Lecture 10

Computing medians and quicksort

Chinmay Nirkhe | CSE 421 Spring 2025

Previously in CSE 421…

Multiplication

- **Matrix multiplication**
  - $O(n^{2.87})$ time algorithm for $n \times n$ matrices
  - Strassen's divide and conquer algorithm
- **Integer multiplication**
  - $O(n^{1.58})$ time algorithm for multiplying n-bit numbers
  - Karatsuba's divide and conquer algorithm
- **Polynomial multiplication**
  - $O(n \log n)$ time algorithm for multiplying degree n polynomials
  - Convert to evaluation basis via Fast Fourier transform for quick evaluation

Today

Median

- **Input**: Input list $X = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ for n odd.
- **Output**: The median element i.e. $y_{(n+1)/2}$ when $Y = \mathrm{sort}(X)$.

- An upper bound for the runtime is $O(n \log n)$ from sorting + selecting.
- Can we do better? Could we achieve $O(n)$?

Median

- Consider a divide and conquer algorithm for median
- What would the recurrence relation have to be for $T(n) = O(n)$?
- **Case 1**: $T(n) = 2T(n/2) + O(1)$
  - Challenge is to split the problem X into two halves with $O(1)$ compute
  - And to “stitch” the solutions to the two subproblems together in $O(1)$ compute
- **Case 2**: $T(n) = T(n/2) + O(n)$
  - With $O(n)$ time, we can make a constant number of passes through the list X
  - After constant number of passes, we need to find a sublist X' of size $n/2$ which must contain the median
  - Then we recurse on the sublist X'

Selection

- Let's define a more general problem called "Selection"
    - **Input:** list $(X, k) \in \mathbb{R}^n \times [n]$.
    - **Output:** The k-th element y_k when $\vec{y} = \mathrm{sort}(\vec{x})$.
- Generalizes the median problem

