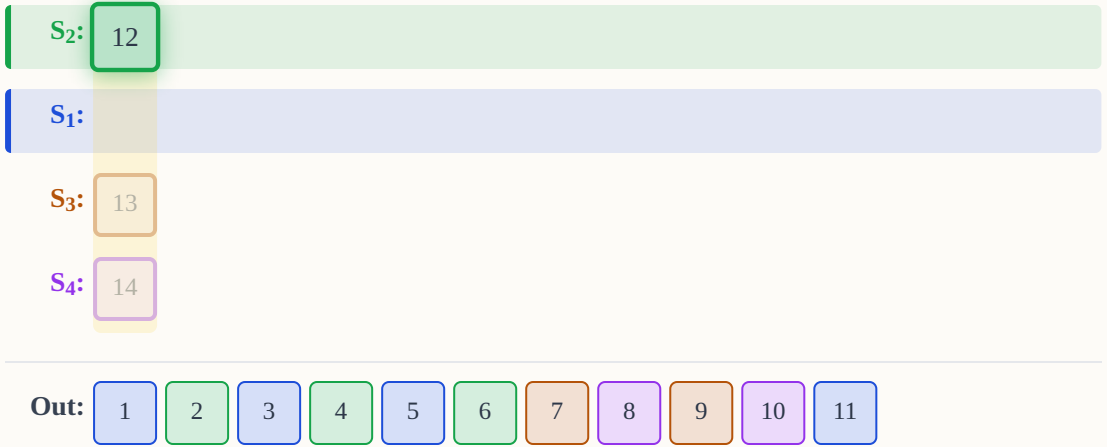
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

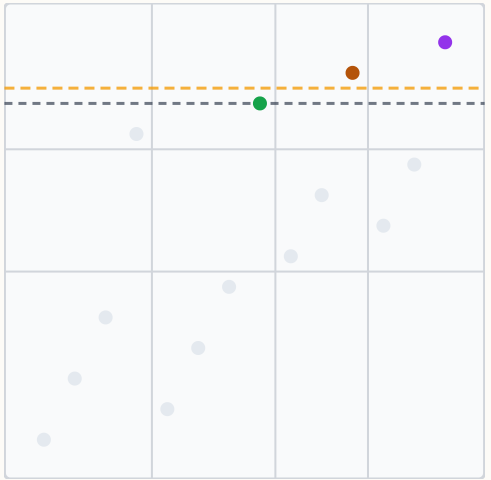




Pattern-avoiding merge

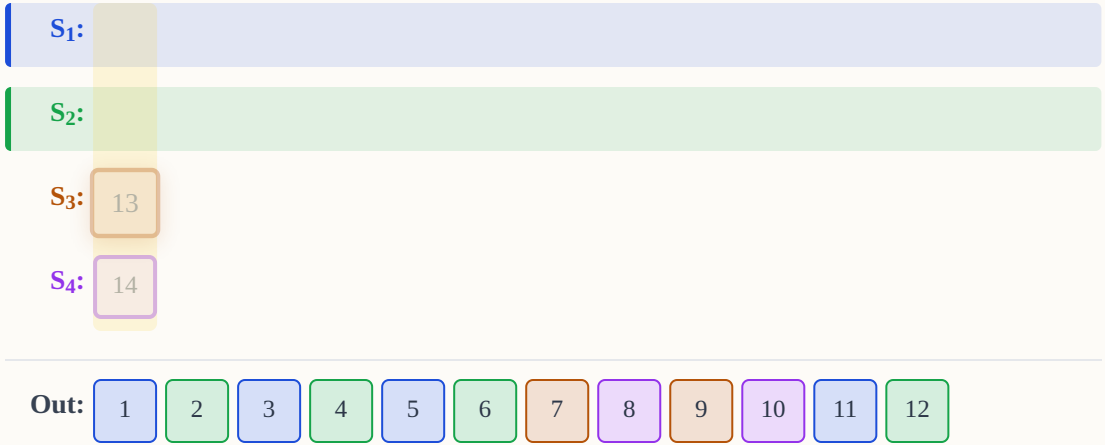
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

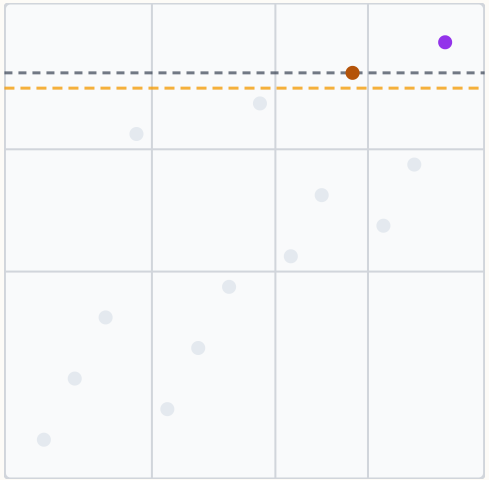




Pattern-avoiding merge

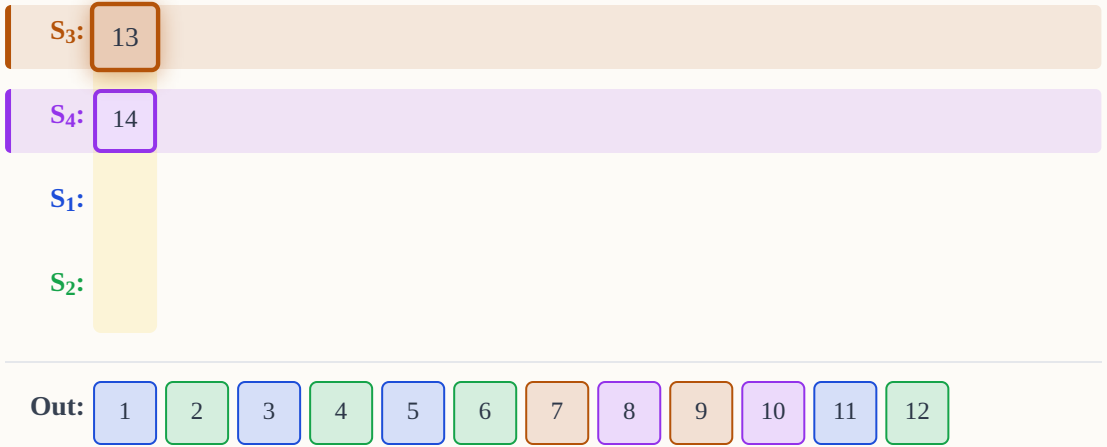
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

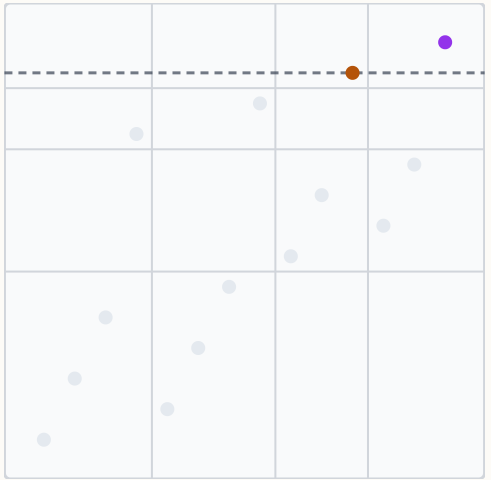




Pattern-avoiding merge

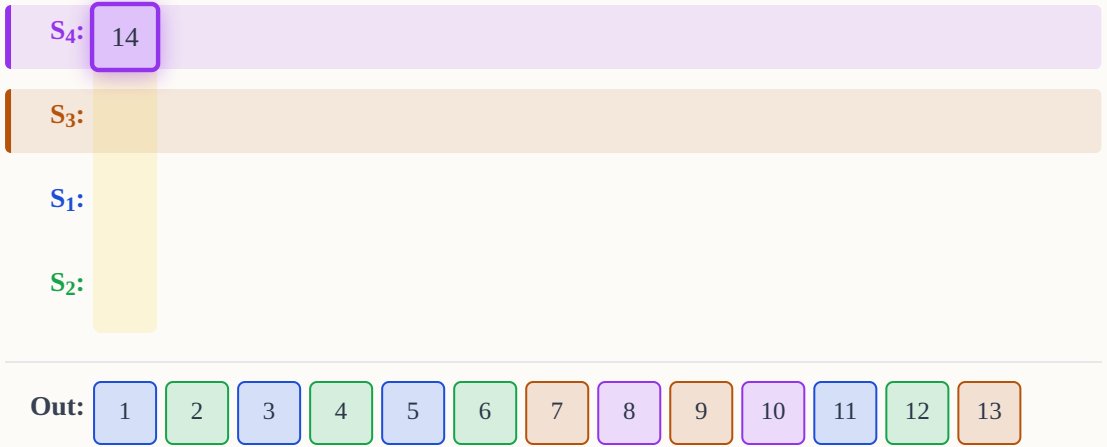
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

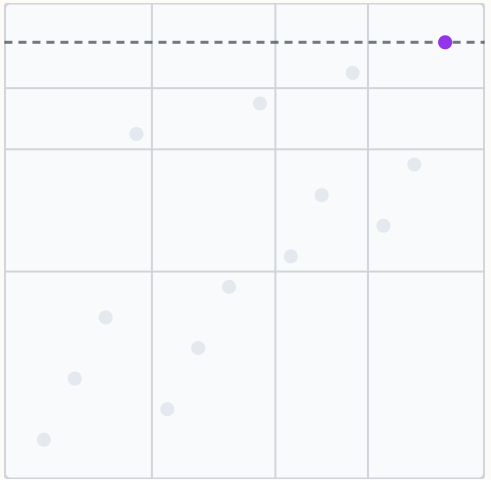




Pattern-avoiding merge

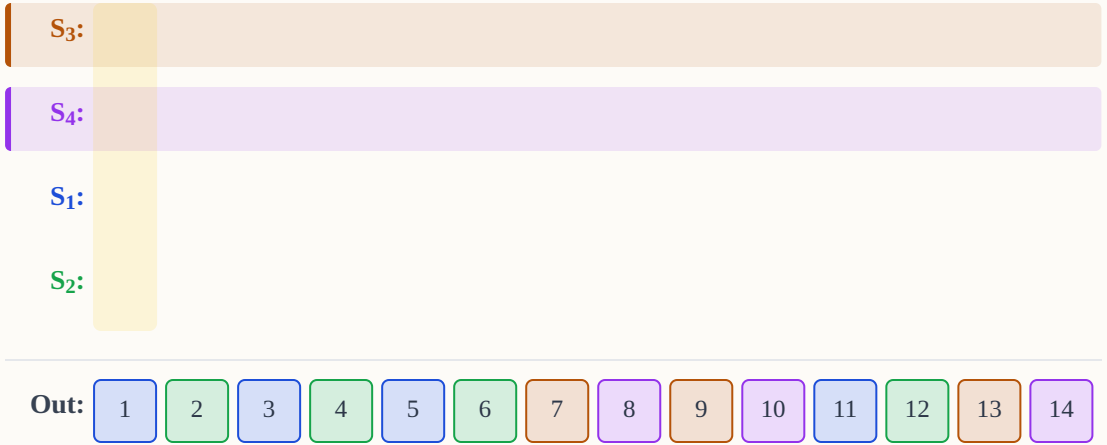
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round





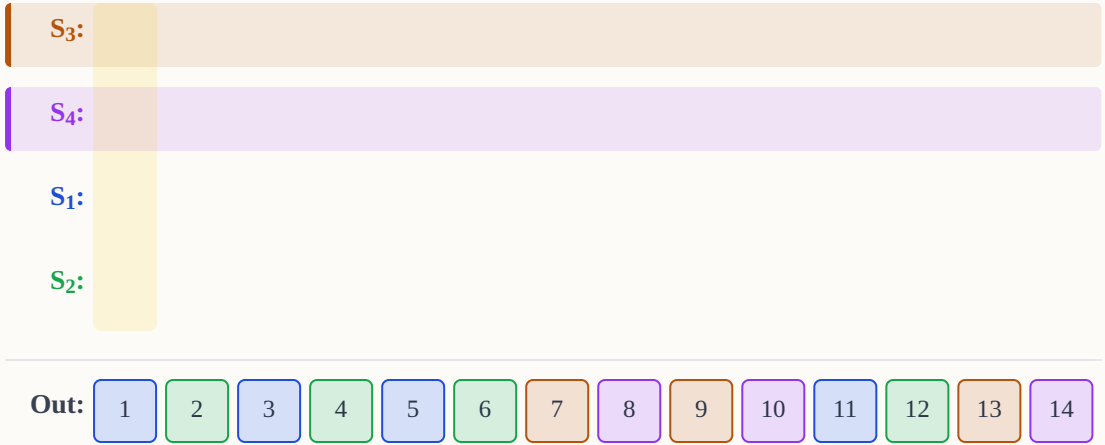
Pattern-avoiding merge

- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round



Pattern-avoiding merge

- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round



- Runtime: $\mathcal{O}(n \cdot \log d) + \mathcal{O}(R \cdot d \cdot \log n)$ where R is the number of rounds.

Pattern-avoiding merge

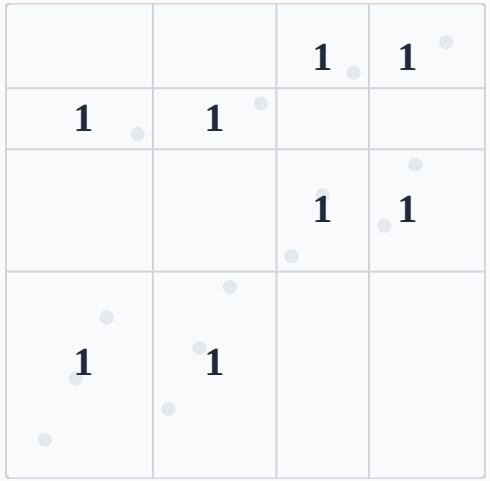
- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

Claim: The procedure terminates in $R \leq m$ rounds.

Proof: Matrix with m columns (one per sequence) and R rows (one per round); each round except for the last one involved exactly d sequences.

If $R > m$, then $\#\text{1-entries} \geq d \cdot (R-1) > c_\pi \cdot \max(m, R)$

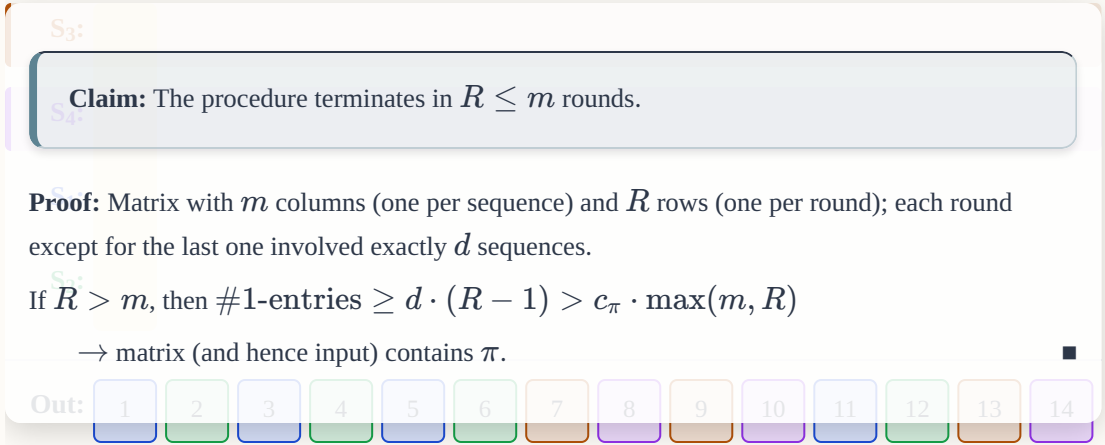
$\rightarrow$ matrix (and hence input) contains π. ■



- Runtime: $\mathcal{O}(n \cdot \log d) + \mathcal{O}(R \cdot d \cdot \log n)$ where R is the number of rounds.

Pattern-avoiding merge

- Input: m presorted sequences $S_1, \ldots, S_m$ of combined length n
- Pick the $d \leftarrow \lceil 2c_\pi \rceil$ sequences with smallest heads as **active**; the $(d+1)$-th defines the **round cutoff** x
- Output from the d active sequences while their heads are $< x$, then start a new round

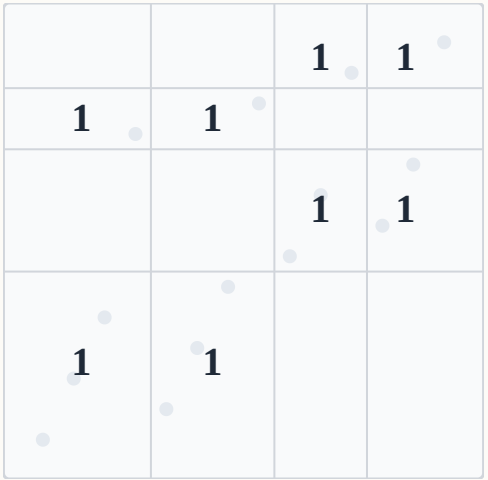


Claim: The procedure terminates in $R \leq m$ rounds.

Proof: Matrix with m columns (one per sequence) and R rows (one per round); each round except for the last one involved exactly d sequences.

If $R > m$, then $\#\text{1-entries} \geq d \cdot (R-1) > c_\pi \cdot \max(m, R)$

$\rightarrow$ matrix (and hence input) contains π. ■



$$\mathcal{O}(n \cdot \log d + m \cdot d \cdot \log n)$$

- Runtime: ~~$\mathcal{O}(n \cdot \log d) + \mathcal{O}(R \cdot d \cdot \log n)$~~ where R is the number of rounds.

