CS 331, Fall 2026

Lecture 2 (8/26)

Today: - Recursion trees
- MergeSort
- Inversions
- Selection

Today: runtimes + correctness.

Last time: Multiply(a, b)

a: n digits, b: n digits

NaiveMultiply(a, b) # $ab = a_1b_1 10^n + (a_1b_0 + a_0b_1)10^{n/2} + a_0b_0$

$(a_1, a_0) \leftarrow (\text{int}(a/10^{n/2}), a \% 10^{n/2})$

$(b_1, b_0) \leftarrow (\text{int}(b/10^{n/2}), b \% 10^{n/2})$

} first, second halves $\frac{n}{2}$ digits

$S_{big} \leftarrow$ NaiveMultiply(a_1, b_1)

$S_{mid} \leftarrow$ NaiveMultiply(a_1, b_0) + NaiveMultiply(a_0, b_1)

$S_{sm} \leftarrow$ NaiveMultiply(a_0, b_0)

Return: $S_{big} \cdot 10^n + S_{mid} \cdot 10^{n/2} + S_{sm}$

Runtime: $\boxed{T(n) = 4T(\frac{n}{2}) + O(n)}$ How to analyze?

Karatsuba (a, b) # $(a_1 + a_0)(b_1 + b_0)$
$- a_1 b_1 - a_0 b_0 = a_1 b_0 + a_0 b_1$

$S_{big} \leftarrow$ Karatsuba(a_1, b_1)
$S_{sm} \leftarrow$ Karatsuba(a_0, b_0)
$S_{mid} \leftarrow$ Karatsuba$(a_1 + a_0, b_1 + b_0) - S_{big} - S_{sm}$

Return: $S_{big} \cdot 10^n + S_{mid} \cdot 10^{n/2} + S_{sm}$

Runtime: $\boxed{T(n) = 3T\left(\frac{n}{2}\right) + O(n)}$

Clearly better... how much?

Recursion trees (Part II, Section 3)

Last time we proved:

$a_0 + a_0 r + a_0 r^2 + \dots + a_0 r^k$ (geometric series)

$= a_0(1 + r + r^2 + \dots + r^k) = a_0 \cdot \frac{r^{k+1} - 1}{r - 1}$

if $r < 1$, it's $\Theta(1)$
$r > 1$ it's $\Theta(r^k)$
$r = 1$ it's $\Theta(k)$

r is constant

e.g. $3/4$, $1/6$, 12

So,

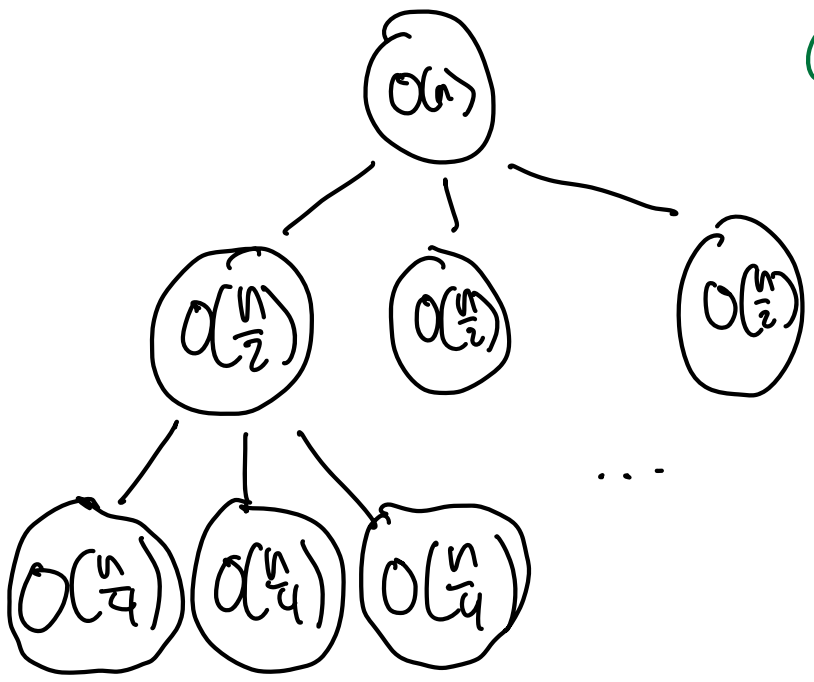
$$a_0 \sum_{i=0}^{k} r^i = \begin{cases} \Theta(a_0) & \text{"first term dominates"} \quad \text{if } r < 1 \\ \Theta(a_0 k) & \text{"balanced"} \quad r = 1 \\ \Theta(a_0 r^k) & \text{"last term dominates"} \quad r > 1 \end{cases}$$