Selection

Find the 6th element

0	1	7	2	1	3	4	4	8	2	1	0	3	3	6	5	5	2	2	3

Selection

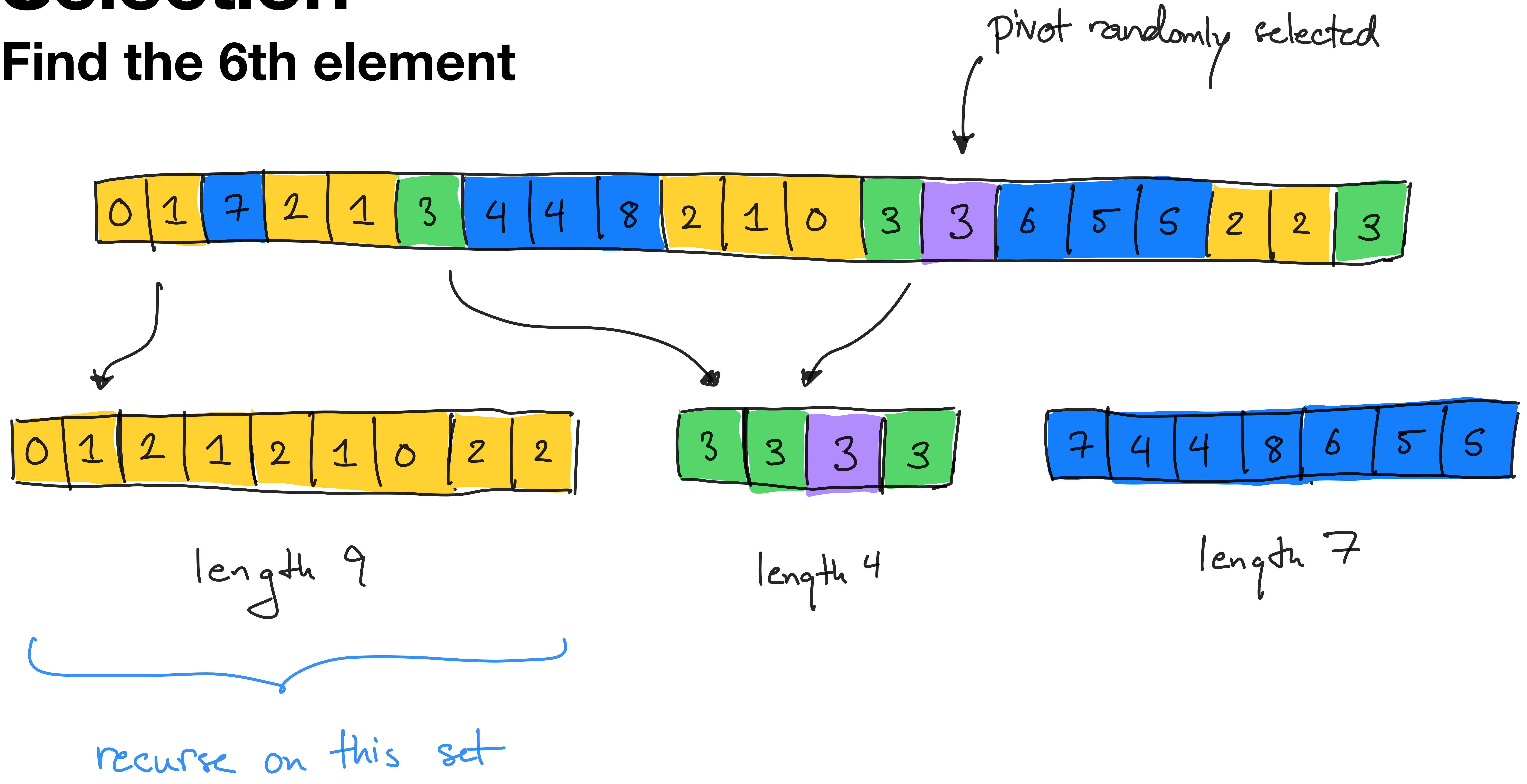
Find the 6th element

pivot randomly selected

0	1	7	2	1	3	4	4	8	2	1	0	3	3	6	5	5	2	2	3

Selection

Find the 6th element



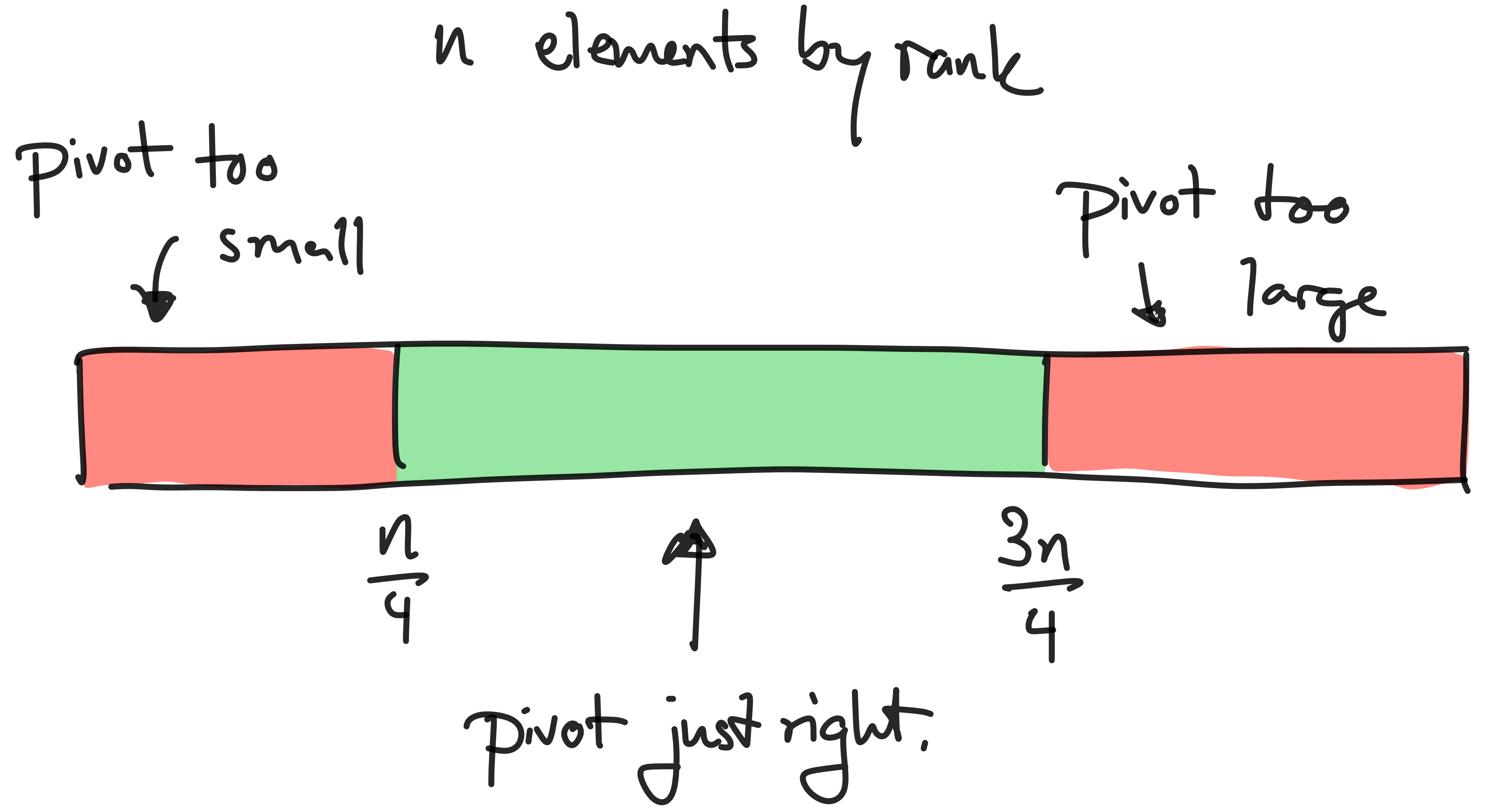
Selection

- **Recursive algorithm** $\mathrm{Selection}(X, k)$:
  - Randomly sample j from $[n]$. Call x_j the "**pivot**".
  - Filter X into X_L, X_E, and X_R based on if $x_i < x_j$, $x_i = x_j$, or $x_i > x_j$.
  - If $|X_L| \geq k$, recursively output $\mathrm{Selection}(X_L, k)$.
  - Else if, $|X_L| + |X_E| \geq k$, output x_j.
  - Else, recursively output $\mathrm{Selection}(X_R, k - |X_L| - |X_E|)$.

Runtime analysis

- In order to apply the master theorem, we would need to argue that each recursive call was reducing the input size from n to n/b for $b > 1$
  - $T(n) = T(n/b) + cn \implies T(n) = \dfrac{c}{1 - 1/b}n$
- However, each call may not reduce the size from n to n/b
- Depends on how close the randomly chosen x_j is to the middle
  - If pivot x_j was the largest element, then $|X_L| = n - 1, |X_E| = 1$, and $|X_R| = 0$.
  - Decreases instance size from n to $n - 1$.
  - Fortunately, the probability this occurs is $1/n$.

Runtime analysis



- **Amortized analysis**:
  - If pivot x_j is the ℓ-th element, then the next problem is of size $\leq \max\{\ell, n - \ell\}$.
  - With probability $\geq 1/2$, pivot x_j is the ℓ-th element for $\ell \in \{n/4, \ldots, 3n/4\}$.
  - The expected compute in reducing from n-sized instance to a $3n/4$-sized instance is $O(n)$.
- Total **expected** runtime: $T(n) = T(3n/4) + O(n) \implies T(n) = O(n)$.