Pattern-avoiding merge

Let $d \leftarrow \lceil 2c_\pi \rceil,\ m \leftarrow n/\log n$

Input: π-avoiding sequence S partitioned into m presorted sequences $S_1, \ldots, S_m$

1. $S'_1, \ldots, S'_{\lfloor m/d \rfloor} \leftarrow$ merge consecutive d-tuples of sequences
2. while there are some non-exhausted sequences:
3. $\quad S'_{i_1}, \ldots, S'_{i_{d+1}} \leftarrow d+1$ sequences with smallest initial elements
4. $\quad x \leftarrow$ initial element of $S'_{i_{d+1}}$ (largest out of these)
5. $\quad$ output merge of $S'_{i_1}, \ldots, S'_{i_d}$ while smaller than x

Pattern-avoiding merge

Let $d \leftarrow \lceil 2c_\pi \rceil,\ m \leftarrow n/\log n$

Input: π-avoiding sequence S partitioned into m presorted sequences $S_1, \ldots, S_m$

1. $S'_1, \ldots, S'_{\lfloor m/d \rfloor} \leftarrow$ merge consecutive d-tuples of sequences — $\mathcal{O}(\log d \cdot n)$
2. while there are some non-exhausted sequences:
3. $\quad S'_{i_1}, \ldots, S'_{i_{d+1}} \leftarrow d+1$ sequences with smallest initial elements — $\mathcal{O}(d \cdot \log n)$
4. $\quad x \leftarrow$ initial element of $S'_{i_{d+1}}$ (largest out of these) — $\mathcal{O}(\log n)$
5. $\quad$ output merge of $S'_{i_1}, \ldots, S'_{i_d}$ while smaller than x — $\mathcal{O}(\log d \cdot n)$ overall

Pattern-avoiding merge

Let $d \leftarrow \lceil 2c_\pi \rceil,\ m \leftarrow n/\log n$

Input: π-avoiding sequence S partitioned into m presorted sequences $S_1, \ldots, S_m$

1. $S'_1, \ldots, S'_{\lfloor m/d \rfloor} \leftarrow$ merge consecutive d-tuples of sequences — $\mathcal{O}(\log d \cdot n)$
2. while there are some non-exhausted sequences:
3. $\quad S'_{i_1}, \ldots, S'_{i_{d+1}} \leftarrow d+1$ sequences with smallest initial elements — $\mathcal{O}(d \cdot \log n)$
4. $\quad x \leftarrow$ initial element of $S'_{i_{d+1}}$ (largest out of these) — $\mathcal{O}(\log n)$
5. $\quad$ output merge of $S'_{i_1}, \ldots, S'_{i_d}$ while smaller than x — $\mathcal{O}(\log d \cdot n)$ overall

Can be used recursively to sort π-avoiding sequences in time $\mathcal{O}(\log c_\pi \cdot n \log^* n)$.

(II) Sorting Many Short Sequences

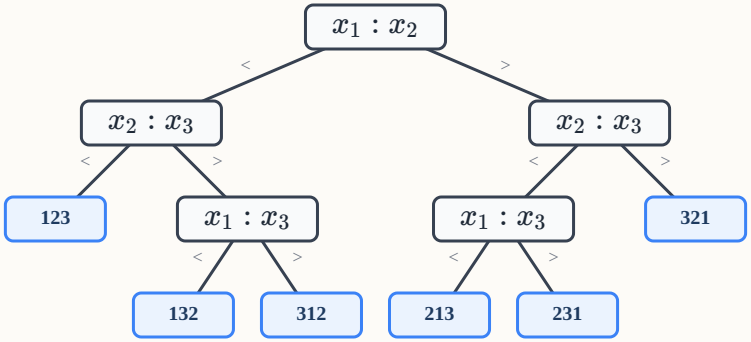
Decision trees

- A *decision tree* T for input length k is a full binary tree with internal nodes labeled by elements of $[k]^2$ and leaves labeled by permutations of length k.

Lemma

There are at most $2^{2^{h+o(k)+o(h)}}$ decision trees of height at most h for input length k.

In our case $k = \log\log\log n$ and thus, there are $o(n)$ trees to consider.



A decision tree for k = 3: nodes labeled by [3]², leaves labeled by permutations of length 3.

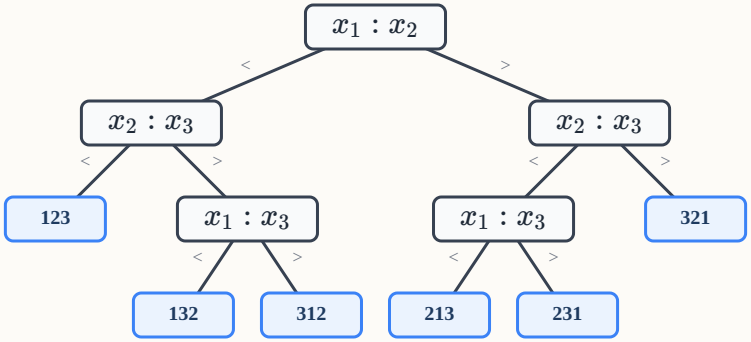
Decision trees

- A *decision tree* T for input length k is a full binary tree with internal nodes labeled by elements of $[k]^2$ and leaves labeled by permutations of length k.

Lemma

There are at most $2^{2^{h+o(k)+o(h)}}$ decision trees of height at most h for input length k.

In our case $k = \log\log\log n$ and thus, there are $o(n)$ trees to consider.



A decision tree for k = 3: nodes labeled by [3]², leaves labeled by permutations of length 3.

Corollary (*Fredman, 1976* • *Marcus, Tardos, 2004*)

For every pattern π, there exists a decision tree of depth $\mathcal{O}((\log s_\pi + 1) \cdot k)$ that sorts every π-avoiding permutation of length k.

Brute-force algorithm

Recall that we assume the knowledge of π (and s_π).

1. $P \leftarrow$ generate all π-avoiding permutations of length k
2. for every decision tree T of depth $\leq \mathcal{O}((\log s_\pi + 1) \cdot k)$:
3. if T sorts every permutation from P:
4. $T_{\mathsf{opt}} \leftarrow T$ and break
5. sort every sequence on input using T_{opt}.

Brute-force algorithm

Recall that we assume the knowledge of π (and s_π).

1. $P \leftarrow$ generate all π-avoiding permutations of length k
2. for every decision tree T of depth $\leq \mathcal{O}((\log s_\pi + 1) \cdot k)$:
3. if T sorts every permutation from P:
4. $T_{\mathsf{opt}} \leftarrow T$ and break
5. sort every sequence on input using T_{opt}.

$o(n)$

$\mathcal{O}\big((\log s_\pi + 1) \cdot k \cdot \frac{n}{k}\big)$

Brute-force algorithm

Recall that we assume the knowledge of π (and s_π).

1. $P \leftarrow$ generate all π-avoiding permutations of length k — $o(n)$
2. for every decision tree T of depth $\leq \mathcal{O}((\log s_\pi + 1) \cdot k)$:
3. if T sorts every permutation from P:
4. $T_{\mathsf{opt}} \leftarrow T$ and break
5. sort every sequence on input using T_{opt}. — $\mathcal{O}\big((\log s_\pi + 1) \cdot k \cdot \frac{n}{k}\big)$

Total runtime

$$o(n) + \mathcal{O}\big((\log s_\pi + 1) \cdot k \cdot \tfrac{n}{k}\big) = \mathcal{O}((\log s_\pi + 1) \cdot n)$$

Crucially, there is only $o(n)$ of both candidate decision trees and π-avoiding permutations of length $k = \log\log\log n$.

Brute-force without knowing π

Fix an enumeration of decision trees in increasing order by depth.

Input: sequences $S_1, \ldots, S_{n/k}$, each of length k

1. $T \leftarrow$ first decision tree in the enumeration sequence
2. for $i \in \{1, \ldots, n/k\}$:
3. $\quad S'_i \leftarrow$ rearrange S_i using T
4. $\quad$ while S'_i is not sorted:
5. $\quad\quad T \leftarrow$ next decision tree in the enumeration sequence
6. $\quad\quad S'_i \leftarrow$ rearrange S_i using T
7. $\quad$ output S'_i

Brute-force without knowing π

Fix an enumeration of decision trees in increasing order by depth.

Input: sequences $S_1, \ldots, S_{n/k}$, each of length k

1. $T \leftarrow$ first decision tree in the enumeration sequence
2. for $i \in \{1, \ldots, n/k\}$:
3. $\quad S'_i \leftarrow$ rearrange S_i using T — $\mathcal{O}(\log s_\pi \cdot k)$
4. $\quad$ while S'_i is not sorted: — $\mathcal{O}(k)$
5. $\quad\quad T \leftarrow$ next decision tree in the enumeration sequence — $\mathcal{O}(1)$ amortized
6. $\quad\quad S'_i \leftarrow$ rearrange S_i using T — $\mathcal{O}(\log s_\pi \cdot k)$
7. $\quad$ output S'_i

Brute-force without knowing π

Fix an enumeration of decision trees in increasing order by depth.

Input: sequences $S_1, \ldots, S_{n/k}$, each of length k

1. $T \leftarrow$ first decision tree in the enumeration sequence
2. for $i \in \{1, \ldots, n/k\}$:
3. $\quad S'_i \leftarrow$ rearrange S_i using T — $\mathcal{O}(\log s_\pi \cdot k)$
4. $\quad$ while S'_i is not sorted: — $\mathcal{O}(k)$
5. $\quad\quad T \leftarrow$ next decision tree in the enumeration sequence — $\mathcal{O}(1)$ amortized
6. $\quad\quad S'_i \leftarrow$ rearrange S_i using T — $\mathcal{O}(\log s_\pi \cdot k)$
7. $\quad$ output S'_i

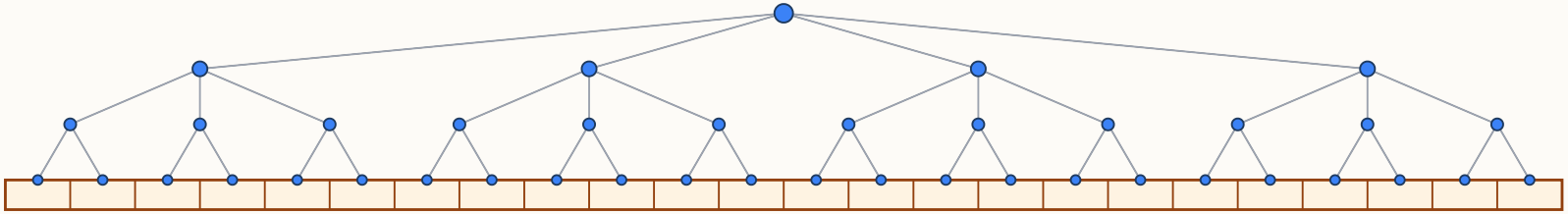
Claim: In total, there are exactly n/k successful and $o(n)$ unsuccessful applications of T.

Proof: The sequence S_i is sorted after a successful application of $T \Rightarrow 1$ per sequence. Each unsuccessful application advances the enumeration $\Rightarrow 1$ per decision tree, $o(n)$ in total. ∎

Putting it together

Input: π-avoiding sequence S of length n

1. Split S into blocks of length $k = \log \log \log n$
2. Sort each block via brute force over decision trees — (II)
3. Merge the sorted blocks via three levels of merge sort using the efficient merge — (I)

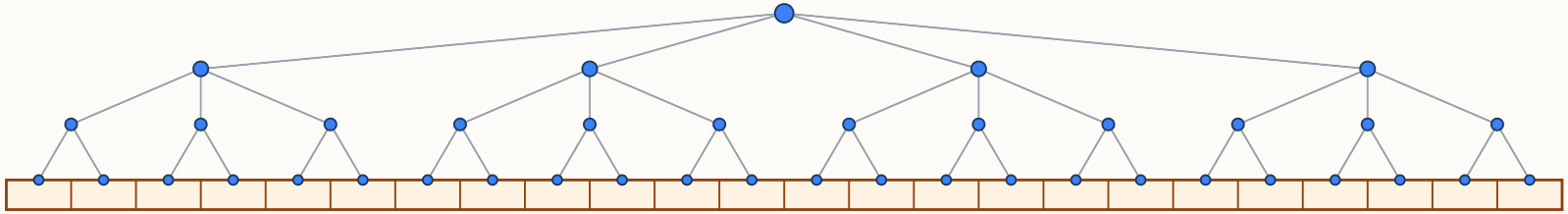


Recursion tree: depth 3, the leaves are the blocks of length k.

Putting it together

Input: π-avoiding sequence S of length n

1. Split S into blocks of length $k = \log\log\log n$ — $\mathcal{O}(n)$
2. Sort each block via brute force over decision trees — (II) — $\mathcal{O}((\log s_\pi + 1) \cdot n)$
3. Merge the sorted blocks via three levels of merge sort using the efficient merge — (I) — $\mathcal{O}((\log s_\pi + 1) \cdot n)$



Recursion tree: depth 3, the leaves are the blocks of length k.

Compact data structure

Data structures

We want to represent a π-avoiding permutation τ, supporting queries:

- Rank(i) which returns $\tau(i)$
- Unrank(j) which returns $\tau^{-1}(j)$

> To represent objects from some set Γ, we need at least $\log_2(\Gamma)$ bits.

Data structure is:

- **compact** if it needs $\mathcal{O}(\log_2(\Gamma))$ bits
- **succinct** if it needs $(1 + o(1)) \cdot \log_2(\Gamma)$ bits

Succinct data structures with $\mathcal{O}(1)$ queries exist for trees (implementing various queries), planar graphs, interval graphs, …

Our goal: space-efficient data structure for permutations with Rank/Unrank in $\mathcal{O}(1)$ time.

Technical interlude: Word-RAM

Definition ***(Word-RAM model)***

- Memory is an array of cells, each independently addressable and accessible in $\mathcal{O}(1)$ time.
- Memory is addressed in **words** of $w = \Theta(\log n)$ bits.
- Standard arithmetic, comparison, and bitwise operations in $\mathcal{O}(1)$ time over pairs of words *(Hagerup, 1998)*.

Crucially, we measure **space in bits**, not words – exactly what allows asymptotically matching information-theoretic lower bounds.

Previous results

General permutations:

- Trivial: $\Theta(n \log n)$ bits, $\mathcal{O}(1)$-time rank/unrank.
- $(1+\varepsilon)n \log n$ bits with $\mathcal{O}(1/\varepsilon)$-time queries *(Munro et al., 2012)*; this tradeoff is optimal *(Golynski, 2009)*.
- Succinct ($n \log n + o(n \log n)$ bits) possible, but query time is then necessarily super-constant *(Munro et al., 2012)*.

Separable permutations:

- Avoid $2413, 3142$; built from sums/skew-sums – binary tree representation.
- Succinct data structure with $\mathcal{O}(1)$-time queries *(Chakraborty et al., 2024)*.

Baxter permutations:

- Defined via vincular patterns, in bijection with floorplans *(Ackerman et al., 2006)*, *(Felsner et al., 2011)*; also natural tree representation.
- Succinct data structure, but with polylogarithmic query time *(Chakraborty et al., 2024)*.

Previous results

General permutations:

- Trivial: $\Theta(n \log n)$ bits, $\mathcal{O}(1)$-time rank/unrank.
- $(1+\varepsilon)n \log n$ bits with $\mathcal{O}(1/\varepsilon)$-time queries *(Munro et al., 2012)*; this tradeoff is optimal *(Golynski, 2009)*.
- Succinct ($n \log n + o(n \log n)$ bits) possible, but query time is then necessarily super-constant *(Munro et al., 2012)*.

Separable permutations:

- Avoid $2413, 3142$; built from sums/skew-sums – binary tree representation.
- Succinct data structure with $\mathcal{O}(1)$-time queries *(Chakraborty et al., 2024)*.

Baxter permutations:

- Defined via vincular patterns, in bijection with floorplans *(Ackerman et al., 2006)*, *(Felsner et al., 2011)*; also natural tree representation.
- Succinct data structure, but with polylogarithmic query time *(Chakraborty et al., 2024)*.

These techniques are specific to separable/Baxter permutations -- no such hierarchical structure is apparent for *arbitrary* pattern-avoiding permutations.

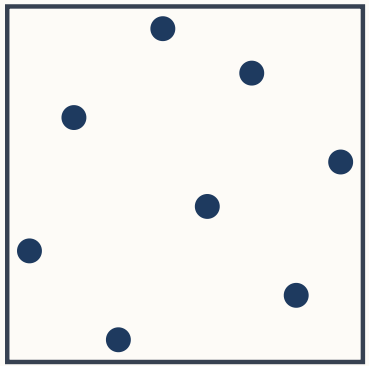
Near-optimal data structure

Theorem (*Kozma, O. 2025*)

Any π-avoiding permutation τ of size n can be represented in $2n \log s_\pi + o(n)$ bits, constructible in $\mathcal{O}(n \log s_\pi)$ time, supporting:

- $\tau(i)$ and $\tau^{-1}(j)$ queries in $\mathcal{O}(1)$ time;
- rectangle range minimum and rectangle range counting queries in $\mathcal{O}(\log \log n)$ time.

A priori knowledge of π is not required.



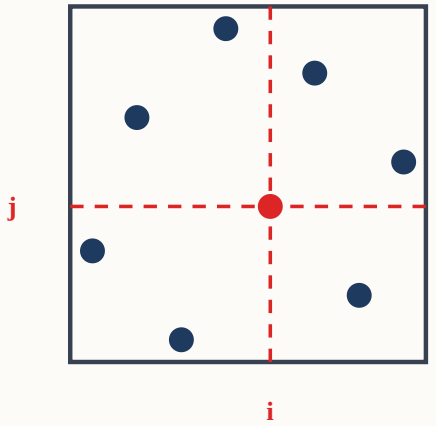
Near-optimal data structure

Theorem (*Kozma, O. 2025*)

Any π-avoiding permutation τ of size n can be represented in $2n \log s_\pi + o(n)$ bits, constructible in $\mathcal{O}(n \log s_\pi)$ time, supporting:

- $\tau(i)$ and $\tau^{-1}(j)$ queries in $\mathcal{O}(1)$ time;
- rectangle range minimum and rectangle range counting queries in $\mathcal{O}(\log \log n)$ time.