Idea: Write runtime of recursive algo as geometric sequence

Standard recurrence: $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

- a: # of subproblems
- $\frac{n}{b}$: subproblem size
- $f(n)$: base cost

Example $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$ (multiplication)



Level 0 — Cost: $O(n)$

Level 1 — $O\left(\frac{3n}{2}\right)$

Level 2 — $O\left(\frac{9n}{4}\right)$

Level k

$k = \lceil \log_2(n) \rceil$

Note: there are technically "rounding errors",
we should write $T(n) = a\,T(\lfloor \frac{n}{b} \rfloor) + (2-a)\,T(\lceil \frac{n}{b} \rceil) \ldots$
e.g. can't split odd-sized lists exactly in half.

OK to ignore in this course, See lecture notes.

General cost of recursion tree

Example $f(n) = n^c$

level 0	$O(f(n)) \times 1$	$O(n^c)$
level 1	$O(f(\frac{n}{b})) \times a$	$O(n^c) \cdot \frac{a}{b^c}$
level 2	$O(f(\frac{n}{b^2})) \times a^2$	$O(n^c) \cdot \frac{a^2}{b^{2c}}$
⋮		⋮
level k	$O(f(\frac{n}{b^k})) \times a^k$	$O(n^c) \cdot \frac{a^{\log_b(n)}}{n^c}$
	$=$	$=$
$k \cong \log_b(n)$	$O(1) \times n^{\log_b(a)}$	$O(n^{\log_b(a)})$

Why? See lecture notes or try it yourself.

In a geometric sequence, first or last term wins.

In this class, $f(n)$ almost always of the form

$$f(n) = \Theta(n^c \log^d(n)) \text{ for } \begin{matrix} c \geq 1 \\ d \geq 0 \end{matrix}.$$

In that case, we can use the...

Master Theorem

Suppose recurrence looks like

$$T(n) = a\,T\left(\frac{n}{b}\right) + \overbrace{\Theta(n^c \log^d(n))}^{f(n)}$$

(common ratio: $\frac{a}{b^c}$ ← is it $>, <, = 1$?)

"root-heavy"

$c > \log_b(a)$: $T(n) = \Theta(f(n))$

"leaves-heavy"

$c < \log_b(a)$: $T(n) = \Theta(n^{\log_b(a)})$

"balanced"

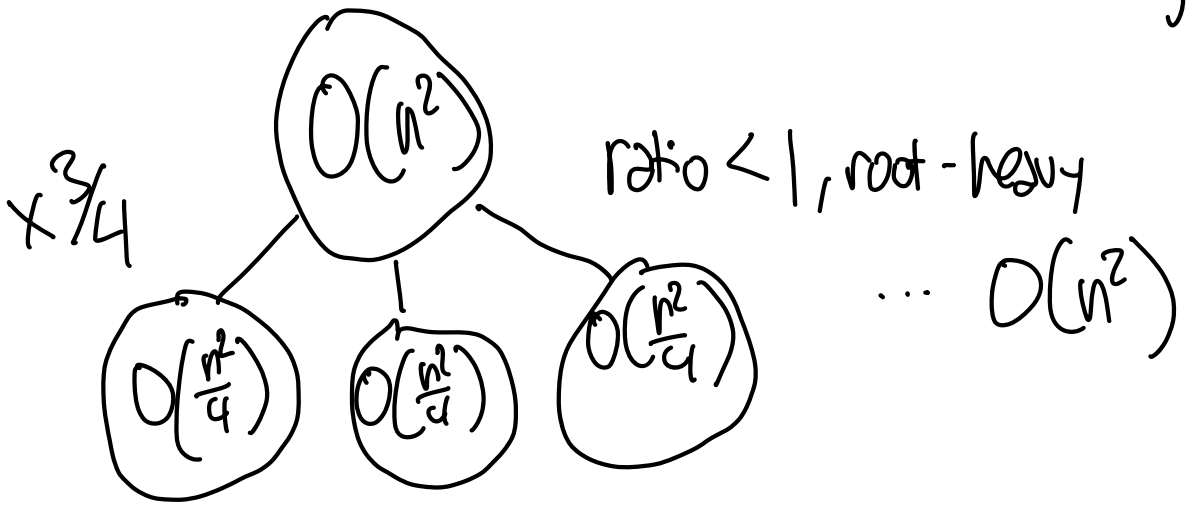
$c = \log_b(a)$: $T(n) = \Theta(f(n)\log(n))$

Quick tip: just draw one layer.