Runtime analysis

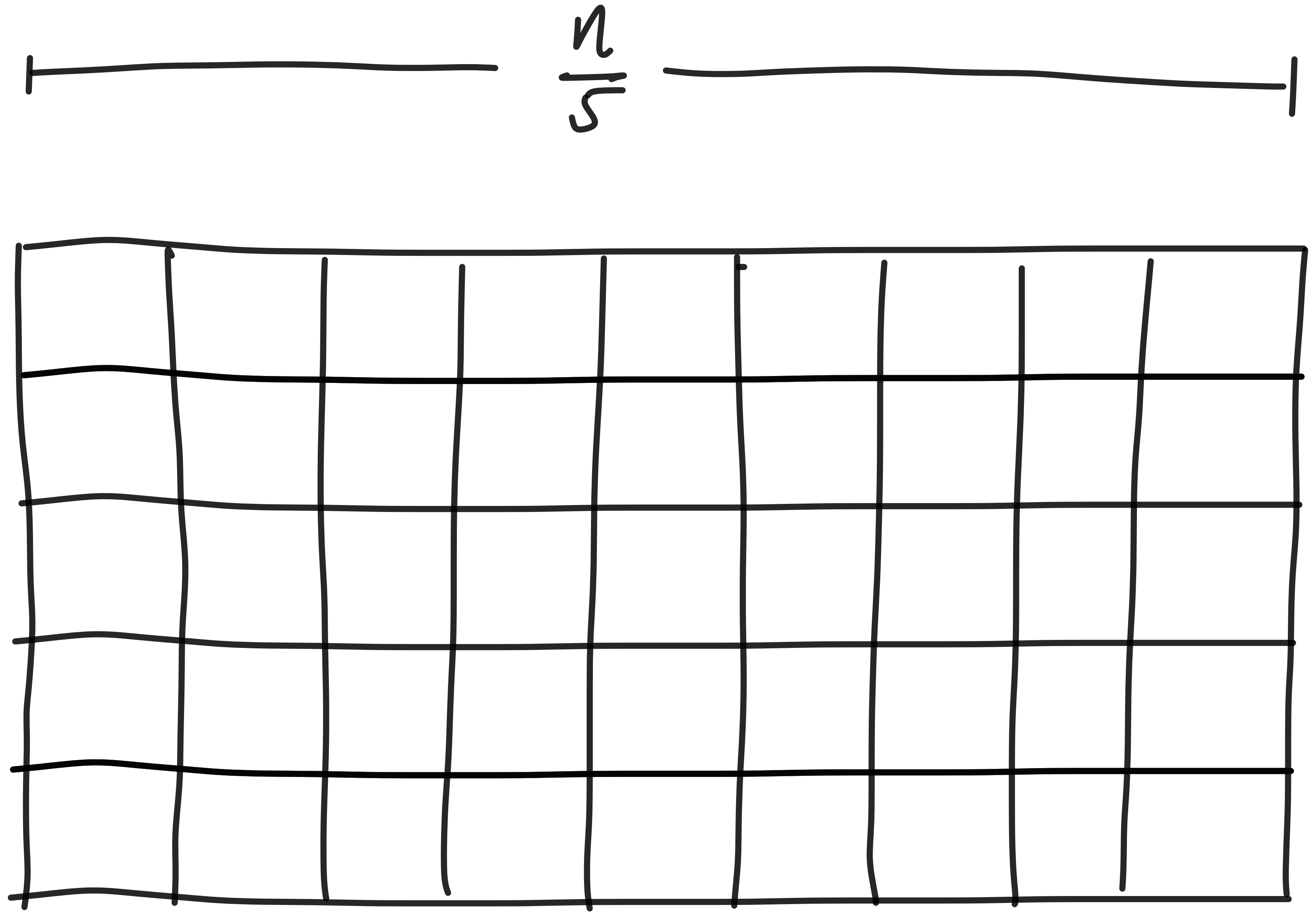
- **Amortized analysis:**
  - If pivot x_j is the ℓ-th element, then the next problem is of size $\leq \max\{\ell, n-\ell\}$.
  - With probability $\geq 1/2$, pivot x_j is the ℓ-th element for $\ell \in \{n/4,\ldots,3n/4\}$.
  - The expected compute in reducing from n-sized instance to a $3n/4$-sized instance is $O(n)$.
    - $\geq 1/2$ probability, shrinks in 1 reduction.
    - $\geq 1/4$ probability, shrinks in 2 reductions.
    - … $\geq 1/2^j$ probability, shrinks in j reductions …
    - Expected compute is $\leq O(n) \cdot (\frac{1}{2} + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \ldots) = O(n) \cdot 2$
- Total **expected** runtime: $T(n) = T(3n/4) + O(n) \implies T(n) = O(n)$.

Derandomization

- The worst case runtime is $O(n^2)$.
    - Only happens with $2^{-\Omega(n \log n)}$ probability.
- But, is there an algorithm that didn't require randomness?
- If we could guarantee that the pivot x_j was in the middle half, then each recursion would decrease in size by $3/4$.
- **Blum-Pratt-Floyd-Rivest-Tarjan (1973)**: Calculate a pivot in the middle $4n/10$ in time $O(n)$.