A priori knowledge of π is not required.



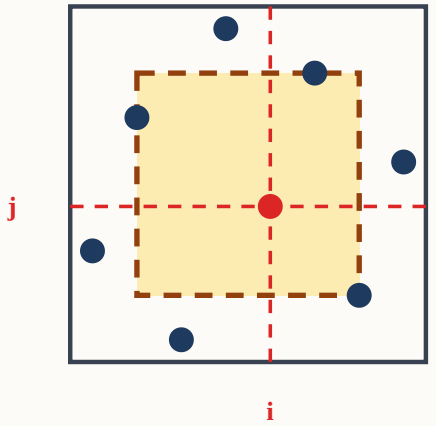
Near-optimal data structure

Theorem (*Kozma, O. 2025*)

Any π-avoiding permutation τ of size n can be represented in $2n \log s_\pi + o(n)$ bits, constructible in $\mathcal{O}(n \log s_\pi)$ time, supporting:

- $\tau(i)$ and $\tau^{-1}(j)$ queries in $\mathcal{O}(1)$ time;
- rectangle range minimum and rectangle range counting queries in $\mathcal{O}(\log \log n)$ time.

A priori knowledge of π is not required.



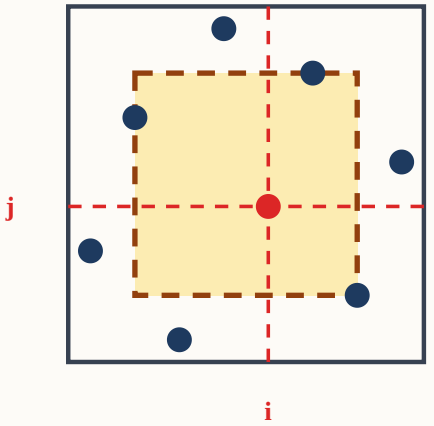
Near-optimal data structure

Theorem (*Kozma, O. 2025*)

Any π-avoiding permutation τ of size n can be represented in $2n \log s_\pi + o(n)$ bits, constructible in $\mathcal{O}(n \log s_\pi)$ time, supporting:

- $\tau(i)$ and $\tau^{-1}(j)$ queries in $\mathcal{O}(1)$ time;
- rectangle range minimum and rectangle range counting queries in $\mathcal{O}(\log \log n)$ time.

A priori knowledge of π is not required.

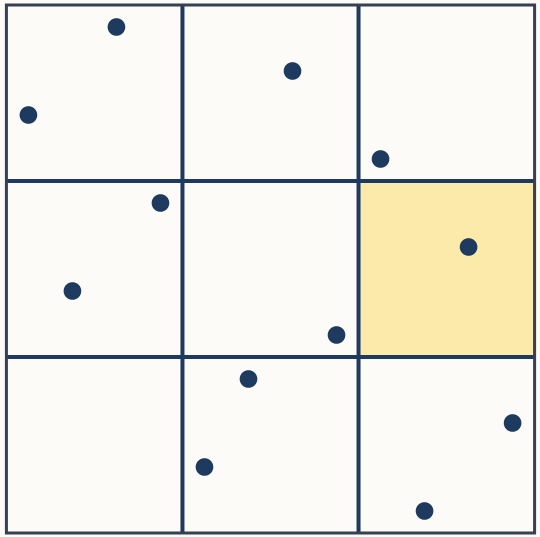


- Space matches the information-theoretic lower bound up to the multiplicative factor of 2.
- Generalizes sorting: construction in $\mathcal{O}(n \log s_\pi)$ time, then output $\tau^{-1}(1), \ldots, \tau^{-1}(n)$ in $\mathcal{O}(n)$ time.

Divisions

Partition the rows and columns of a permutation matrix M into contiguous intervals $\mathcal{R} = (I_1, \ldots, I_k)$ and $\mathcal{C} = (J_1, \ldots, J_{k'})$ – a **division** of M.

- The **cell** (i, j) is the submatrix of M induced by rows in I_i and columns in I_j.
- A cell is **non-zero** if it contains a 1-entry.
- $M(\mathcal{R}, \mathcal{C})$ is the $k \times k'$ 0-1 matrix with a 1 in cell (i, j) iff that cell is non-zero.
- Division $(\mathcal{R}', \mathcal{C}')$ is a **coarsening** of $(\mathcal{R}, \mathcal{C})$ – and $(\mathcal{R}, \mathcal{C})$ a **refinement** of $(\mathcal{R}', \mathcal{C}')$ – if $\mathcal{R}', \mathcal{C}'$ are obtained from $\mathcal{R}, \mathcal{C}$ by a sequence of merging neighboring intervals.



$M[I_2, J_3]$ highlighted, a non-zero cell of $M(\mathcal{R}, \mathcal{C})$

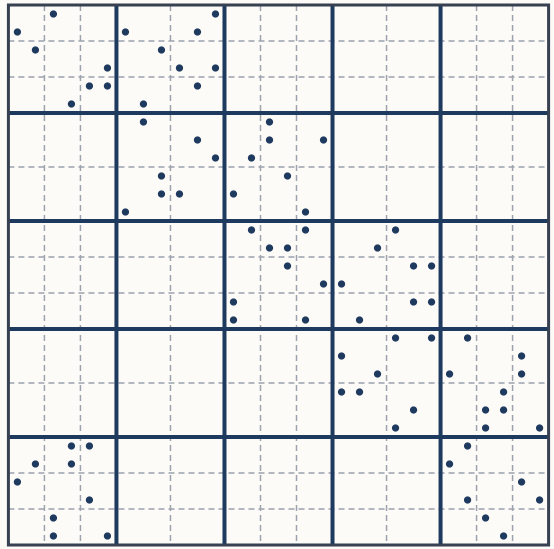
Structural decomposition

Theorem ***(Balanced decomposition)***

For $m_1 = n/\log^2 n$ and $m_2 = n/\sqrt{\log n}$, every π-avoiding $n \times n$ permutation matrix M admits divisions $(\mathcal{R}_1, \mathcal{C}_1)$, $(\mathcal{R}_2, \mathcal{C}_2)$ such that:

- $(\mathcal{R}_1, \mathcal{C}_1)$ coarsens $(\mathcal{R}_2, \mathcal{C}_2)$,
- $M(\mathcal{R}_i, \mathcal{C}_i)$ has exactly m_i rows/columns,
- each row/column of $M(\mathcal{R}_i, \mathcal{C}_i)$ has $\mathcal{O}(c_\pi)$ non-zero cells, and
- height/width of $\mathcal{R}_i, \mathcal{C}_i$ rows/cols is $\mathcal{O}(\log^2 n)$ ($i = 1$), resp. $\mathcal{O}(\sqrt{\log n})$ ($i = 2$),
- each row/column of $\mathcal{R}_1, \mathcal{C}_1$ merges at most $\mathcal{O}(\log^{1.5} n)$ rows/columns of $\mathcal{R}_2, \mathcal{C}_2$.

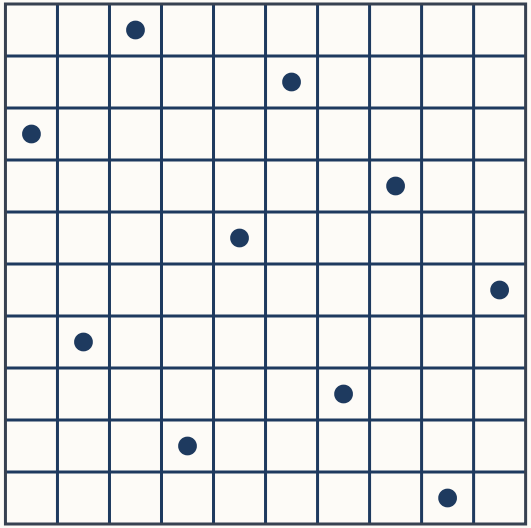
Moreover, $(\mathcal{R}_1, \mathcal{C}_1)$, $(\mathcal{R}_2, \mathcal{C}_2)$ can be constructed in $\mathcal{O}(\log s_\pi \cdot n)$ time.



Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

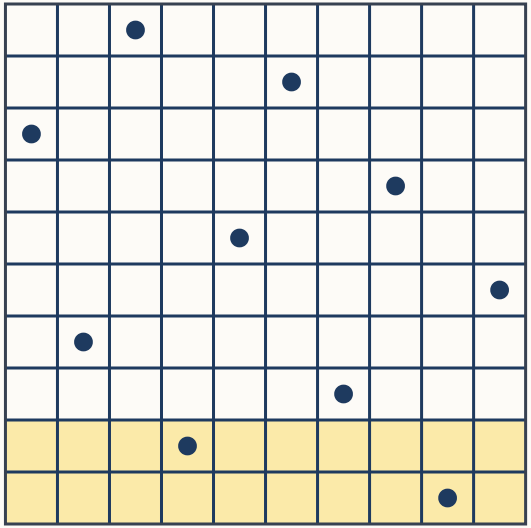


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

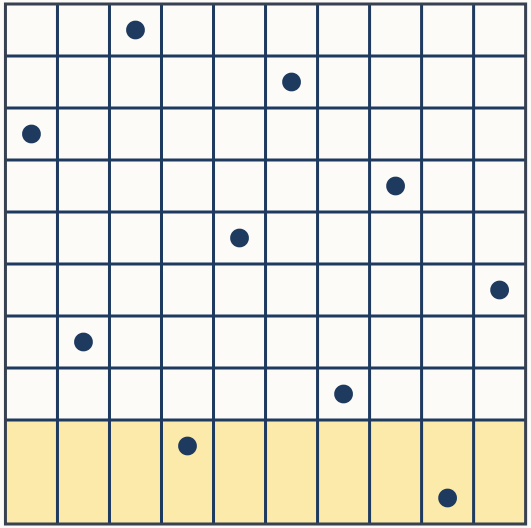


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

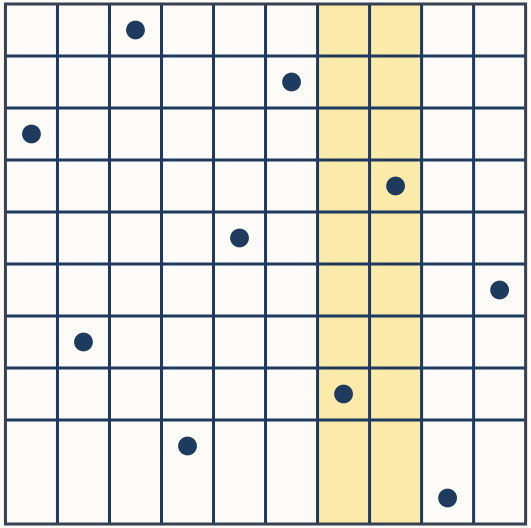


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

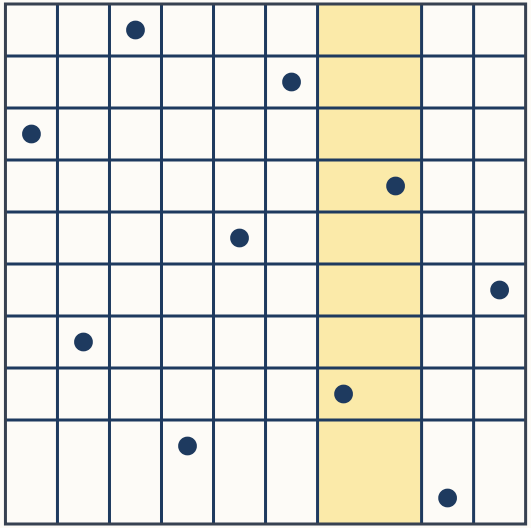


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

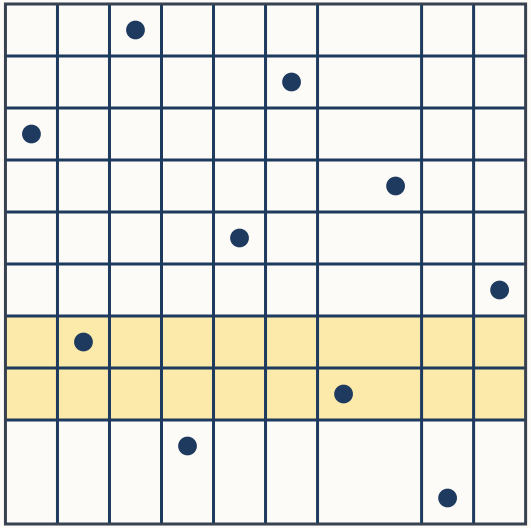


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

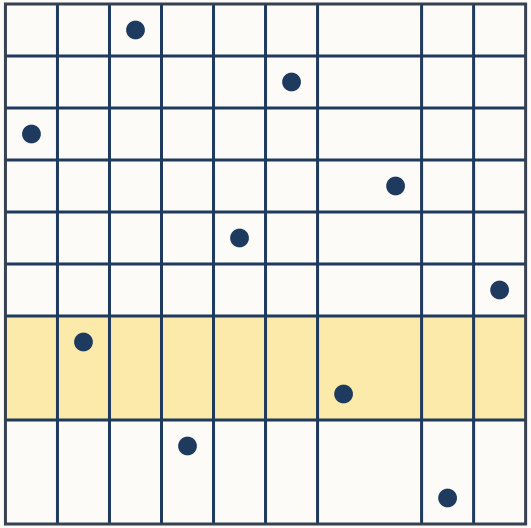


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

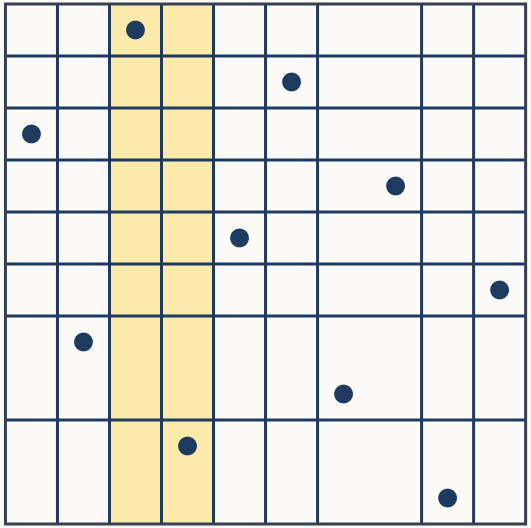


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

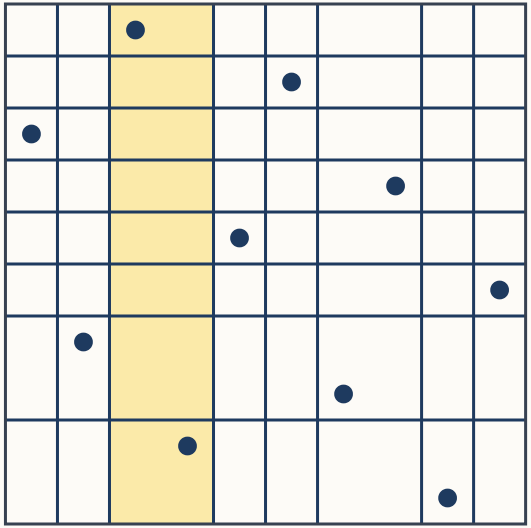


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

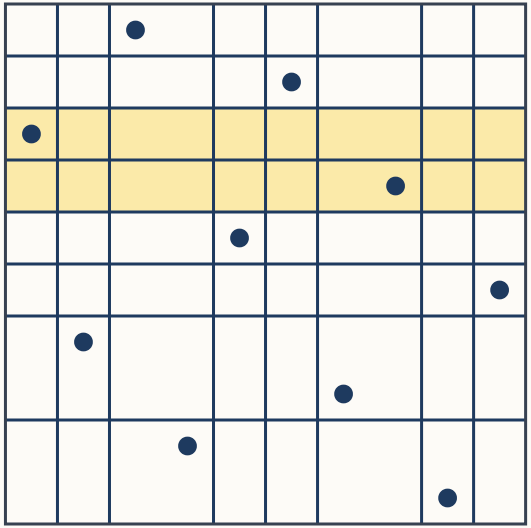


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

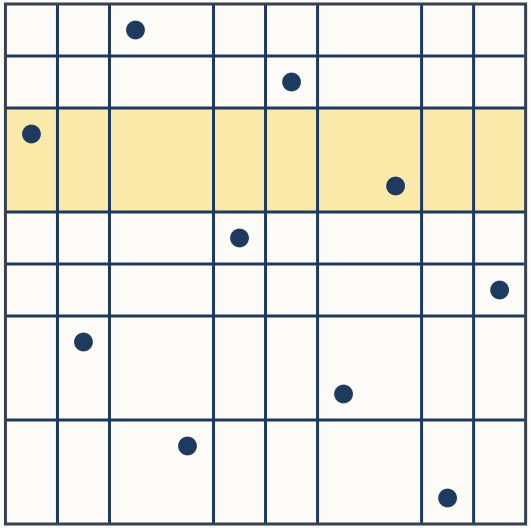


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

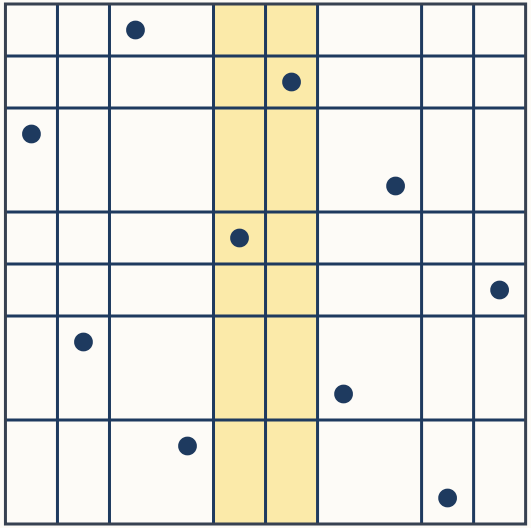


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

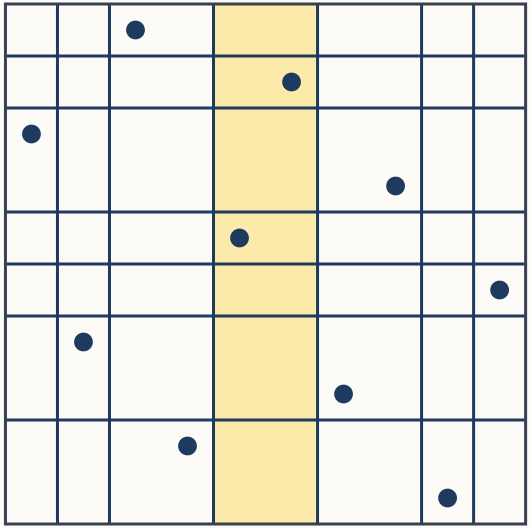


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

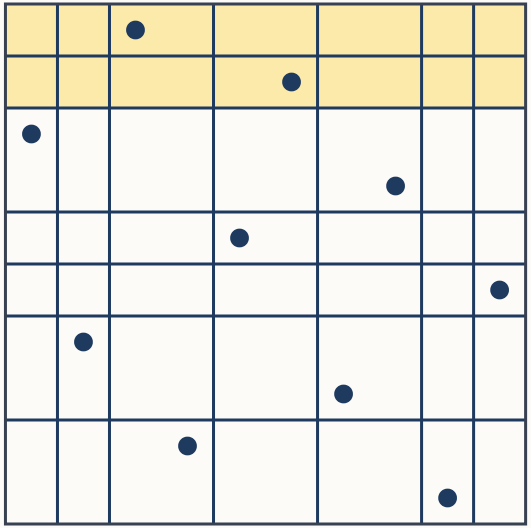


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

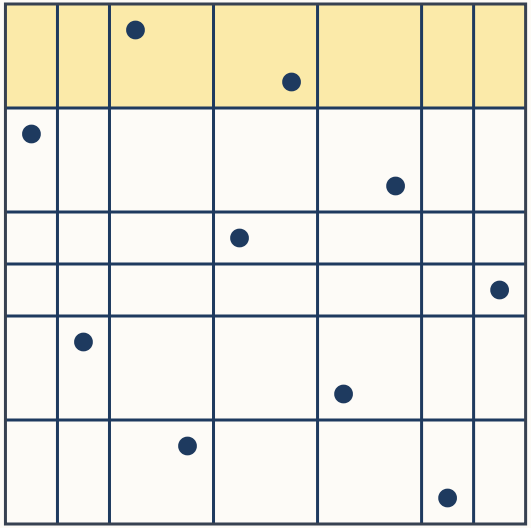


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

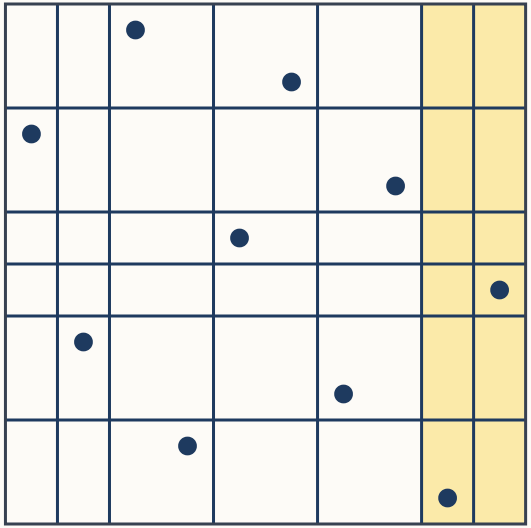


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

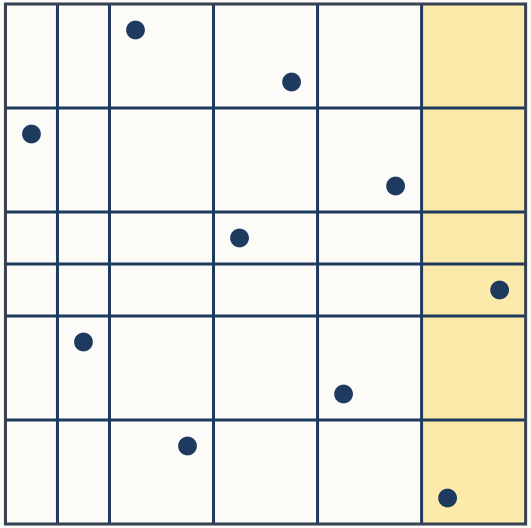


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

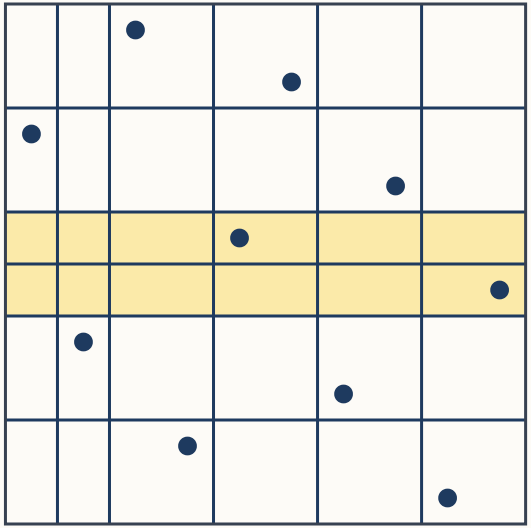


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

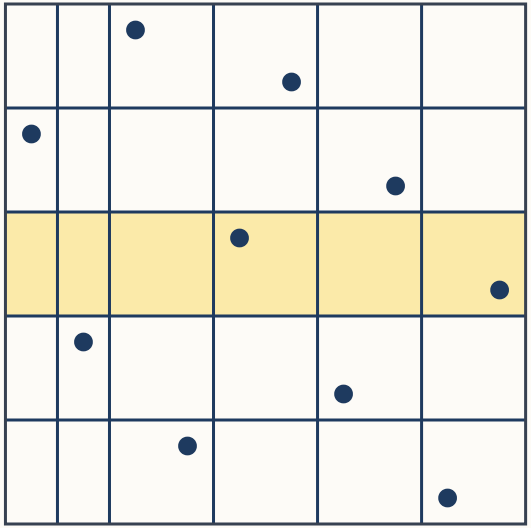


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

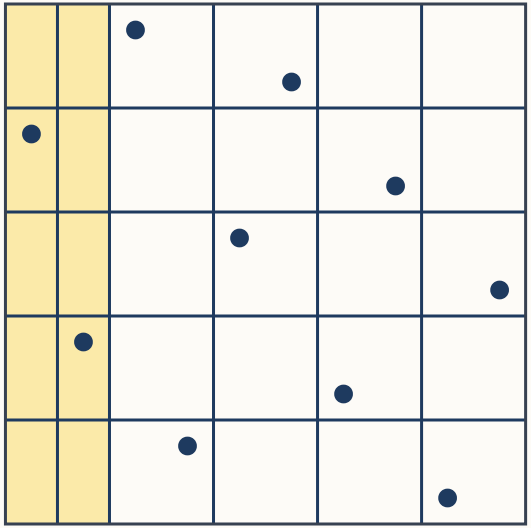


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

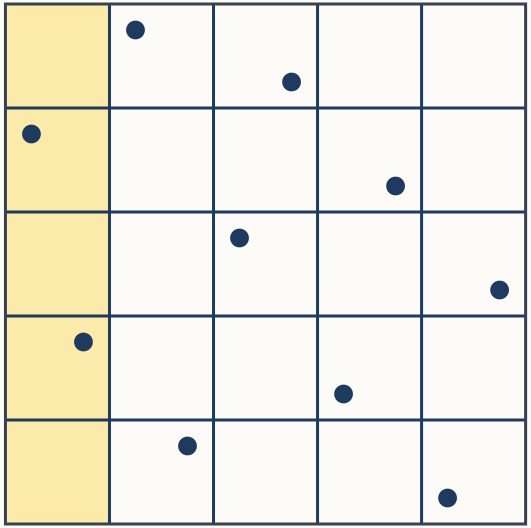


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

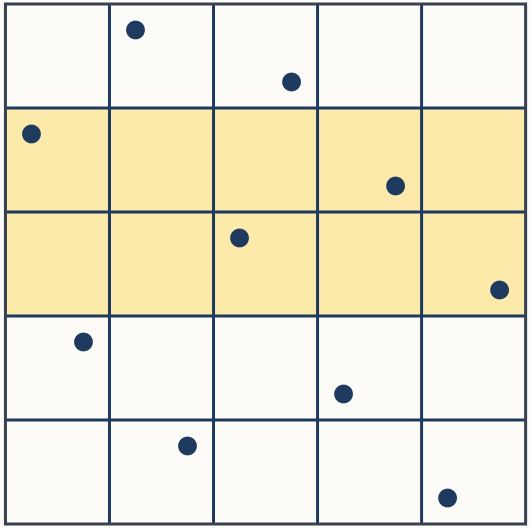


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

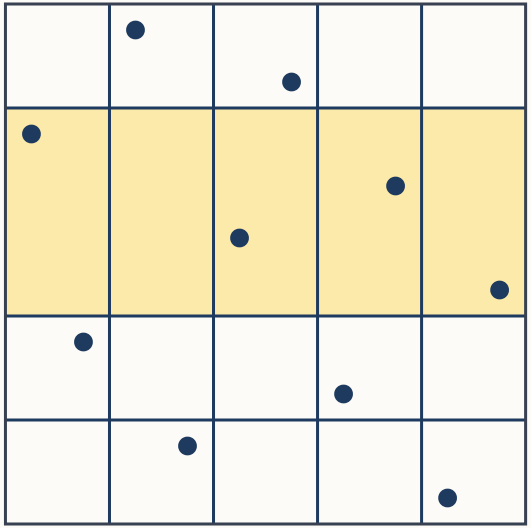


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

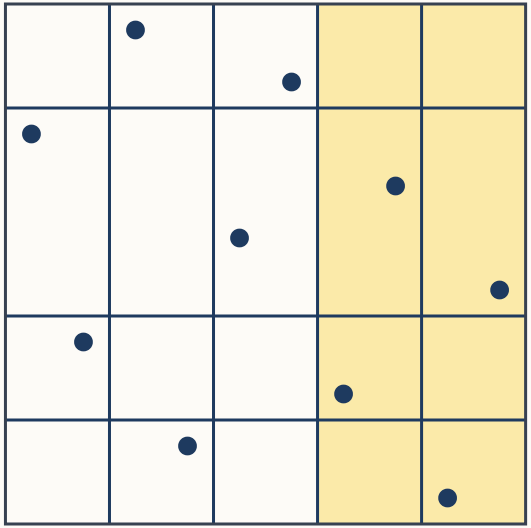


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

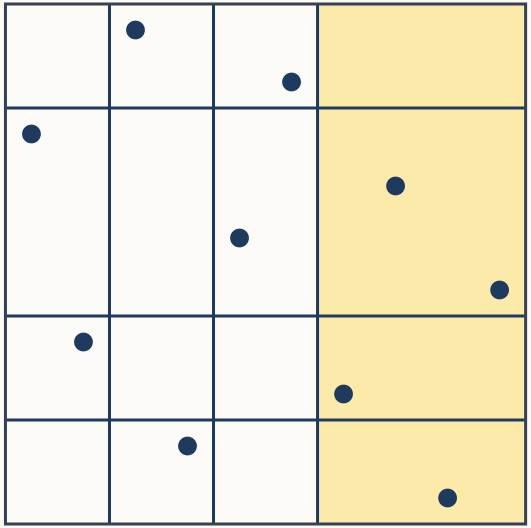


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

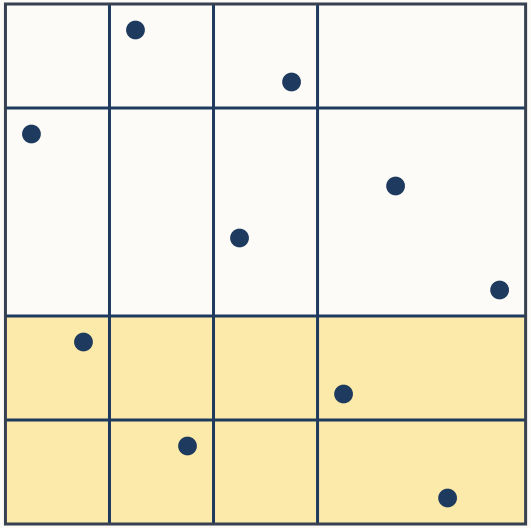


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

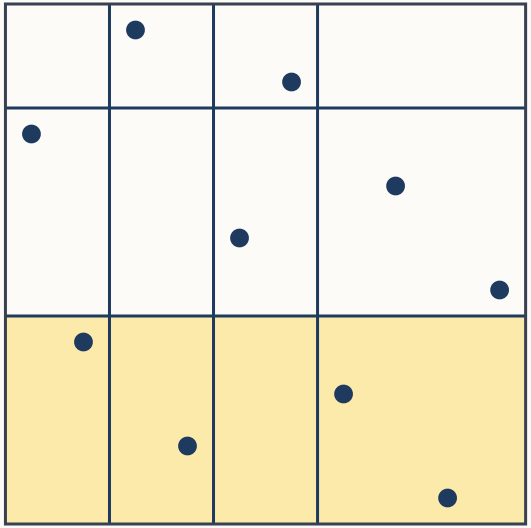


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

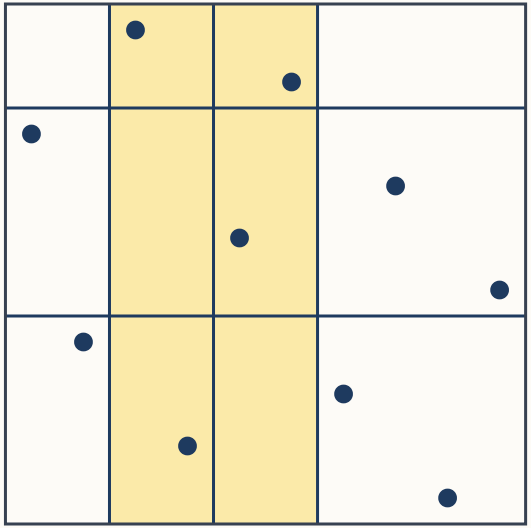


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

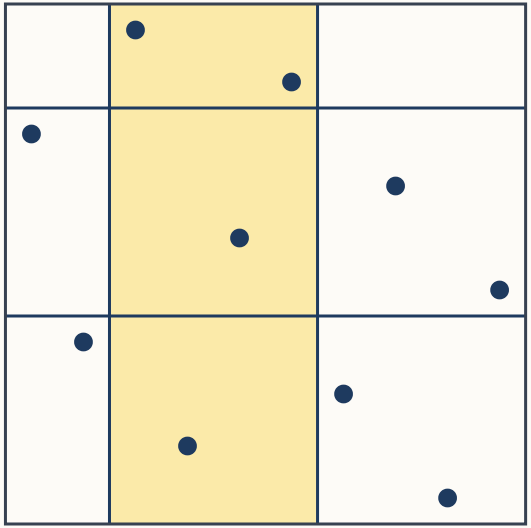