You can see geometric sequence already.

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n^2)$$

... in general, even if Master Thm doesn't apply, can still use recursion tree to induce a geometric sequence



ratio < 1, root-heavy

$\cdots$ $O(n^2)$

$\cdots$ the trend continues

Exercise

$$T(n) = 6T\left(\frac{n}{3}\right) + O(n^2)$$

$$T(n) = 8T\left(\frac{n}{4}\right) + O(n^{1.5})$$ Simplify.

$$T(n) = T\left(\frac{4n}{5}\right) + T\left(\frac{n}{3}\right) + O(n)$$

Merge Sort (Part II, Section 5.1)

The most famous recurrence in algos:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n).$$

Balanced case: $T(n) = O(n \log(n))$

e.g. Mergesort $(18, 1, 7, 6, 45, 3, 9, 0)$

① Sort two halves recursively

$(1\ 6\ 7\ 18)(0\ 3\ 9\ 45)$

first half: $T(\frac{n}{2})$ second half: $T(\frac{n}{2})$

② Stitch two halves together

$(1\ 6\ 7\ 18)(0\ 3\ 9\ 45)$

$(0\ 1\ 3\ 6\ \ldots)$

We can implement ② in $O(n)$ time.

Invariant: Smallest element remaining is either smallest in first half, or smallest in second half.

Why? Recursion fairy sorted the halves.
Just keep peeling off the smallest element.

```
MergeSort(L):
 n ← |L|
 If n==1: return L
 Else:
    L1 ← L[1 : ⌈n/2⌉],  L2 ← [⌈n/2⌉+1 : n]
    L1 ← MergeSort(L1)        Runtime: T(n/2)
    L2 ← MergeSort(L2)        Runtime: T(n/2)
    i1 ← 1, i2 ← 1            // pointers to current smallest elements
    For i ∈ [n]:
       If L1[i1] ≤ L2[i2]: L[i] ← L1[i1], i1 ← i1+1
       Else: L[i] ← L2[i2], i2 ← i2+1
```

Runtime: $O(n)$

Proof: Can only increment $O(n)$ times

Inversions (Part II, Section 5.2)

Input: L is list of n elements in $\mathbb{R}$

Output: # of (i,j): $1 \le i < j \le n$

$L[i] > L[j]$ (inversion)

Example $L = [3,5,6,2,4,1]$

inversions: $(i,j) = (1,4)\ (2,4)\ (3,4)$
$(1,6)\ (2,5)\ (3,5)$
$(4,6)\ (2,6)\ (3,6)$
$(5,6)$

Application: Metric between rankings "Kendall τ"
(e.g. comparing preferences)
sort lists "together" so one is sorted.
How many inversions in the other list?

Observation: n^2 time algo: compare all pairs.

Can we do better?

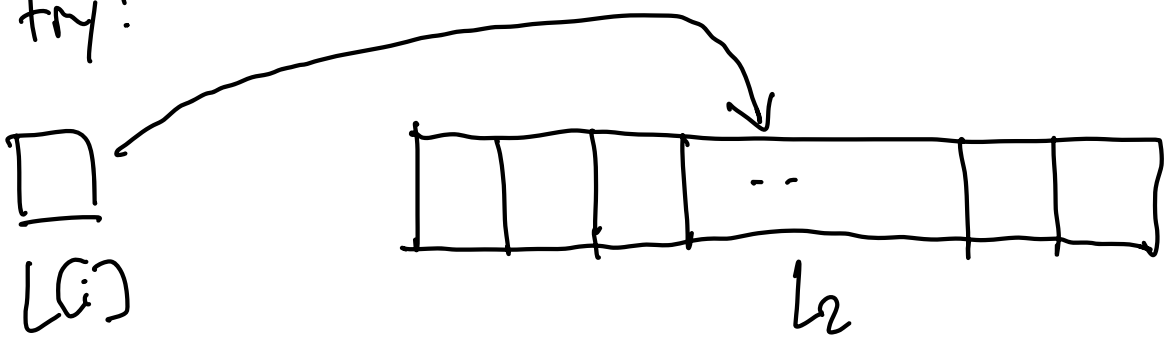
Idea: recursion?