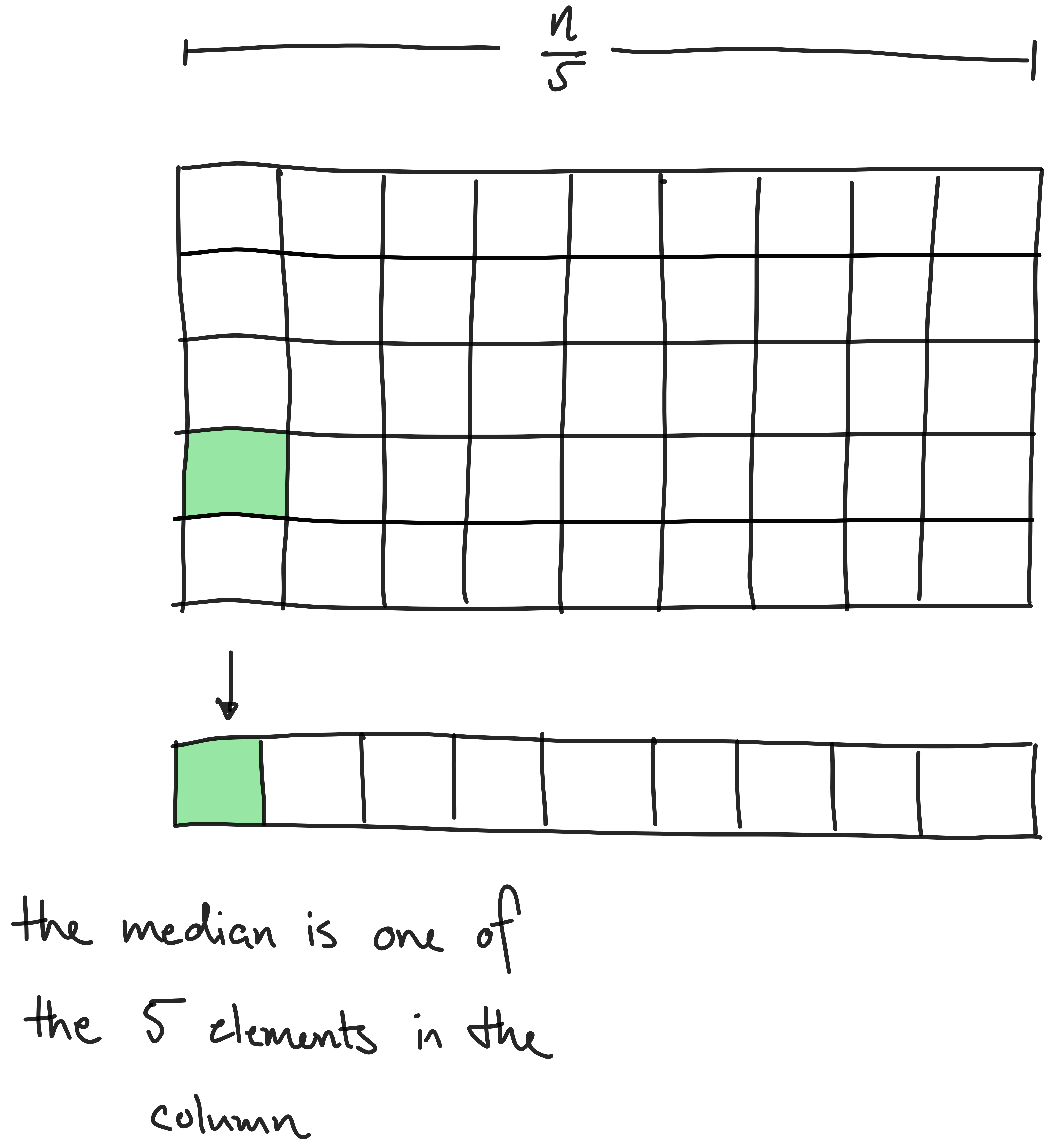
Pivot selection algorithm

- Express the n elements as a $5 \times (n/5)$ matrix of elements



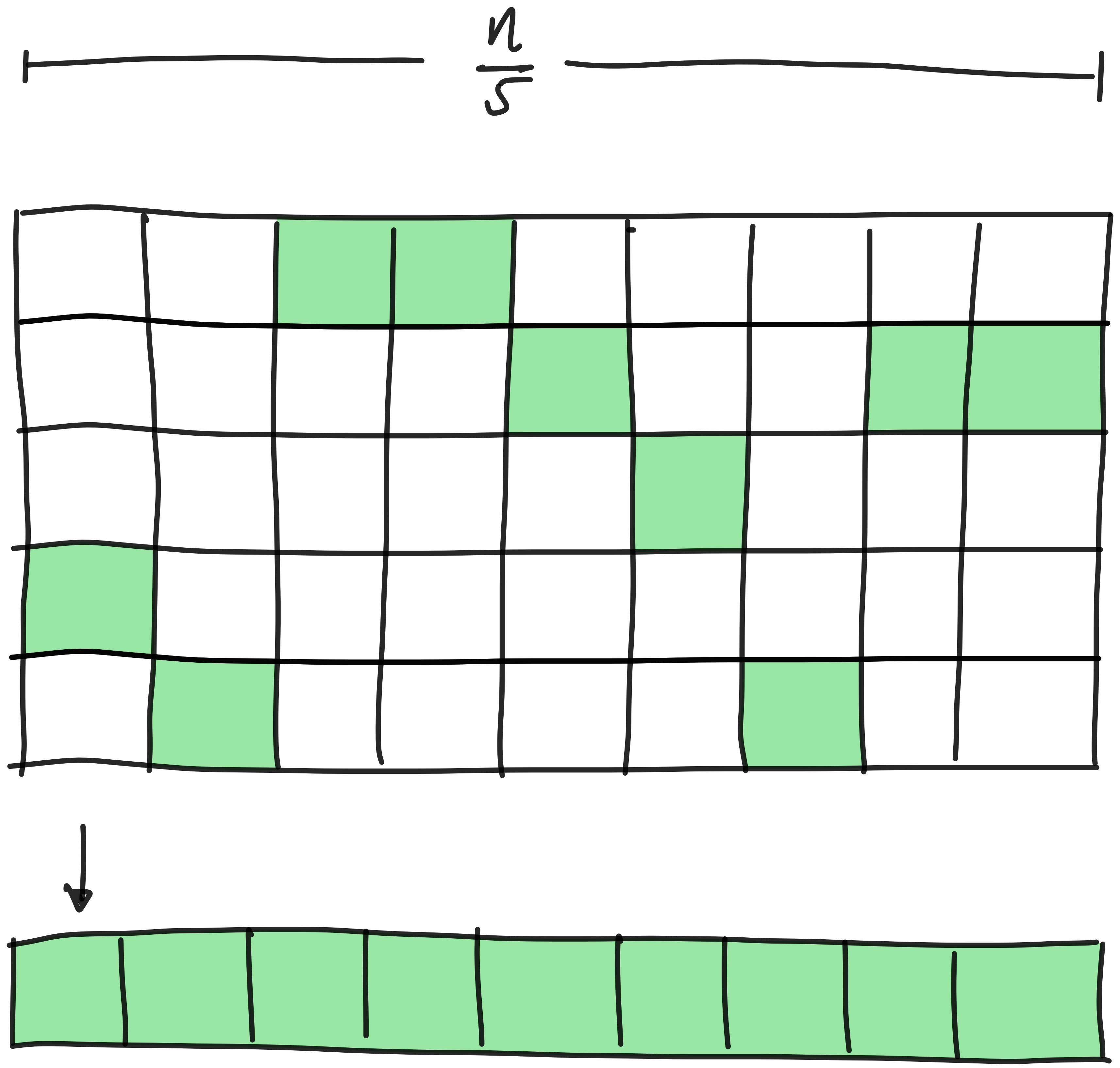
Pivot selection algorithm

- Express the n elements as a $5 \times (n/5)$ matrix of elements
- Calculate the medians of each of the columns: $Y = (y_1, y_2, \ldots, y_{n/5})$