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

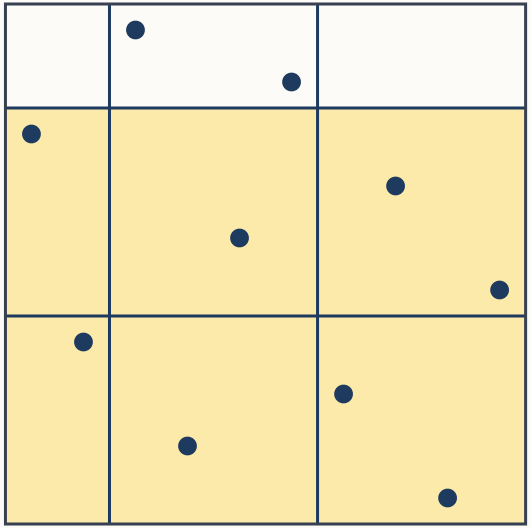


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

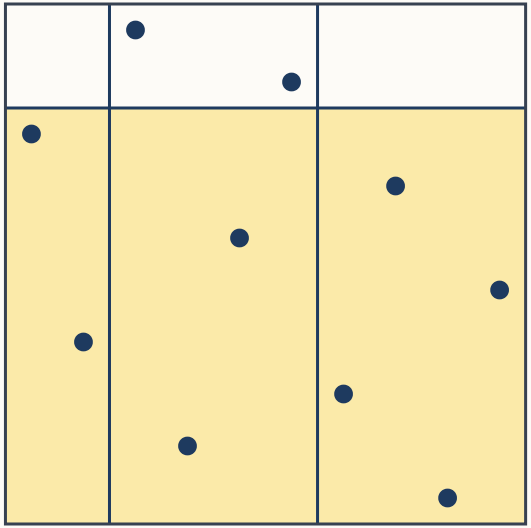


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

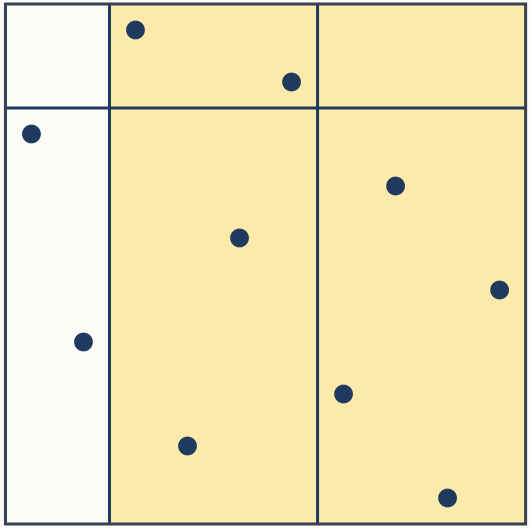


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

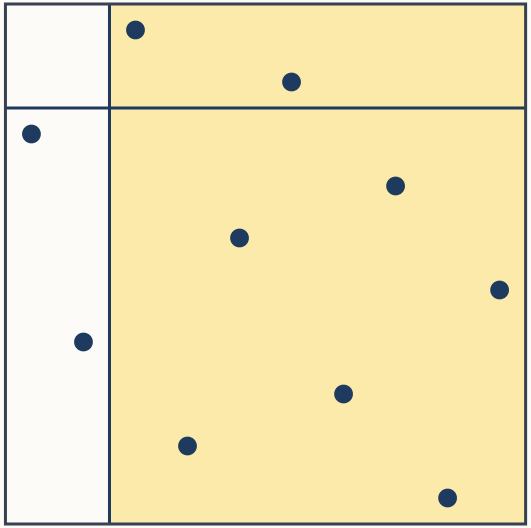


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

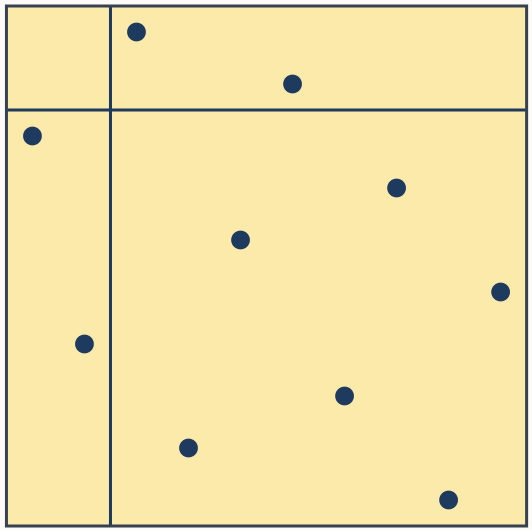


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

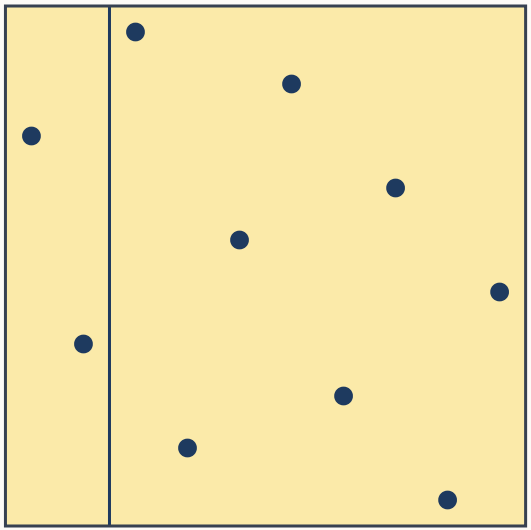


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

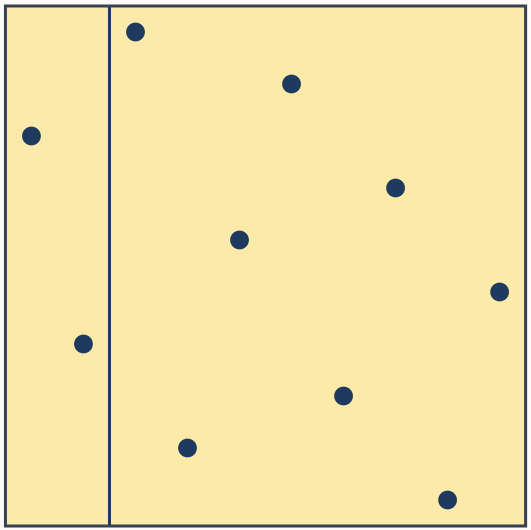


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.

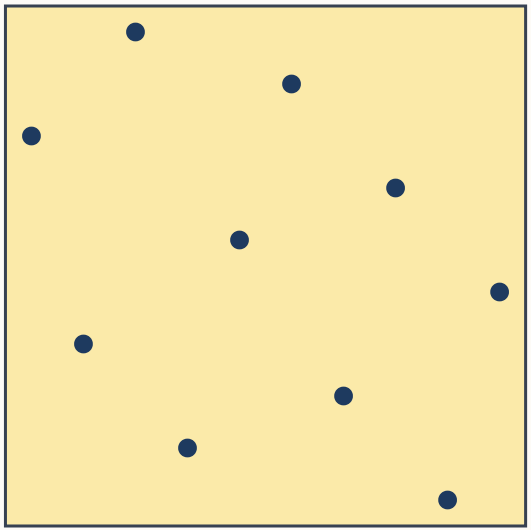


Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

Greedy construction

Ignoring the height/width and branching conditions, a division can be built **greedily**:

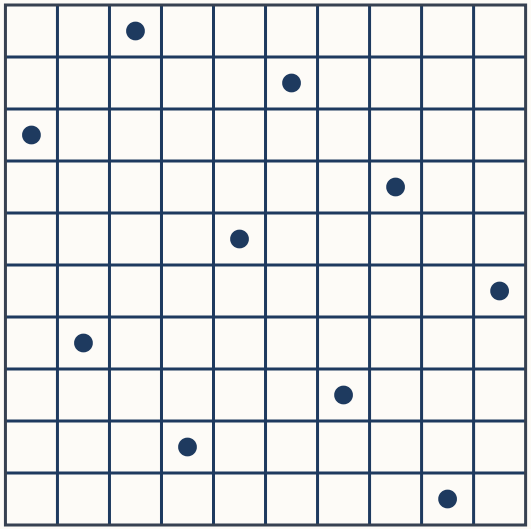
1. Set $d := 2$
2. Initialize with the trivial (finest) division.
3. While more than 1 row or column:
4. Merge adjacent rows with at most d non-zero cells.
5. Merge adjacent columns with at most d non-zero cells.
6. Save the division if its dimensions are exactly m_1 or m_2.
7. If any of the merges was not possible: double d.



Similar decomposition (with d fixed) was first used by Guillemot and Marx *(Guillemot & Marx, 2014)*.

When do we have to double the parameter?

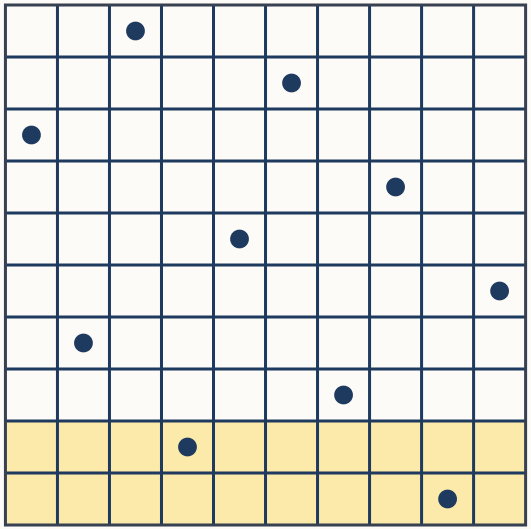
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

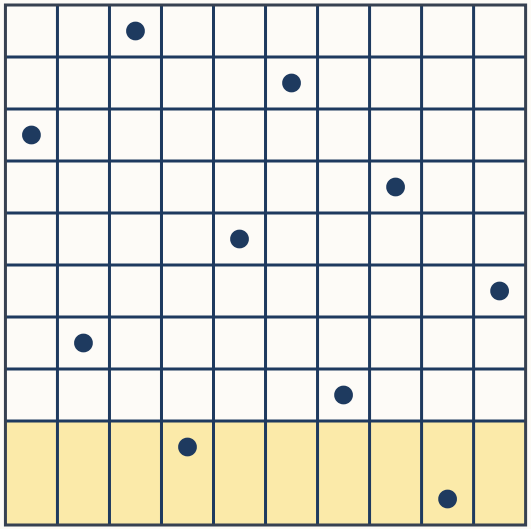
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

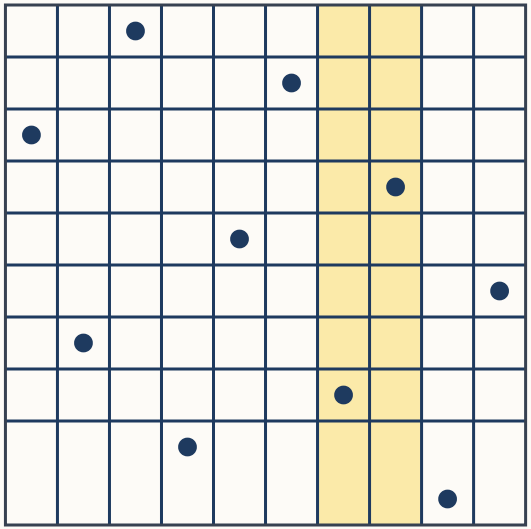
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

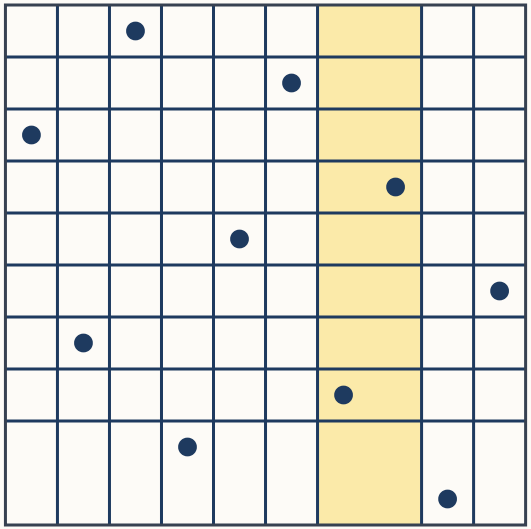
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

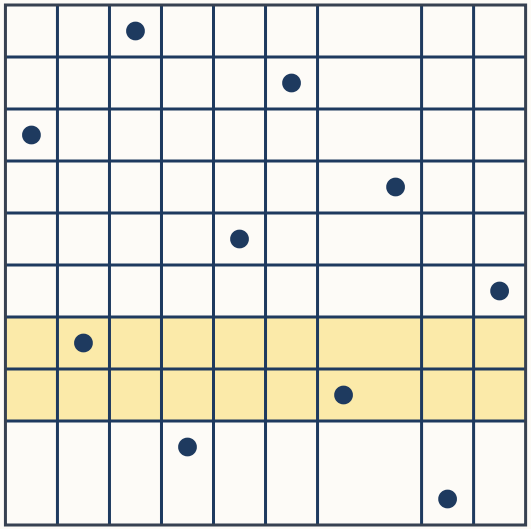
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

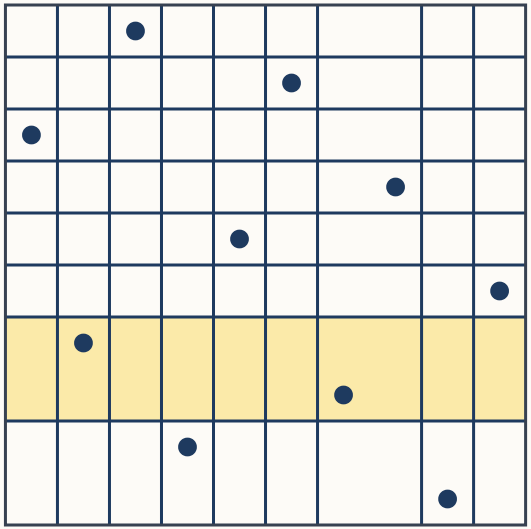
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

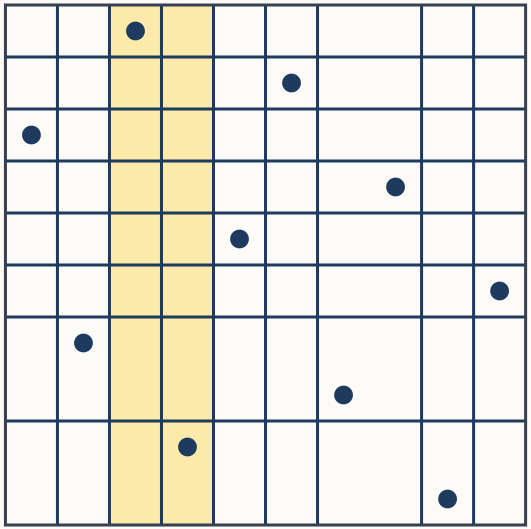
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

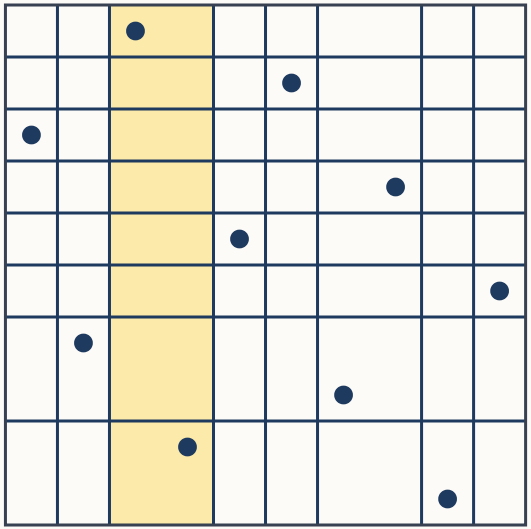
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

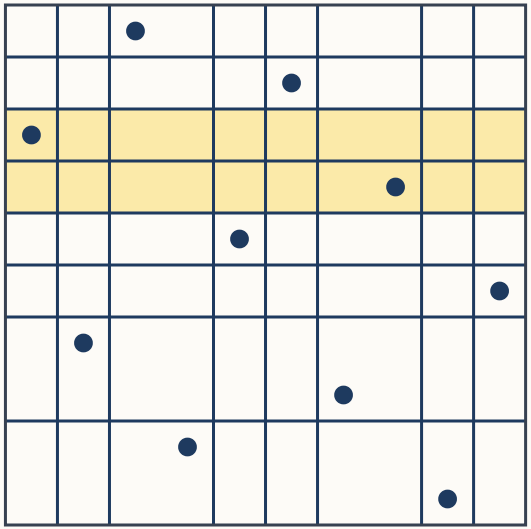
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

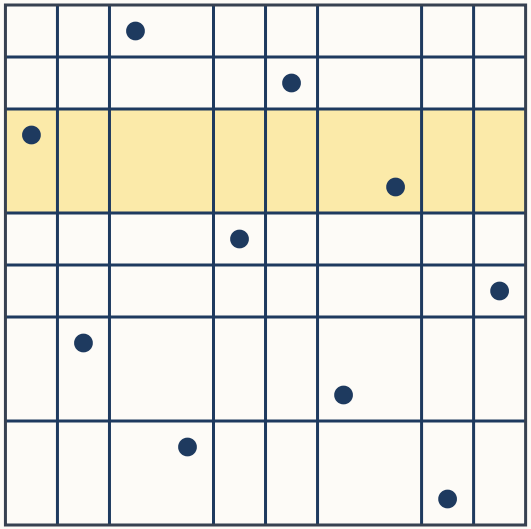
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

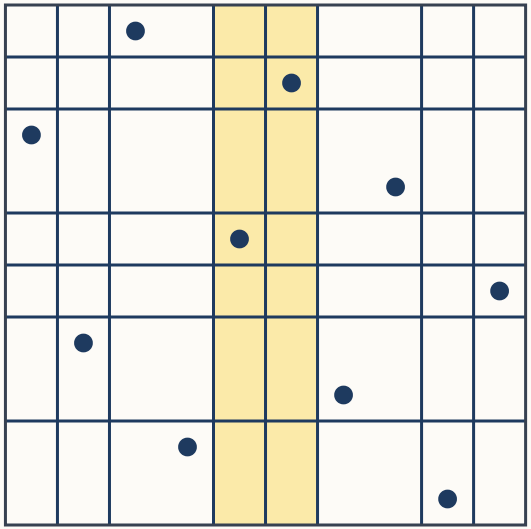
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

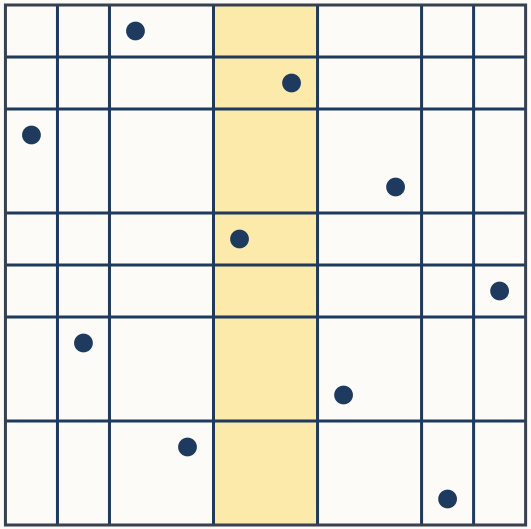
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

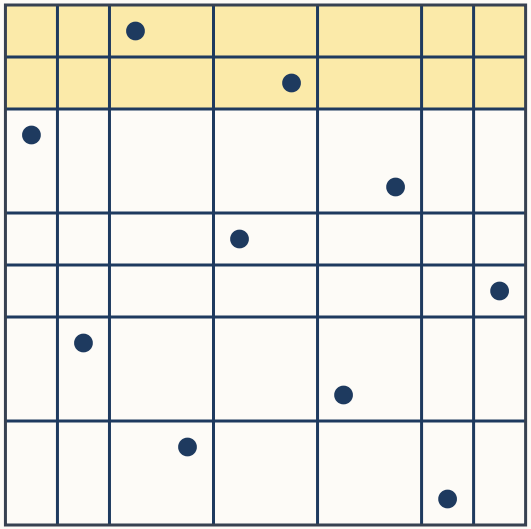
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