$$\text{Inversions}(L) = \underbrace{\text{Inversions}(L_1)}_{T(\frac{n}{2})} + \underbrace{\text{Inversions}(L_2)}_{T(\frac{n}{2})} + \underbrace{\left|\left\{ i \in [\tfrac{n}{2}], j \in [n] \setminus [\tfrac{n}{2}] : L[i] > L[j] \right\}\right|}_{???}$$

"stitching" step

How to do stitching faster than n^2?

First try:



Time to compute $\left|\left\{ j \in [n] \setminus [\tfrac{n}{2}] \mid L[j] < L[i] \right\}\right|$

- $O(n)$ per i (naive)
- $O(\log(n))$ per i (binary search, if L_2 sorted)

Theme: trade off subproblems with stitching

Inversions recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log(n))$$

recurse on halves

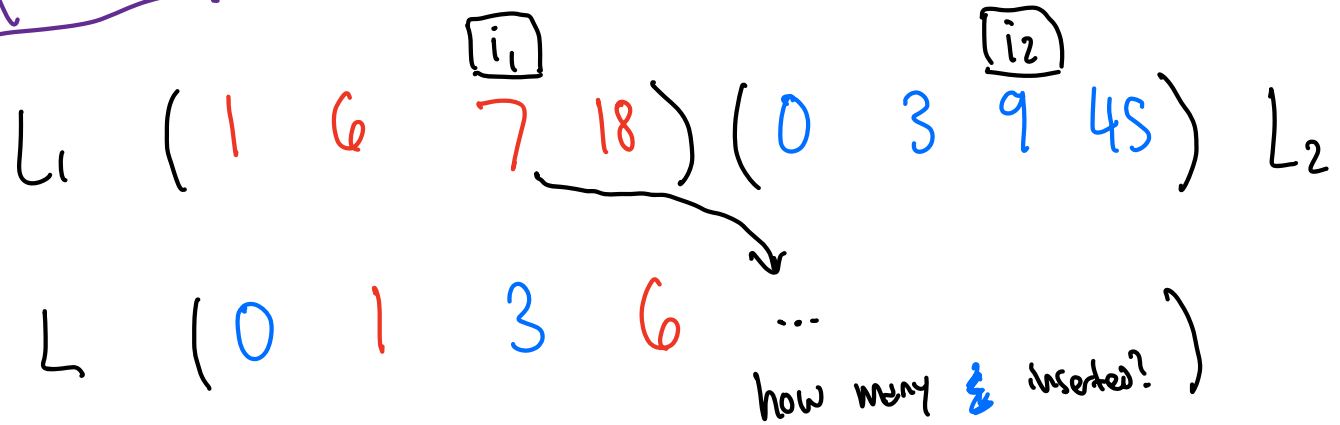
1) Sort L_2
2) binary search for each $L_1(i)$

Balanced case: $T(n) = O(n \log^2(n))$

Improvement: piggy back off MergeSort stitching

```
... // L1, L2 sorted
i1 ← 1, i2 ← 1, count ← 0  // pointers to smallest els, # of inversions
For i ∈ (n):
    If L1(i1) ≤ L2(i2): L(i) ← L1(i1), i1 ← i1+1,
                        count ← count + (i2 - 1)   O(1) time!
    Else: L(i) ← L2(i2), i2 ← i2+1
```

Example



Improved runtime:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log(n))$$

Takeaway 1: Can enforce stronger recursive guarantees

e.g. Sorted sublists + inversion counts.

Takeaway 2: For arrays, pointer / queue tricks save time

e.g. $O(\log(n)) \to O(1)$ per index.

e.g. "sliding window" techniques: will revisit in Part III, Section 2

Selection (Part II, Section 5.3)

Input: L is a list of n elements in $\mathbb{R}$

i is an index $1 \leq i \leq n$

Output: i^{th} largest element in L.

Example

Selection $([10, 7, 16, 3, 2, 80, 1], 4)$

$= 7$

Selection $([-500, -30, -7, 2, 50, 70, 100], 4)$

$= 2$ easier because sorted !!!