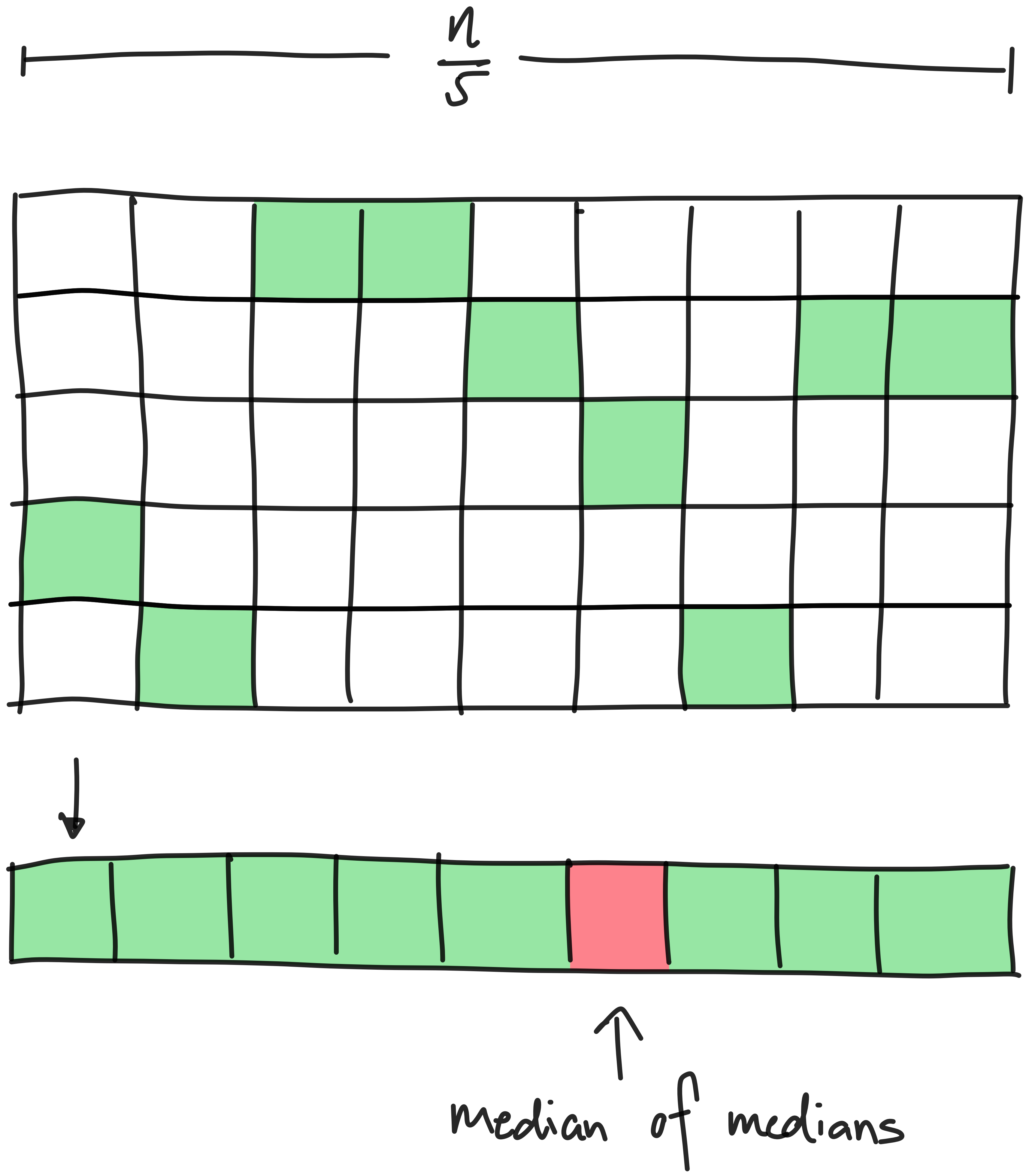
Pivot selection algorithm

- Express the n elements as a $5 \times (n/5)$ matrix of elements
- Calculate the medians of each of the columns: $Y = (y_1, y_2, \ldots, y_{n/5})$



Pivot selection algorithm

- Express the n elements as a $5 \times (n/5)$ matrix of elements
- Calculate the medians of each of the columns: $Y = (y_1, y_2, \ldots, y_{n/5})$
- Choose the pivot as the median of the medians: $p \leftarrow \text{median}(Y)$



Pivot selection algorithm

Runtime analysis

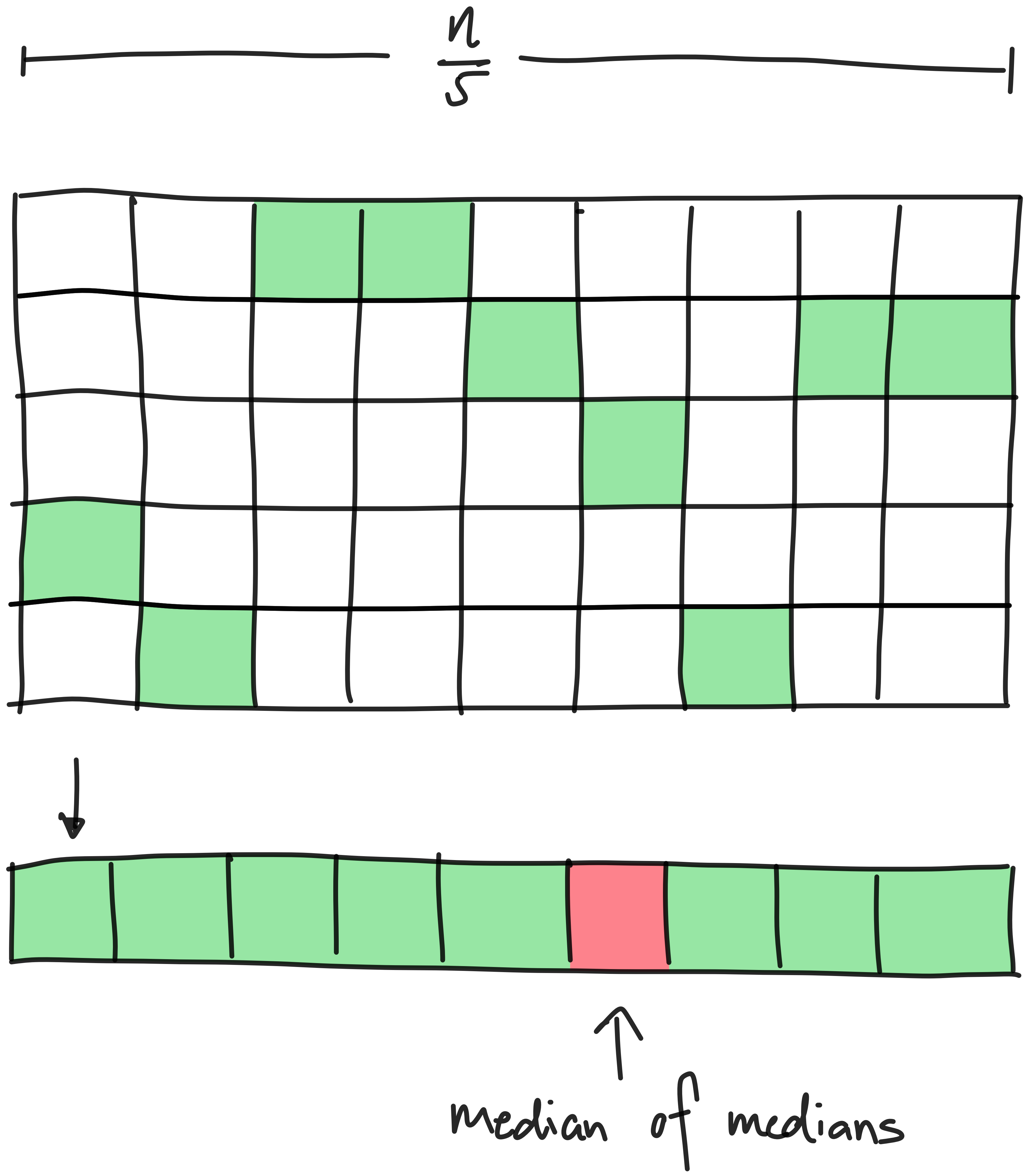
- Express the n elements as a $5 \times (n/5)$ matrix of elements

Only semantic.

- Calculate the medians of each of the columns: $Y = (y_1, y_2, \ldots, y_{n/5})$

O(1) per col. Total O(n).