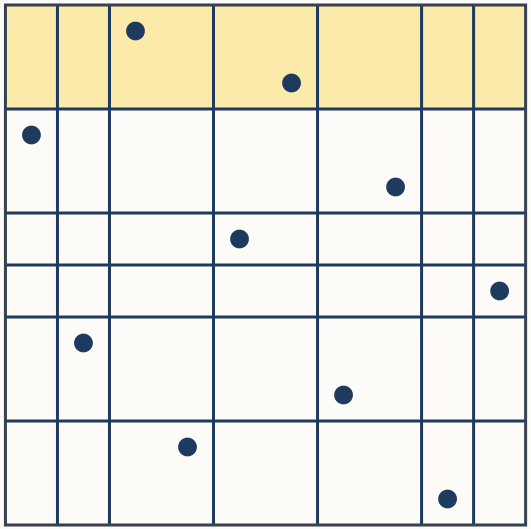
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

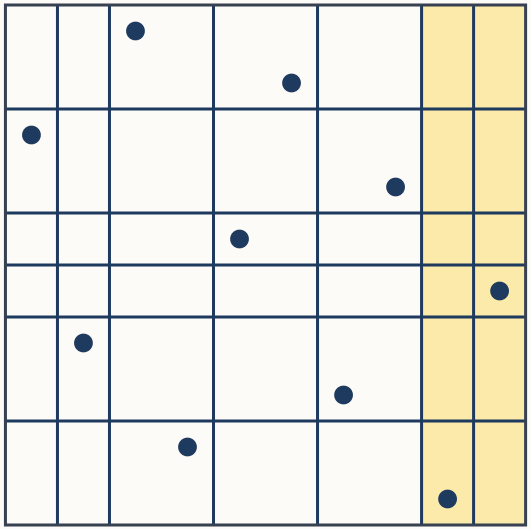
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

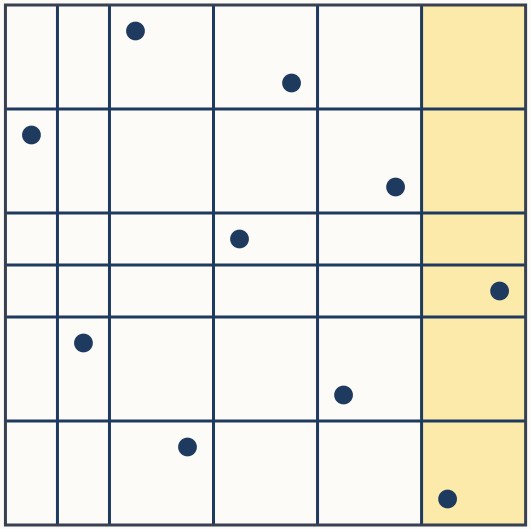
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

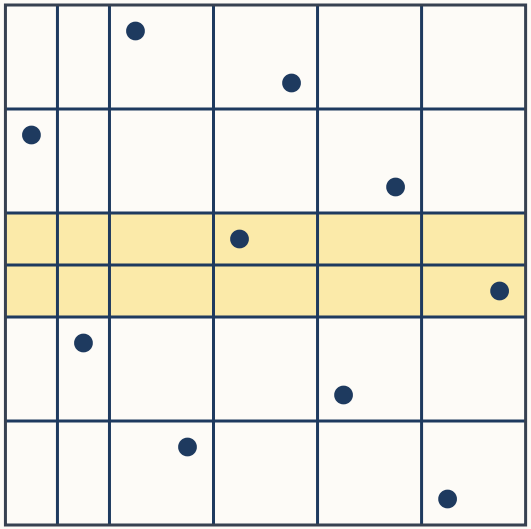
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

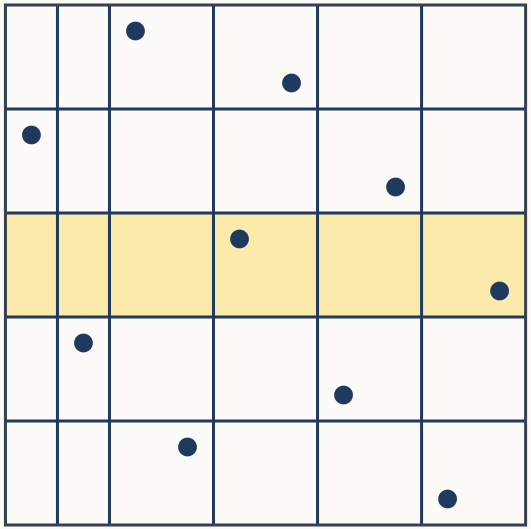
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

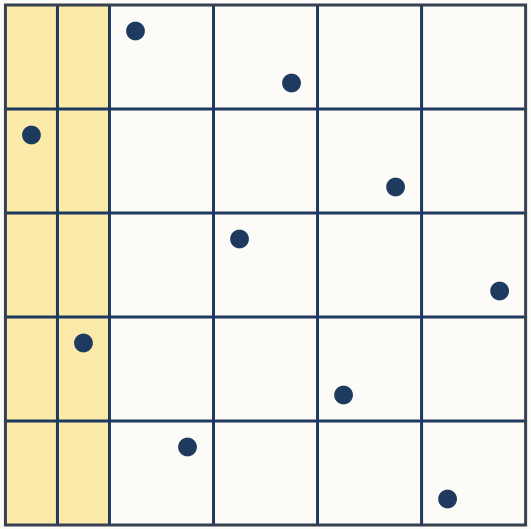
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

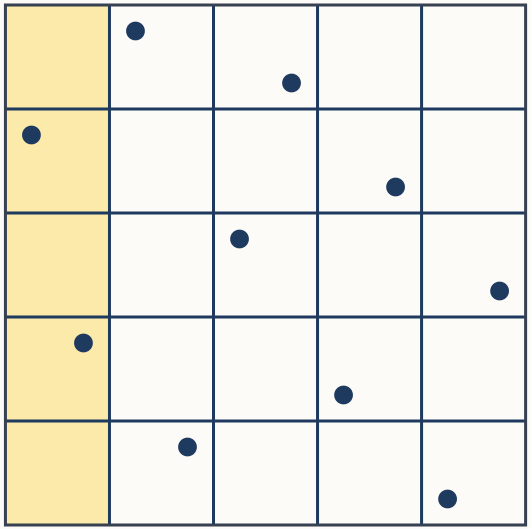
Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

Claim: The process never reaches $d > 8c_\pi$.



d = 2 gets stuck at division of side length 5

When do we have to double the parameter?

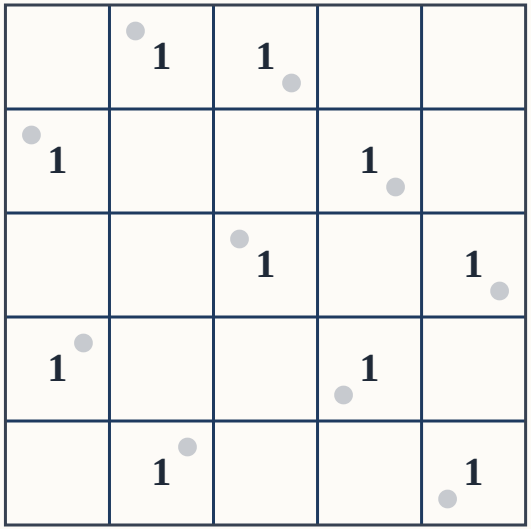
Claim: The process never reaches $d > 8c_\pi$.

Proof:

- Assume for a contradiction that the last doubling happens when $d > 4c_\pi$.
- Let m be the number of rows (and columns) of the current division $(\mathcal{R}, \mathcal{C})$.
- Since the process cannot continue with current d, the number of non-zero entries in $M(\mathcal{R}, \mathcal{C})$ is more than

$$\left\lfloor \frac{m}{2} \right\rfloor \cdot d \geq \frac{m-1}{2} \cdot d \geq c_\pi \cdot m$$

- But then $M(\mathcal{C}, \mathcal{R})$ contains π and so does the input permutation. ■

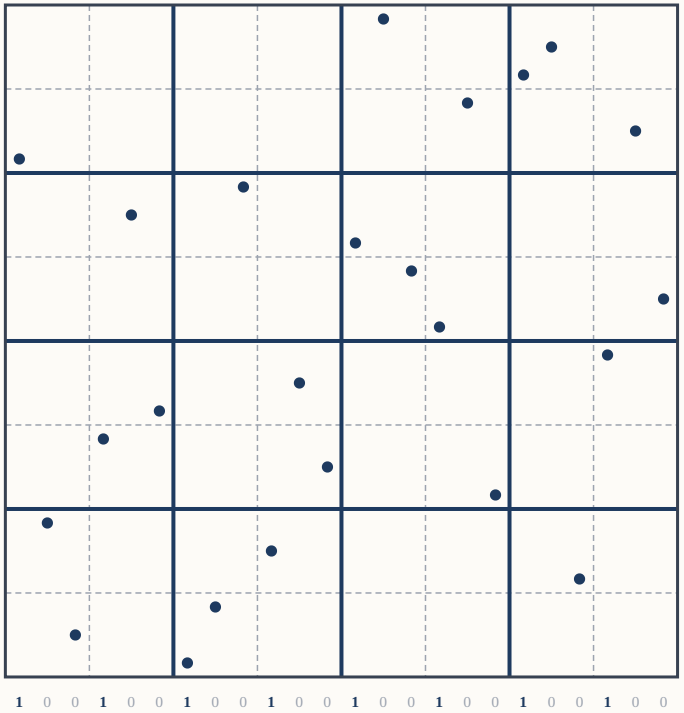


d = 2 gets stuck at division of side length 5

Locating fine column

First we locate the fine column containing i:

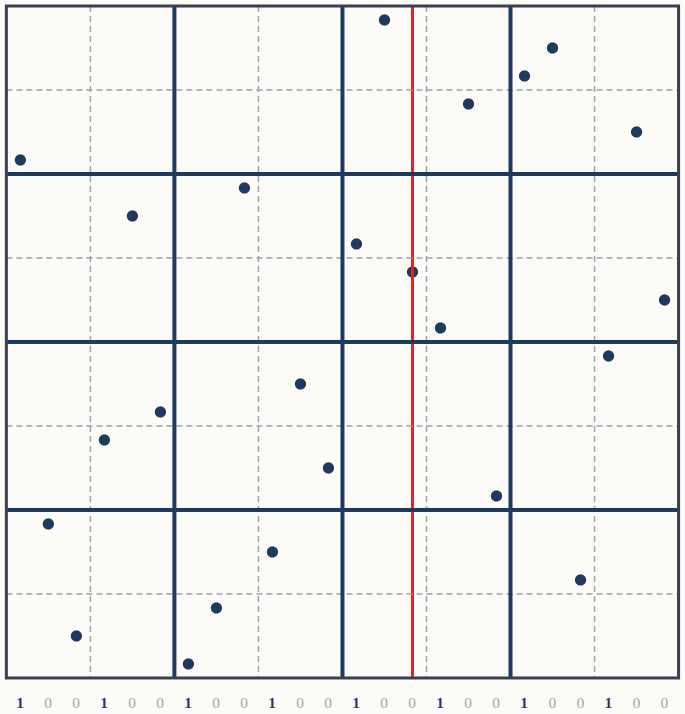
- Store a bitvector marking the start of every fine column.
- The number of 1s up to index i gives precisely the rank of the fine column containing i.
- The position of closest previous 1 gives the start of the fine column, hence the offset within.
- Classical Rank and Select problem with succinct implementations.
- Here, the number of 1s is $O(n/\sqrt{\log n}) = o(n)$ and compressed implementation with $\mathcal{O}(1)$ queries in $o(n)$ space exists.



Locating fine column

First we locate the fine column containing i:

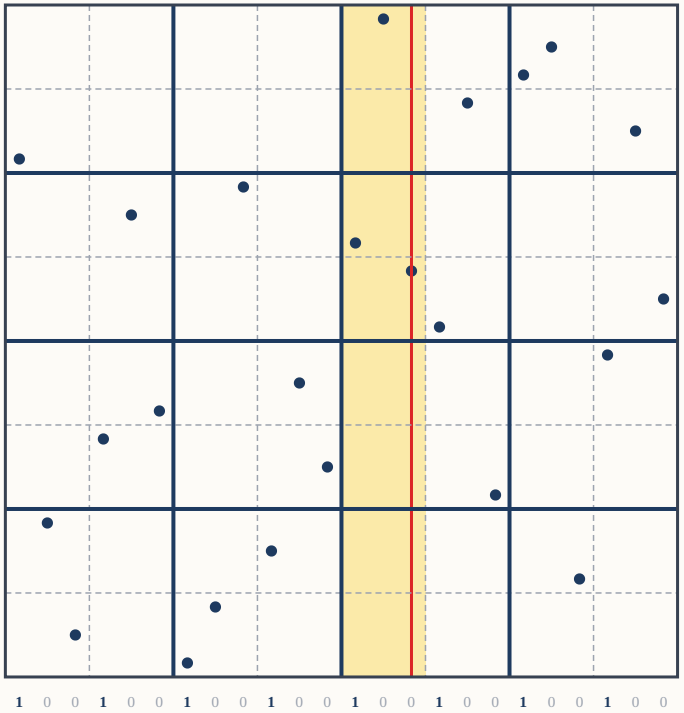
- Store a bitvector marking the start of every fine column.
- The number of 1s up to index i gives precisely the rank of the fine column containing i.
- The position of closest previous 1 gives the start of the fine column, hence the offset within.
- Classical Rank and Select problem with succinct implementations.
- Here, the number of 1s is $O(n/\sqrt{\log n}) = o(n)$ and compressed implementation with $\mathcal{O}(1)$ queries in $o(n)$ space exists.



Locating fine column

First we locate the fine column containing i:

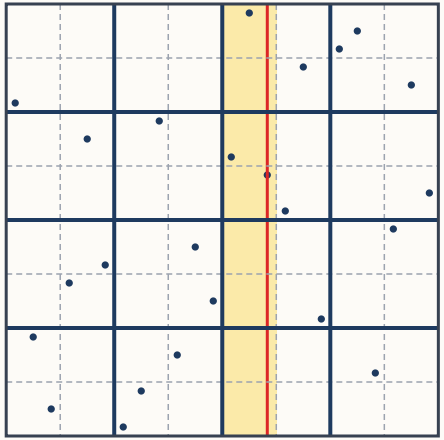
- Store a bitvector marking the start of every fine column.
- The number of 1s up to index i gives precisely the rank of the fine column containing i.
- The position of closest previous 1 gives the start of the fine column, hence the offset within.
- Classical Rank and Select problem with succinct implementations.
- Here, the number of 1s is $O(n/\sqrt{\log n}) = o(n)$ and compressed implementation with $\mathcal{O}(1)$ queries in $o(n)$ space exists.



Column permutation

Forgetting the empty rows, the fine column collapses to a short permutation with $\mathcal{O}(c_\pi)$ cells:

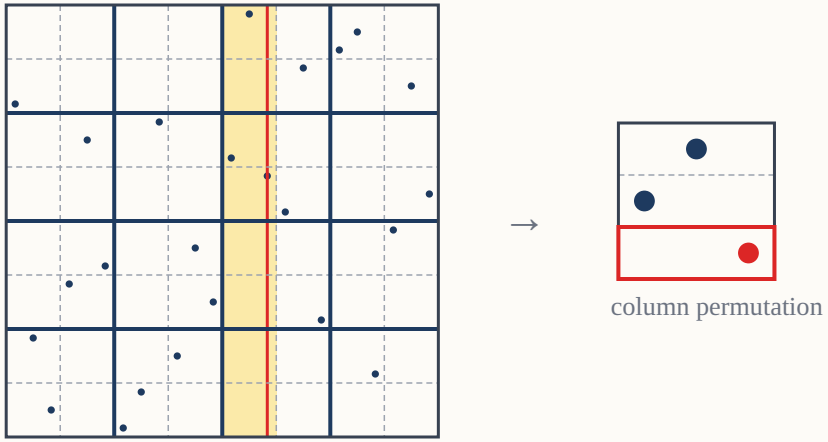
- Store precomputed info per shape – at most s_π^w many, for column width $w = \mathcal{O}(\sqrt{\log n})$ which makes $o(n)$ space in total.
- Each fine column carries only index into this precomputed table.
- We obtain cell rank and rank of the queried point within the cell in $\mathcal{O}(1)$.



Column permutation

Forgetting the empty rows, the fine column collapses to a short permutation with $\mathcal{O}(c_\pi)$ cells:

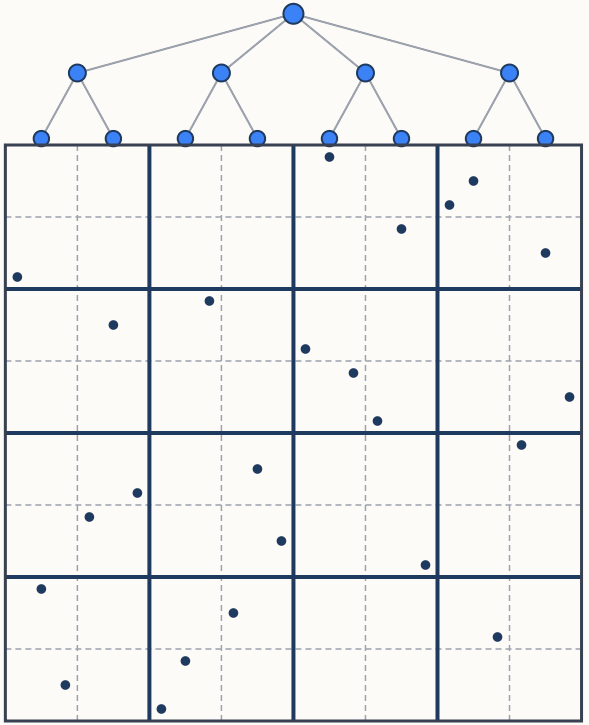
- Store precomputed info per shape – at most s_π^w many, for column width $w = \mathcal{O}(\sqrt{\log n})$ which makes $o(n)$ space in total.
- Each fine column carries only index into this precomputed table.
- We obtain cell rank and rank of the queried point within the cell in $\mathcal{O}(1)$.