Observation 1: $O(n \log(n))$ time algo.

Proof: Sort, take i^{th} largest element

Observation 2: $O(n)$ if i is small $(i = O(1))$

Proof: Compute minimum in $O(n)$ time (just store it). Repeat i times.

Observation 3: $O(n + i \cdot \log(n))$

Proof: heap $O(n)$

ExtractMin() x: $O(i \log(n))$

Main Claim: We can solve selection in

$O(n)$ time, for all $i \in [n]$.

Application: Median in linear time.

Application: Deterministic QuickSort.

Aside QuickSort is a simple sorting algo.

① $x \leftarrow$ findPivot(L)

Easiest choice: random element

② $L \leftarrow$ pivot(L, x) $O(n)$ time.

③ QuickSort two halves recursively.

If we're lucky, x is in the middle, recursion depth $O(\log(n))$

If we're unlucky, x is near the edges, recursion depth $O(n)$

How to use pivoting to solve Selection?

Idea: avoid unlucky pivots using recursion.

```
Selection(L, i):
    n ← |L|
    If n==1: return L[1]
    Else:
① (TBD)     x ← FindPivot(L)
② (k, L) ← Pivot(L, x)        // L is pivoted around x.
                              // x = L[k].
        If k==i: Return x
        Else If k>i:          // search left half
③           Return Selection(L[1:k-1], i)
        Else:                 // search right half
③           Return Selection(L[k+1:n], i-k)
```

Total cost: $T(n) = P(n) + O(n) + T(??)$

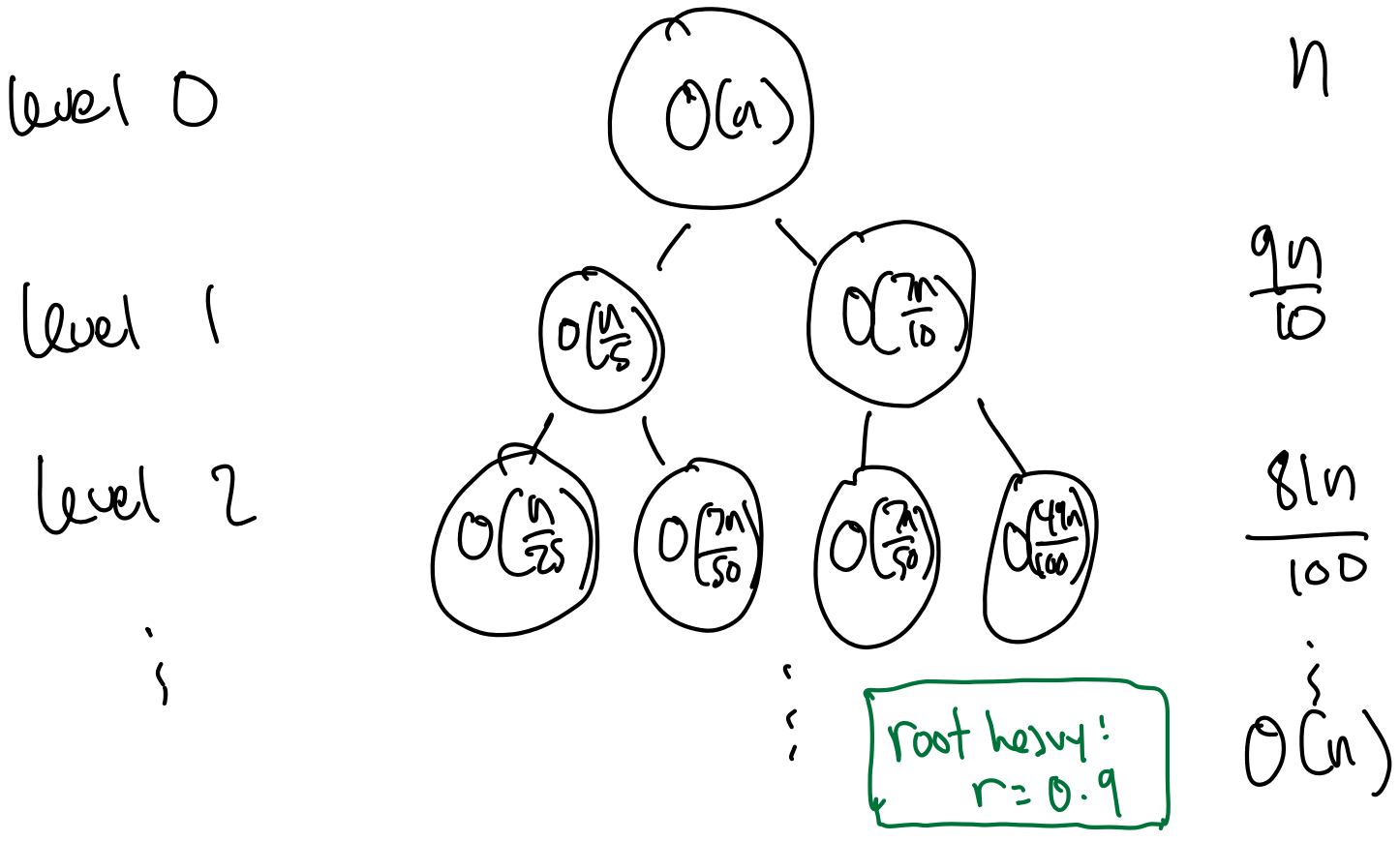
① cost of FindPivot ② ③ cost of recursion