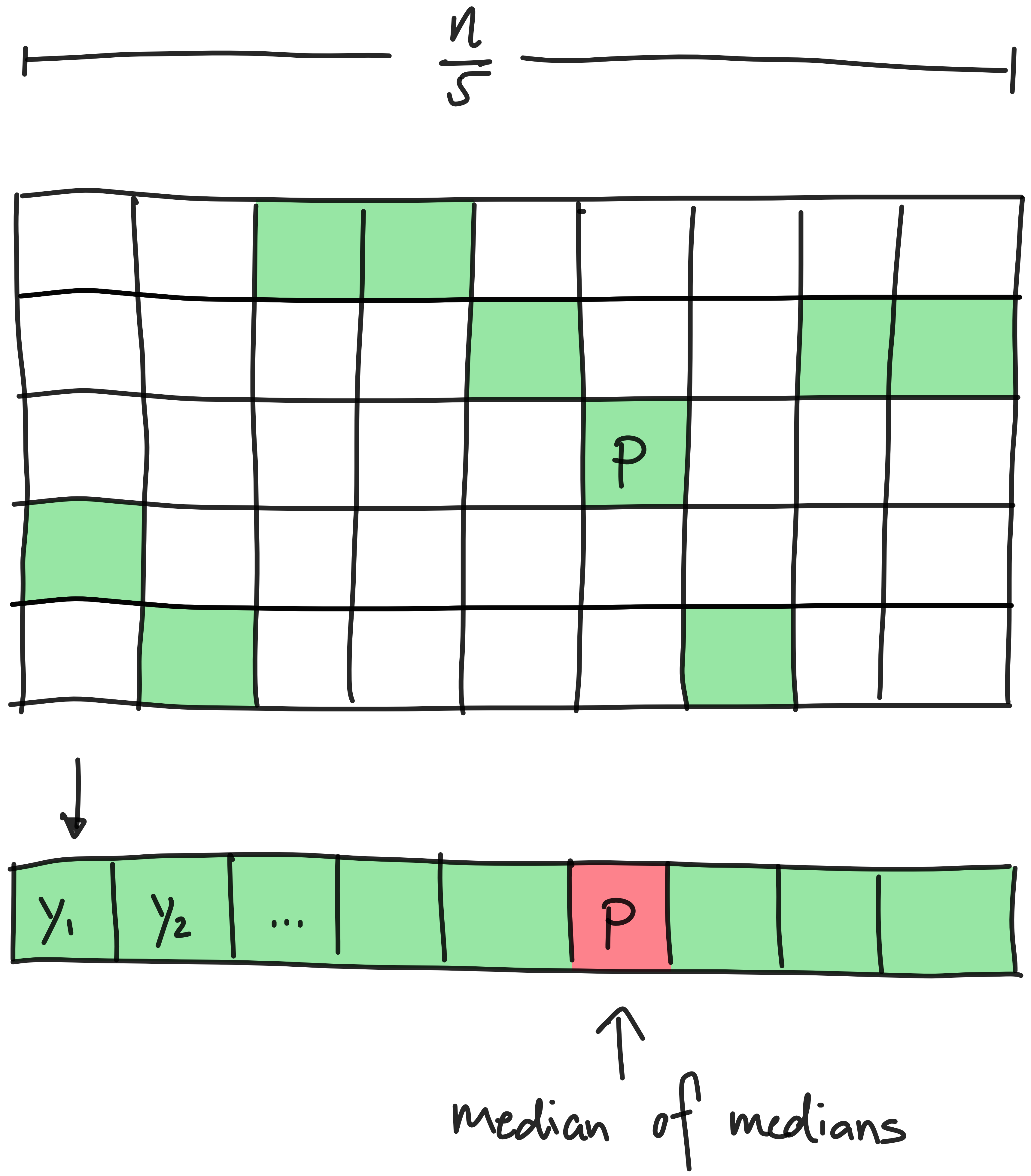
- Choose the pivot as the median of the medians: $p \leftarrow \text{median}(Y)$ T(n/5) recursively



Total time: $T(n) = T(\frac{n}{5}) + O(n) \Rightarrow T(n) = O(n)$.