Column navigation

Next, we navigate a tree that captures column merges from the located leaf to the root:

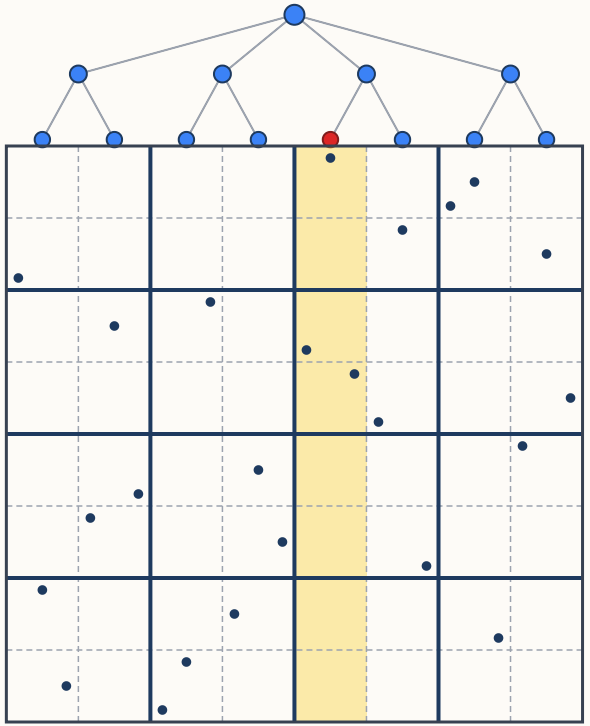
- Using succinct representation, this takes $\mathcal{O}(1)$ time in $O(n/\sqrt{\log n}) = o(n)$ space.
- We obtain the coarse column containing the queried point.
- We can also store a bit more data to obtain cell ranks – here the point lies in the lowest non-zero cell of the coarse division.



Column navigation

Next, we navigate a tree that captures column merges from the located leaf to the root:

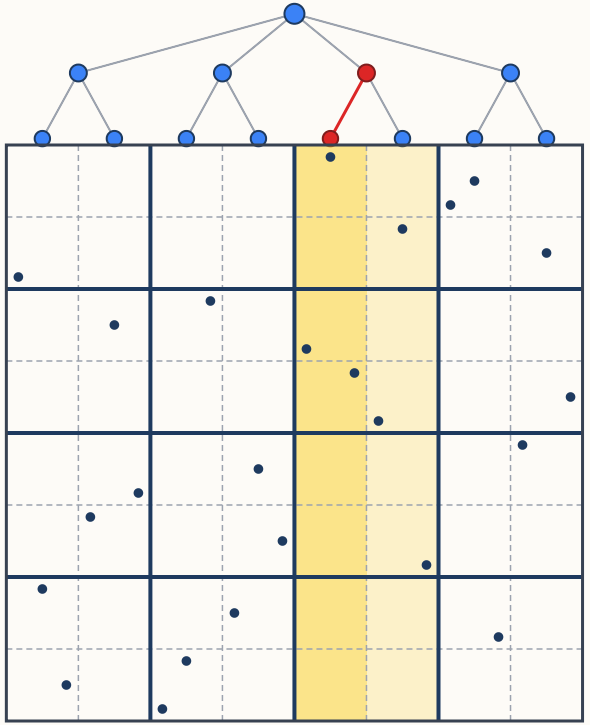
- Using succinct representation, this takes $\mathcal{O}(1)$ time in $O(n/\sqrt{\log n}) = o(n)$ space.
- We obtain the coarse column containing the queried point.
- We can also store a bit more data to obtain cell ranks – here the point lies in the lowest non-zero cell of the coarse division.



Column navigation

Next, we navigate a tree that captures column merges from the located leaf to the root:

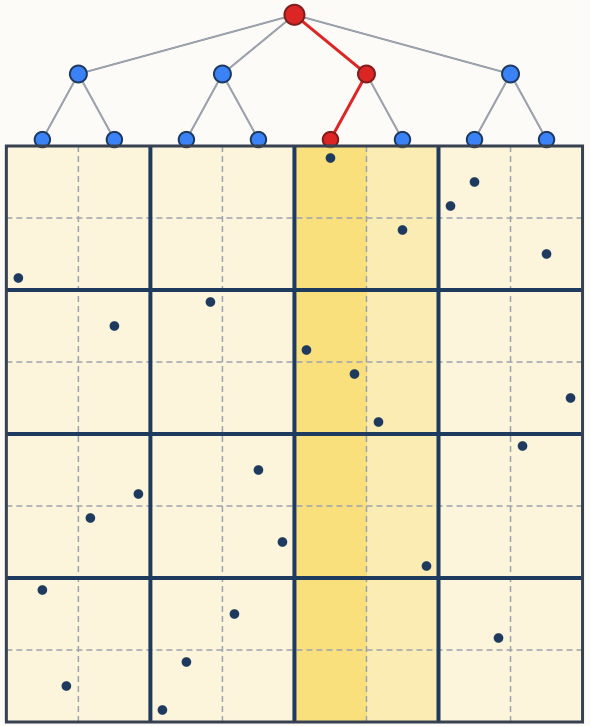
- Using succinct representation, this takes $\mathcal{O}(1)$ time in $O(n/\sqrt{\log n}) = o(n)$ space.
- We obtain the coarse column containing the queried point.
- We can also store a bit more data to obtain cell ranks – here the point lies in the lowest non-zero cell of the coarse division.



Column navigation

Next, we navigate a tree that captures column merges from the located leaf to the root:

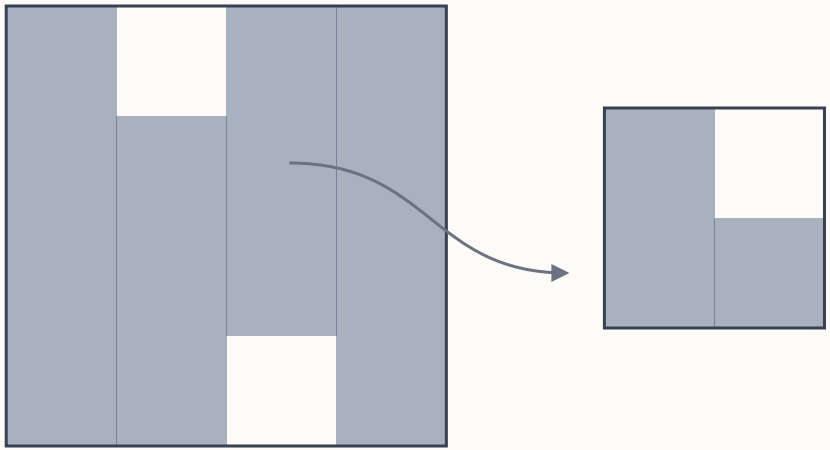
- Using succinct representation, this takes $\mathcal{O}(1)$ time in $O(n/\sqrt{\log n}) = o(n)$ space.
- We obtain the coarse column containing the queried point.
- We can also store a bit more data to obtain cell ranks – here the point lies in the lowest non-zero cell of the coarse division.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

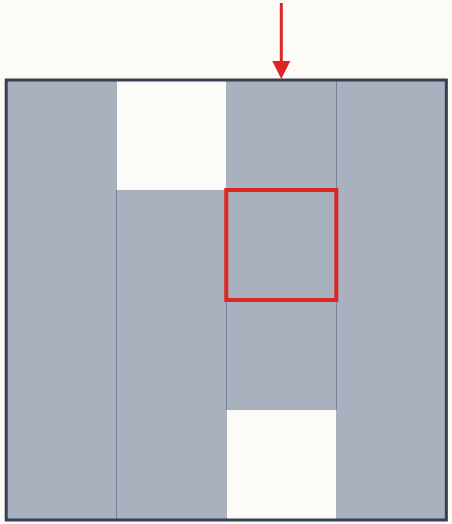
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

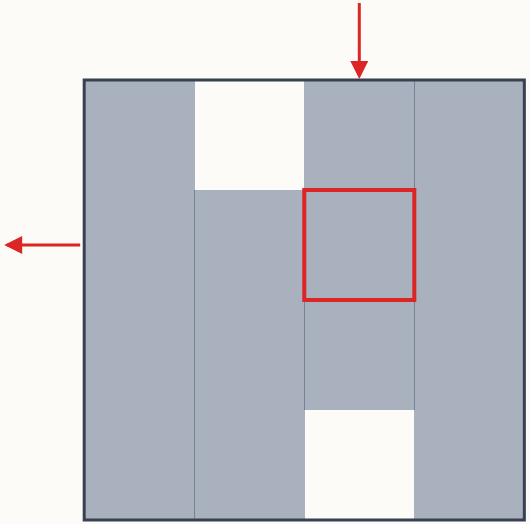
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

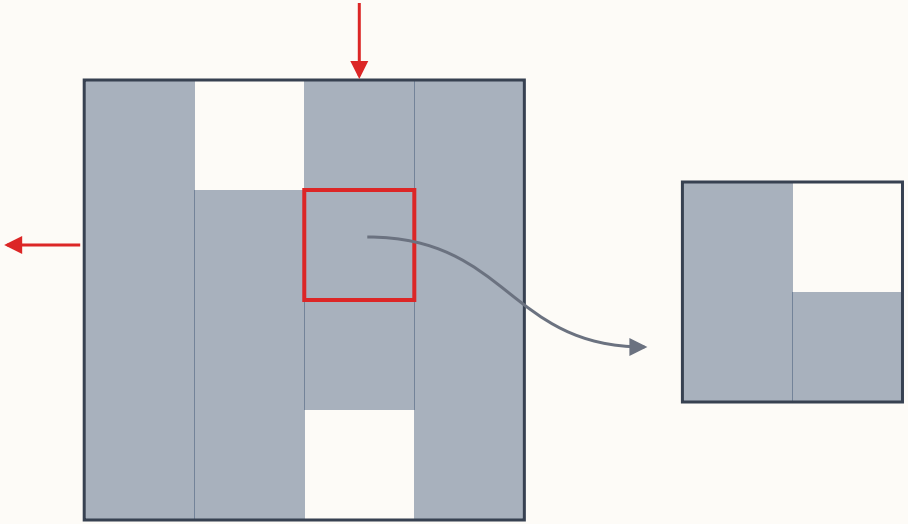
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

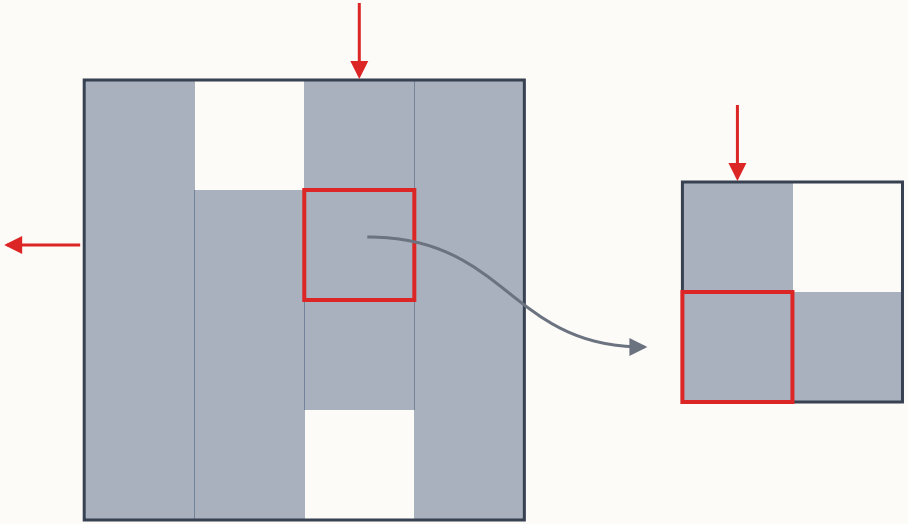
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

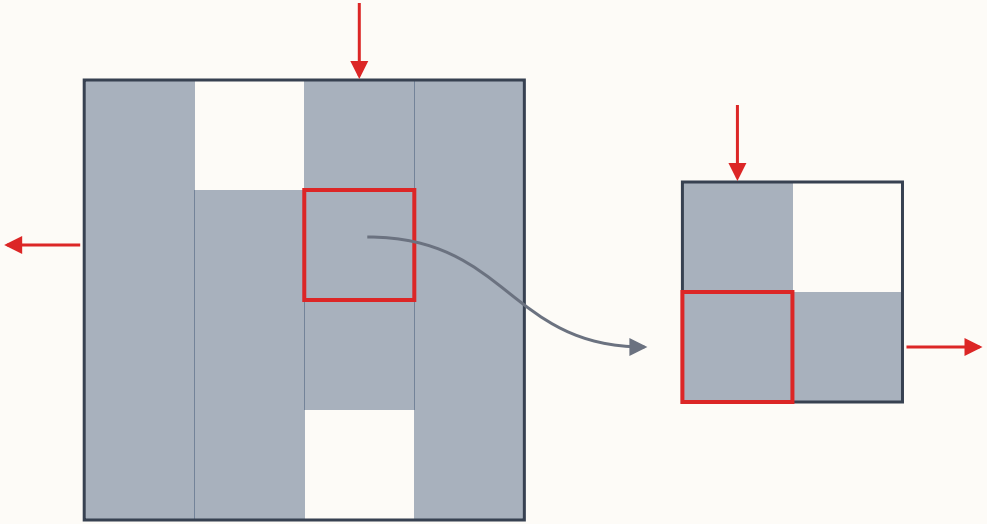
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Recursive cell structure

Each non-zero coarse cell hosts its own fine sub-structure:

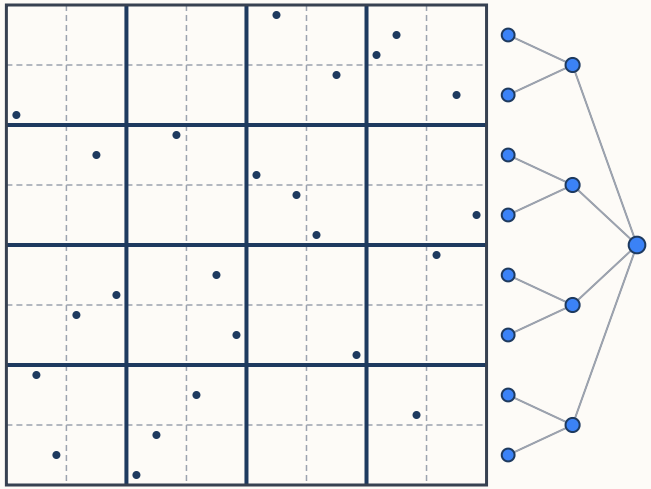
- $M(\mathcal{R}_1, \mathcal{C}_1)$ has $\mathcal{O}(c_\pi)$ non-zero cells per row/column.
- Zooming into one such cell reveals the fine columns (and rows) within it.
- This allows us to get row ranks and within row cell-ranks for both levels.



Row navigation

Symmetrically, we navigate a tree capturing row merges:

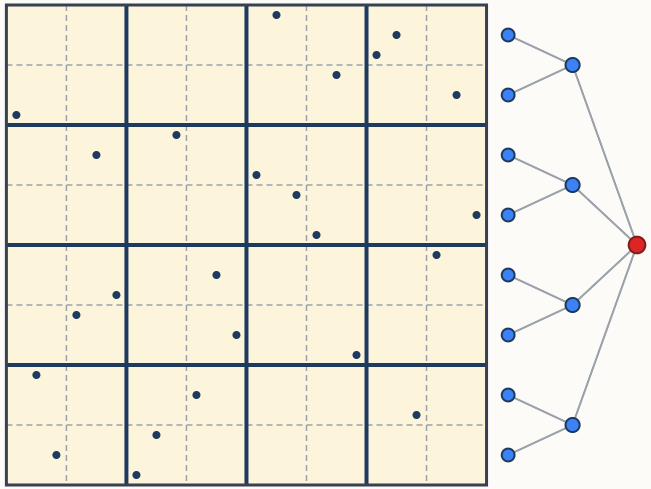
- Start at the root – the point lies somewhere in the matrix.
- The coarse cell rank (from the recursive structure) gives the coarse row.
- The fine cell rank within it gives the exact fine row.
- With succinct representation of trees, this takes $\mathcal{O}(1)$ time in $o(n)$ space.



Row navigation

Symmetrically, we navigate a tree capturing row merges:

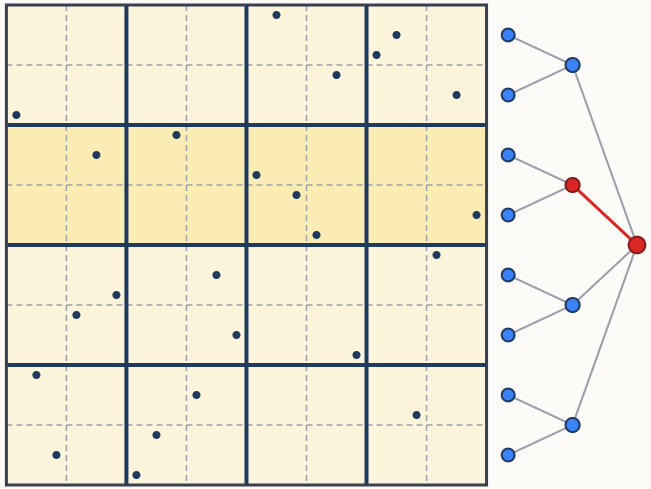
- Start at the root – the point lies somewhere in the matrix.
- The coarse cell rank (from the recursive structure) gives the coarse row.
- The fine cell rank within it gives the exact fine row.
- With succinct representation of trees, this takes $\mathcal{O}(1)$ time in $o(n)$ space.