Key claim: there is FindPivot implementation w/

- $P(n) \leq T\left(\frac{n}{5}\right) + O(n)$.
- $?? \leq \frac{7n}{10}$. (not too unlucky)

Finish runtime analysis:

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



Takeaway: Selection (L1:) in $O(n)$ time

☺

Proof of key Claim

"Median of medians" pivot.

Cost

- Split into blocks of size ≤ 5.
- Compute median of each block

$\Big\}$ $O(n)$

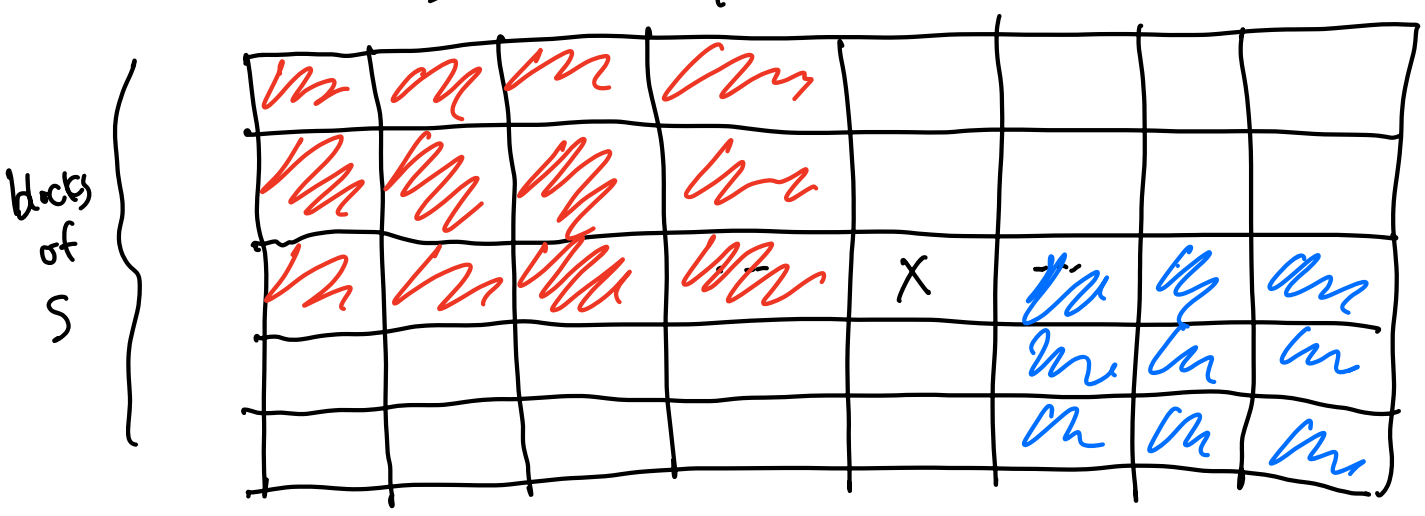
- Compute median of block medians $T(\frac{n}{5})$

  (selection problem on $\frac{n}{5}$ elements)

Magical fact: Median of medians index is

$$\in [0.3n,\ 0.7n]$$

$3/5 \times 1/2 = 3/10$.



▨ means $\leq x$, ▨ means $> x$