Pivot selection algorithm

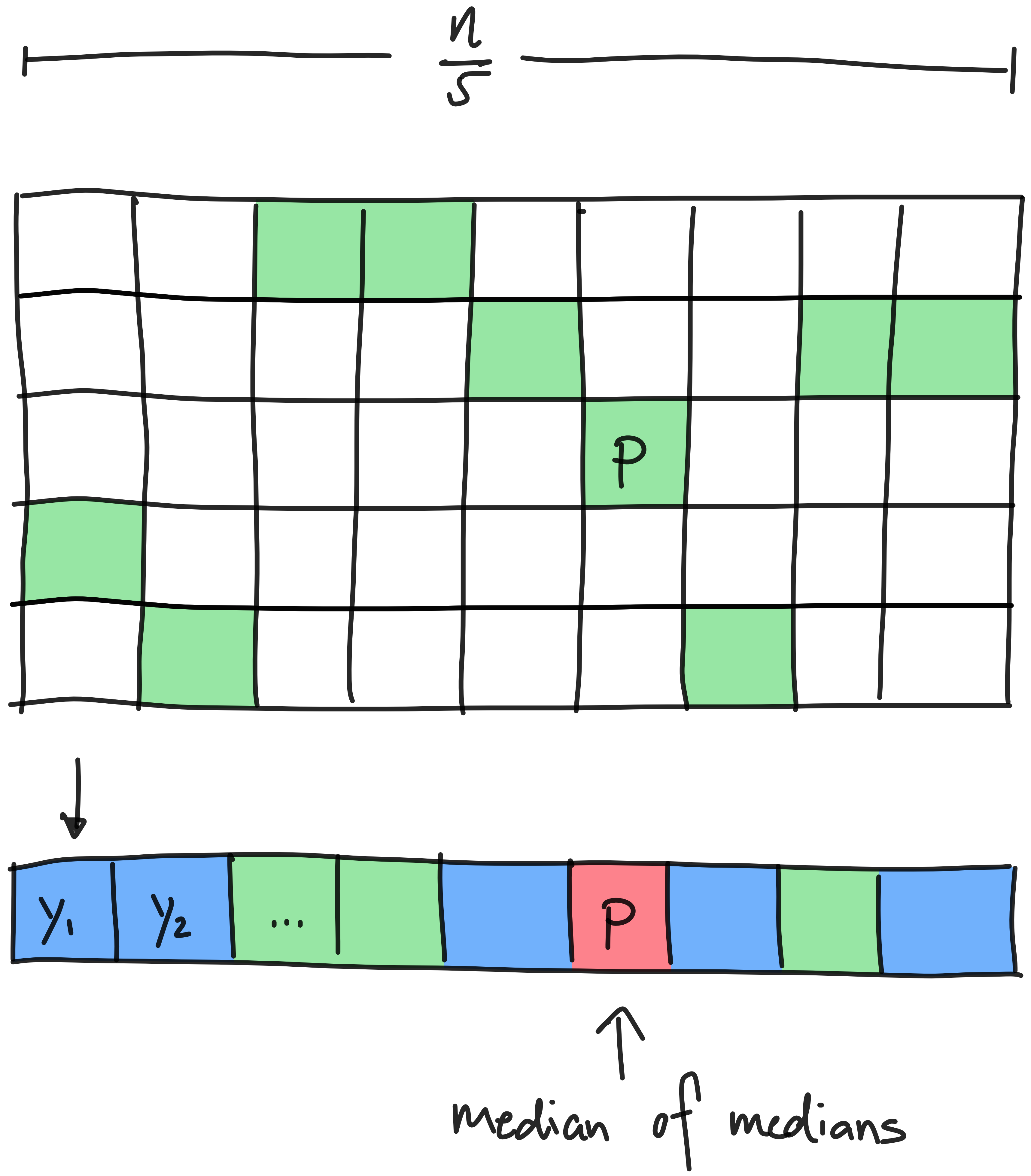
Proof of correctness



Pivot selection algorithm

Proof of correctness

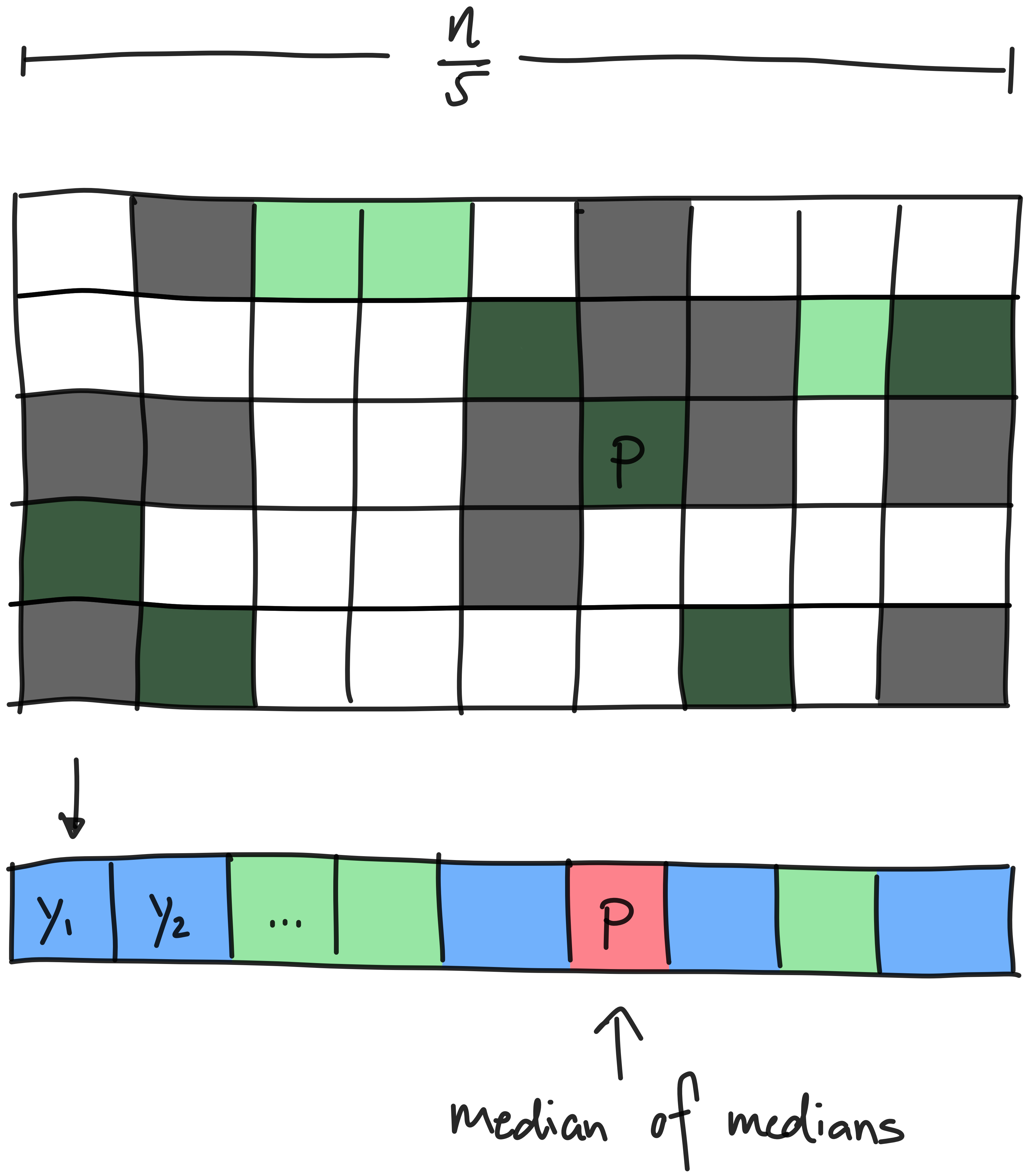
- There are $\geq n/10$ columns such that $y_j \geq p$.



Pivot selection algorithm

Proof of correctness

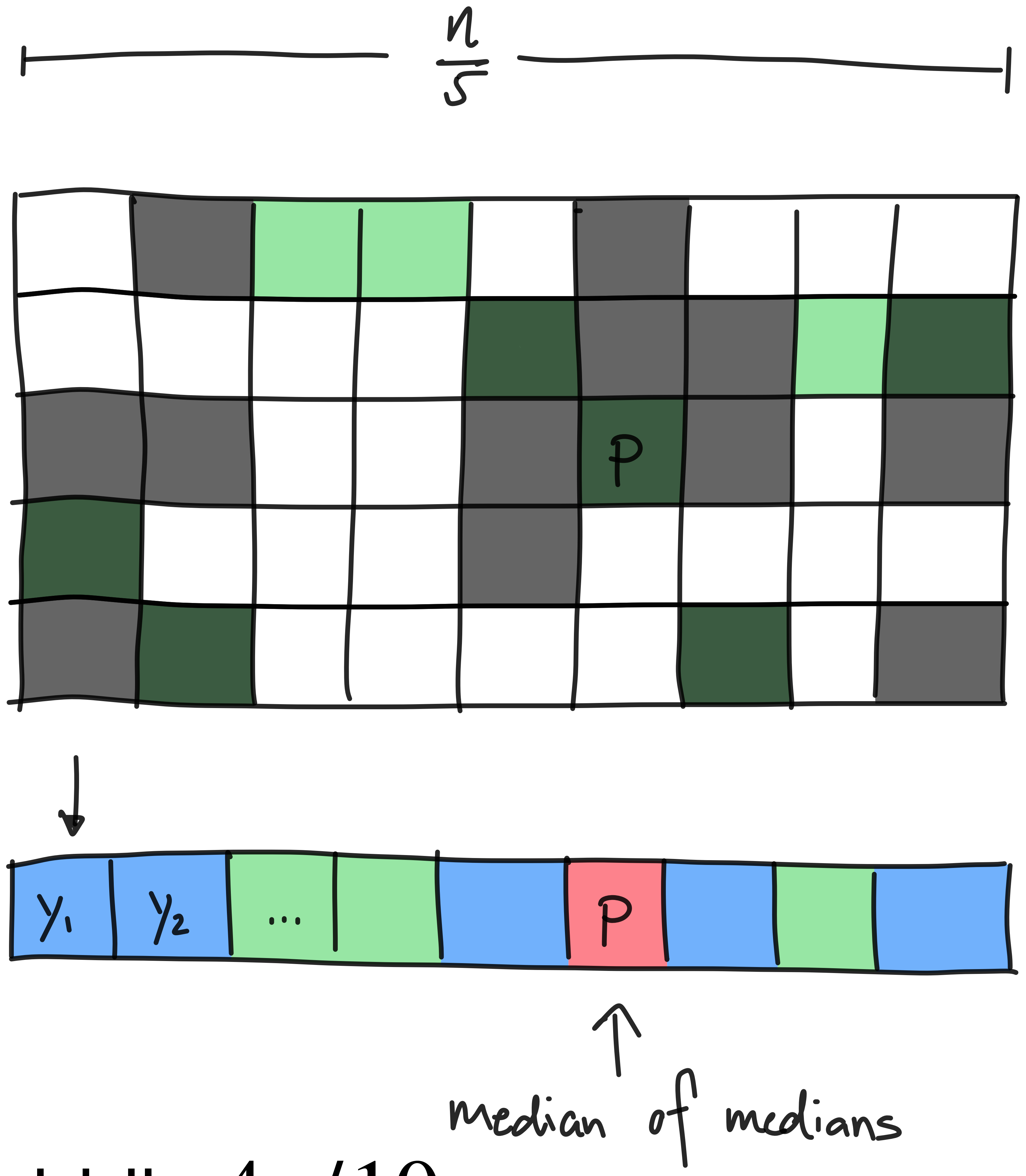
- There are $\geq n/10$ columns such that $y_j \geq p$.
- In each such column, there are 3 elements $\geq y_j$.