Row navigation

Symmetrically, we navigate a tree capturing row merges:

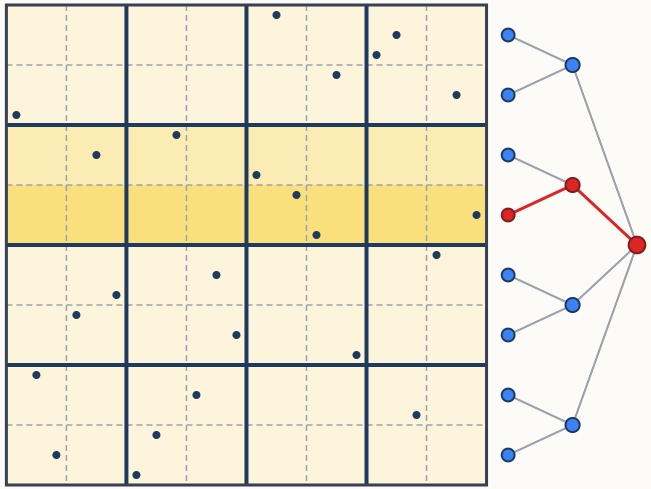
- Start at the root – the point lies somewhere in the matrix.
- The coarse cell rank (from the recursive structure) gives the coarse row.
- The fine cell rank within it gives the exact fine row.
- With succinct representation of trees, this takes $\mathcal{O}(1)$ time in $o(n)$ space.



Row navigation

Symmetrically, we navigate a tree capturing row merges:

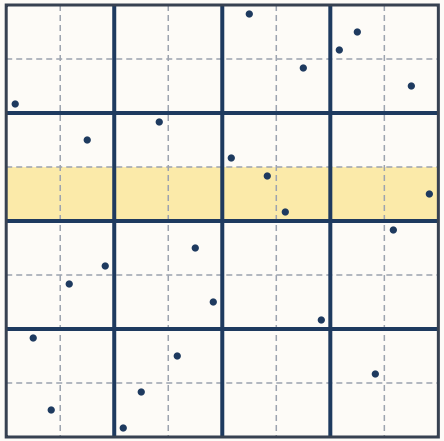
- Start at the root – the point lies somewhere in the matrix.
- The coarse cell rank (from the recursive structure) gives the coarse row.
- The fine cell rank within it gives the exact fine row.
- With succinct representation of trees, this takes $\mathcal{O}(1)$ time in $o(n)$ space.



Row permutation

Forgetting the empty columns, the fine row again collapses to a short permutation with $\mathcal{O}(c_\pi)$ cells:

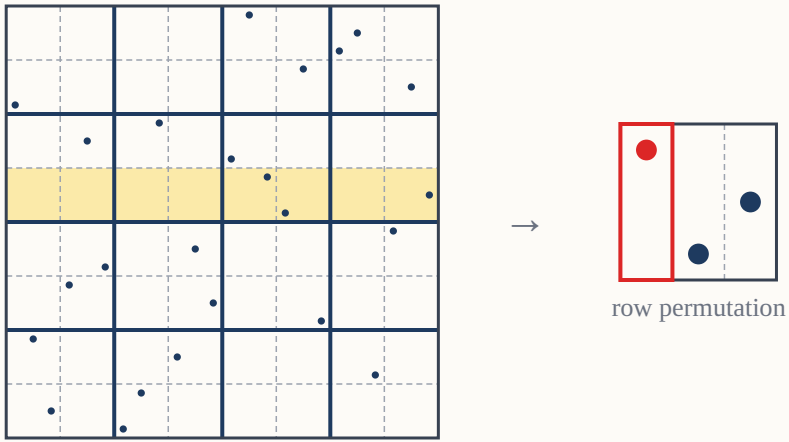
- Same approach as for columns – store precomputed queries for each shape in $o(n)$ space.
- Individual fine rows store only index into this precomputed table.
- Inverse to the column structure – for given cell and within cell rank, return the rank within fine row.



Row permutation

Forgetting the empty columns, the fine row again collapses to a short permutation with $\mathcal{O}(c_\pi)$ cells:

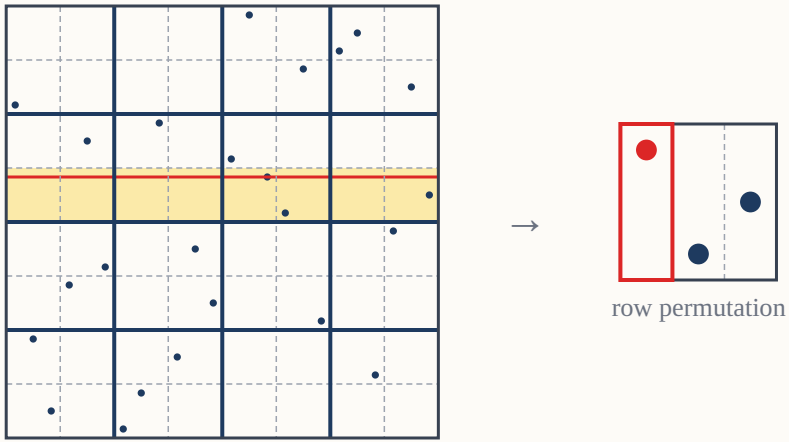
- Same approach as for columns – store precomputed queries for each shape in $o(n)$ space.
- Individual fine rows store only index into this precomputed table.
- Inverse to the column structure – for given cell and within cell rank, return the rank within fine row.



Row permutation

Forgetting the empty columns, the fine row again collapses to a short permutation with $\mathcal{O}(c_\pi)$ cells:

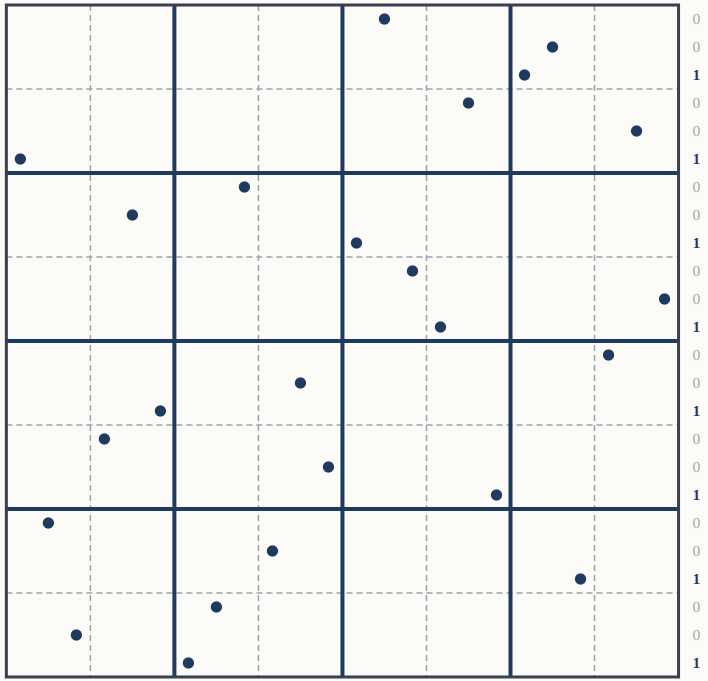
- Same approach as for columns – store precomputed queries for each shape in $o(n)$ space.
- Individual fine rows store only index into this precomputed table.
- Inverse to the column structure – for given cell and within cell rank, return the rank within fine row.



Locating fine row

Finally, we complete the global rank j:

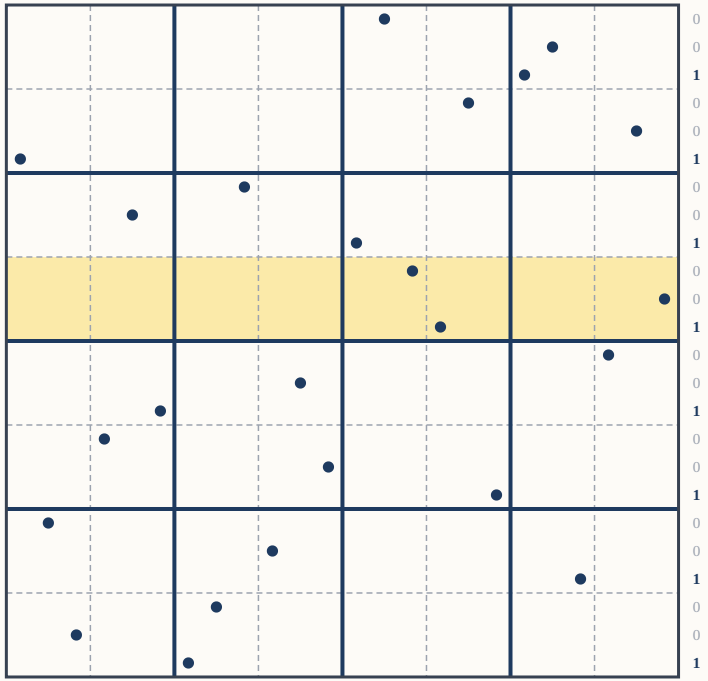
- Store a bitvector marking the bottom of every fine row.
- We compute the global offset of the fine row with queried point and add the rank within fine row obtained in previous step.
- As with columns, Rank and Select data structure with $\mathcal{O}(1)$ time and $o(n)$ space.



Locating fine row

Finally, we complete the global rank j:

- Store a bitvector marking the bottom of every fine row.
- We compute the global offset of the fine row with queried point and add the rank within fine row obtained in previous step.
- As with columns, Rank and Select data structure with $\mathcal{O}(1)$ time and $o(n)$ space.



Conclusion

Conclusion

- We have seen how a result from extremal combinatorics (Marcus-Tardos theorem) can be used to design algorithms and prove their optimality:
  - In sorting, Marcus-Tardos theorem plays the role of a bookkeeping device.
  - In compact representations, it allows explicit sparse structural decomposition.
- In both cases, Marcus-Tardos provides upper bound on runtime (through Füredi-Hajnal) and information-theoretic lower bound (through Stanley-Wilf).

Future directions:

- What about other problems whose input is a sequence of numbers or points in the plane?
- What can we say (asymptotically) about the constants c_π and s_π?
- Does this approach generalize to (sparse) pattern-avoiding matrices?

Conclusion

- We have seen how a result from extremal combinatorics (Marcus-Tardos theorem) can be used to design algorithms and prove their optimality:
  - In sorting, Marcus-Tardos theorem plays the role of a bookkeeping device.
  - In compact representations, it allows explicit sparse structural decomposition.
- In both cases, Marcus-Tardos provides upper bound on runtime (through Füredi-Hajnal) and information-theoretic lower bound (through Stanley-Wilf).

Future directions:

- What about other problems whose input is a sequence of numbers or points in the plane?
- What can we say (asymptotically) about the constants c_π and s_π?
- Does this approach generalize to (sparse) pattern-avoiding matrices?

Thank you!

References

Cibulka, J. (2009). On constants in the Füredi–Hajnal and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, *116*(2), 290–302. https://doi.org/10.1016/j.jcta.2008.06.003

Füredi, Z., & Hajnal, P. (1992). Davenport-Schinzel theory of matrices. *Discret. Math.*, *103*(3), 233–251. https://doi.org/10.1016/0012-365X(92)90316-8

Klazar, M. (2000). The Füredi-Hajnal Conjecture Implies the Stanley-Wilf Conjecture. *Formal Power Series and Algebraic Combinatorics: 12th International Conference, FPSAC'00*, 250–255. https://doi.org/10.1007/978-3-662-04166-6_22

Kővári, T., Sós, V. T., & Turán, P. (1954). On a problem of K. Zarankiewicz. *Colloquium Mathematicum*, *3*, 50–57.

Marcus, A., & Tardos, G. (2004). Excluded permutation matrices and the Stanley–Wilf conjecture. *Journal of Combinatorial Theory, Series A*, *107*(1), 153–160. https://doi.org/10.1016/J.JCTA.2004.04.002

Pettie, S., & Tardos, G. (2025). A Refutation of the Pach-Tardos Conjecture for 0-1 Matrices. *Combinatorica*, *45*(6), 59. https://doi.org/10.1007/s00493-025-00187-7

References (cont.)

Arratia, R. (1999). On the Stanley-Wilf Conjecture for the Number of Permutations Avoiding a Given Pattern. *Electron. J. Comb.*, *6*. https://doi.org/10.37236/1477

Bevan, D., Brignall, R., Price, A. E., & Pantone, J. (2020). A structural characterisation of Av(1324) and new bounds on its growth rate. *European Journal of Combinatorics*, *88*, 103115. https://doi.org/10.1016/j.ejc.2020.103115

Conway, A. R., & Guttmann, A. J. (2015). On the growth rate of 1324-avoiding permutations. *Advances in Applied Mathematics*, *64*, 50–69. https://doi.org/10.1016/j.aam.2014.11.002

Fox, J. (2013). *Stanley-Wilf limits are typically exponential*.

Knuth, D. E. (1968). *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley.

West, J. (1995). Generating trees and the Catalan and Schröder numbers. *Discret. Math.*, *146*(1–3), 247–262. https://doi.org/10.1016/0012-365X(94)00067-1

References (cont.)

Arthur, D. (2007). Fast sorting and pattern-avoiding permutations. *2007 Proceedings of the Fourth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, 169–174. https://doi.org/10.1137/1.9781611972979.1

Berendsohn, B. A., Kozma, L., & Opler, M. (2024). Optimization with Pattern-Avoiding Input. In B. Mohar, I. Shinkar, & R. O'Donnell (Eds.), *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024* (pp. 671–682). ACM. https://doi.org/10.1145/3618260.3649631

Chalermsook, P., Goswami, M., Kozma, L., Mehlhorn, K., & Saranurak, T. (2015). Pattern-Avoiding Access in Binary Search Trees. *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*, 410–423. https://doi.org/10.1109/FOCS.2015.32

Chalermsook, P., Gupta, M., Jiamjitrak, W., Acosta, N. O., Pareek, A., & Yingchareonthawornchai, S. (2023). Improved Pattern-Avoidance Bounds for Greedy BSTs via Matrix Decomposition. *ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*, 509–534. https://doi.org/10.1137/1.9781611977554.ch22

Fredman, M. L. (1976). How Good is the Information Theory Bound in Sorting? *Theor. Comput. Sci.*, *1*(4), 355–361. https://doi.org/10.1016/0304-3975(76)90078-5

Munro, J. I., & Wild, S. (2018). Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs. *26th Annual European Symposium on Algorithms, ESA 2018, August 20-22, 2018, Helsinki, Finland*, *112*, 63:1-63:16. https://doi.org/10.4230/LIPIcs.ESA.2018.63

References (cont.)

Ackerman, E., Barequet, G., & Pinter, R. Y. (2006). A bijection between permutations and floorplans, and its applications. *Discrete Applied Mathematics*, *154*(12), 1674–1684.

Chakraborty, S., Jo, S., Kim, G., & Sadakane, K. (2024). Succinct Data Structures for Baxter Permutation and Related Families. In J. Mestre & A. Wirth (Eds.), *35th International Symposium on Algorithms and Computation, ISAAC 2024, December 8-11, 2024, Sydney, Australia* (Vol. 322, p. 17:1-17:17). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ISAAC.2024.17

Golynski, A. (2009). Cell probe lower bounds for succinct data structures. In C. Mathieu (Ed.), *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009* (pp. 625–634). SIAM. https://doi.org/10.1137/1.9781611973068.69

Hagerup, T. (1998). Sorting and searching on the word RAM. *STACS 98, 15th Annual Symposium on Theoretical Aspects of Computer Science, Paris, France, February 25-27, 1998*, *1373*, 366–398. https://doi.org/10.1007/BFb0028575

Munro, J. I., Raman, R., Raman, V., & Rao, S. S. (2012). Succinct representations of permutations and functions. *Theor. Comput. Sci.*, *438*, 74–88. https://doi.org/10.1016/J.TCS.2012.03.005

Opler, M. (2024). An Optimal Algorithm for Sorting Pattern-Avoiding Sequences. *65th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2024, Chicago, IL, USA, October 27-30, 2024*, 689–699. https://doi.org/10.1109/FOCS61266.2024.00049

References (cont.)

Felsner, S., Fusy, É., Noy, M., & Orden, D. (2011). Bijections for Baxter families and related objects. *Journal of Combinatorial Theory, Series A*, *118*(3), 993–1020.

Guillemot, S., & Marx, D. (2014). Finding small patterns in permutations in linear time. *ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, 82–101. https://doi.org/10.1137/1.9781611973402.7

Kozma, L., & Opler, M. (2025). *Compact representations of pattern-avoiding permutations*.

Step I without prior knowledge of π

Input: π-avoiding sequence S partitioned into m presorted sequences $S_1, \ldots, S_m$

1. $d \leftarrow 1$
2. while there are some elements remaining:
3. $\quad S_1, \ldots, S_{\lfloor m/2 \rfloor} \leftarrow$ merge consecutive pairs of sequences $\qquad \mathcal{O}(n)$
4. $\quad d \leftarrow 2 \cdot d, \quad m \leftarrow \lfloor m/2 \rfloor$
5. $\quad$ repeat m times or until all sequences are exhausted:
6. $\quad\quad S_{i_1}, \ldots, S_{i_{d+1}} \leftarrow d+1$ sequences with smallest initial elements
7. $\quad\quad x \leftarrow$ initial element of $S_{i_{d+1}}$ (largest out of these)
8. $\quad\quad$ output merge $S_{i_1}, \ldots, S_{i_d}$ while smaller than x $\qquad \mathcal{O}(\log d_{\max} \cdot n)$ overall

Claim: The algorithm terminates after at most $\lceil \log c_\pi \rceil + 1$ phases.

Proof: Let d_i and m_i be the values in the i-th phase. Observe $d_i = 2^i$ and $m_i \leq \frac{n}{d_i \log n}$. For $i = \lceil \log c_\pi \rceil + 1$ we have $d_i \geq 2c_\pi$, and the same argument shows m_i rounds suffice. □