Pivot selection algorithm

Proof of correctness

- There are $\geq n/10$ columns such that $y_j \geq p$.
- In each such column, there are 3 elements $\geq y_j$.
- Therefore, there are $\geq 3n/10$ elements $\geq p$.
- Similarly, there are $\geq 3n/10$ elements $\leq p$.



- So, p is in the middle $4n/10$ elements and a good pivot.

Median/Selection algorithm

Total: $T(n) = T\left(\frac{7}{10}n\right) + T\left(\frac{n}{5}\right) + O(n)$

$\Rightarrow T(n) = O(n)$

Pset problem on how to analyze this generalization of Master theorem

- **Input:** $(X, k) \in \mathbb{R}^n \times [n]$
- **Output:** the k-th item in the list X
- **Algorithm:**
  - Calculate $p \leftarrow$ median-of-medians(X) in a $5 \times (n/5)$ division.
  - Filter X into X_L, X_E, and X_R based on p
  - If $|X_L| \geq k$, recurse $\text{Selection}(X_L, k)$
    - Else if $|X_L| + |X_E| \geq k$, return p
    - Else, return $\text{Selection}(X_R, k - |X_L| - |X_E|)$.

recursive $T\left(\frac{n}{5}\right) + O(n)$

recursive $T\left(\frac{7}{10}n\right)$

Quicksort algorithm

- The algorithm we just analyzed, "Quickselect", can be generalized to sorting
- **Sorting algorithm** $\mathrm{Quicksort}(X)$:
  - Pick a pivot p (either randomized or with median-of-medians)
  - Filter X into X_L, X_E, X_R by comparing elements with p
  - Concatenate $\mathrm{Sort}(X_L), X_E, \mathrm{Sort}(X_R)$.

Computing expected runtime is challenging due to variable size

Quicksort algorithm

Runtime analysis

- Runtime depends on pivot selection
- **Median-of-means**:
  - $T(n) \leq T(\alpha n) + T(n - \alpha n) + O(n)$ for $\alpha \in [0.3, 0.7]$
  - $T(n) = O(n)$ by analysis you will solve on your pset
- **Choose random element**:
  - Worst case: $O(n^2)$ time
  - Amortized: $O(n \log n)$ (next!)

Quicksort algorithm

Runtime analysis

[for random choice of pivot]

- **Observations**:
  - The runtime of Quicksort is proportional to the number of comparisons
  - The algorithm only compares two elements if one is the pivot
- Let $Y = (y_1, \ldots, y_n)$ be the sorted version of the input.
- Let $p_{ij} = \mathbf{Pr}\left[y_i \text{ and } y_j \text{ are compared}\right]$
- **Claim**: $p_{ij} \leq \dfrac{2}{j-i+1}$ when $i < j$.

Expected number of comparisons:

$$\sum_{i<j} P_{ij} \leq 2 \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$= 2 \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{1}{k+1}$$

$$= 2 \sum_{i=1}^{n} \underbrace{\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-i+1}\right)}_{\leq \log(n-i+1)+1 \leq \log(n)+1}$$

$$\leq 2n \log n + 2n.$$

Runtime of quicksort $= O(n \log n)$.

Proof of claim

- **Claim:** $p_{ij} \leq \dfrac{2}{j-i+1}$ when $i < j$.
- **Proof:**
    - $y_i \leq y_j$ and y_i and y_j are compared at most once
        - Comparisons only occur when one of them is the pivot
        - Case 1: $y_i, y_j \in X_E$ and we never recurse on X_E
        - Case 2: $y_i \in X_E, y_j \in X_R$ and we never compare between $X_L, X_E,$ and X_R
        - Case 3: $y_i \in X_L, y_j \in X_E$ and we never compare between $X_L, X_E,$ and X_R

- If and when y_i and y_j are compared during $\mathrm{sort}(X')$ then $y_i, y_{i+1}, y_{i+2}, \ldots, y_j \in X'$
    - Can be formally proven via induction
    - So $|X'| \geq j - i + 1$.
    - Probability that either y_i or y_j is chosen as pivot is $\leq \dfrac{2}{j-i+1}$.

Sorting in the real world

- **Quicksort**
  - Fast almost always, especially for in-memory sorting.
  - Works well with caches due to good locality of reference.
  - In practice,
    - Don't filter X_L, X_E, and X_R. Use in-place swaps.
    - When n is small, insertion sorting is a better base case.
    - Pick pivot randomly for small n, median of 3 random values for medium n, and median-of-medians on 9 elements for large n
    - Never actually run the median-of-medians pivot finding routine

Sorting in the real world

- **Mergesort**
  - Used when data is expressed as a linked list and RAM access to entries in the middle of the list is non-existent
  - Sorting over a dataset that cannot be stored in memory
  - Uses $O(n)$ extra space when sorting arrays over Quicksort

Sorting in the real world

- **Insertion sort**
  - Best when data is almost sorted already
  - $O(n^2)$ when far from sorted
- **Heap sort** - memory efficient choice
- **Bucket sort** - distribution aware sorting
- Etc…