Selection & Closest Pair

Selection

Input: n comparable elem. $A[1 \to n]$, $k \in [n]$

Goal: kth smallest element of $A[1..n]$

e.g. median ($k = \lceil n/2 \rceil$) of

10, 56, 77, 91, 70, 24, 36, 27, 64, 50, 65

Obv. approach: sort A, look up kth smallest

$O(n \log n)$

Better?

① Recursive specification

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

① Recursive specification

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

② Implement spec

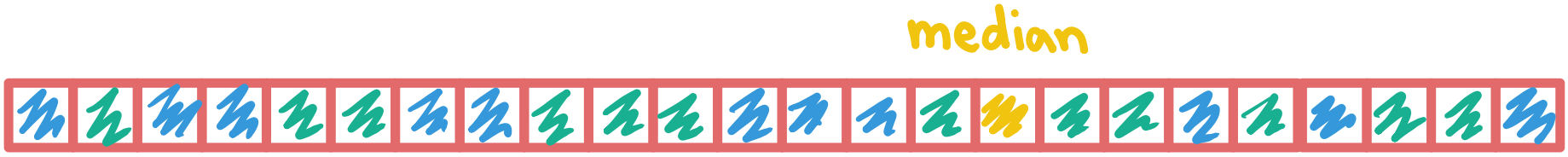
thought experiment: suppose $O(n)$-time subroutine

$$\begin{aligned}\texttt{median}(A[1..n]) &= \texttt{select}(\lceil n/2 \rceil, A[1..n]) \\ &= \text{the } \lceil n/2 \rceil \text{th smallest element out of } A[1..n].\end{aligned}$$

how to use median to find kth element?

"pivot"

① find median



② divide input

③ recurse on appropriate half

① Recursive specification

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec
③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

② Implement spec

thought experiment: suppose O(n)-time subroutine

$$\texttt{median}(A[1..n]) = \texttt{select}(\lceil n/2 \rceil, A[1..n])$$
$$= \text{the } \lceil n/2 \rceil \text{th smallest element out of } A[1..n].$$

`select`$(k, A[1..n])$

/ `select`$(k, A[1..n])$ = the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.) */*

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). *// We assume a linear time subroutine for* `median`$(A[1..n])$.
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$ then return `select`(k, B), where $B[1..\lceil n/2 \rceil - 1]$ is an array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
5. If $k > \lceil n/2 \rceil$ then return `select`$(k - \lceil n/2 \rceil, B)$, where $B[1..n - \lceil n/2 \rceil]$ is an array of the $n - \lceil n/2 \rceil$ elements larger than p.

Running time

$$T(n) = T(n/2) + O(n)$$

	work
n	n
$\vdots$	
$n/2$	$n/2$
$\vdots$	
$n/4$	$n/4$
$\vdots$	
1	1
	$\overline{O(n)}$

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). // *We assume a linear time subroutine for* `median`$(A[1..n])$.
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$ then return `select`(k, B), where $B[1..\lceil n/2 \rceil - 1]$ is an array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
5. If $k > \lceil n/2 \rceil$ then return `select`$(k - \lceil n/2 \rceil, B)$, where $B[1..n - \lceil n/2 \rceil]$ is an array of the $n - \lceil n/2 \rceil$ elements larger than p.

① find median

② divide input

③ recurse on appropriate half

$\Rightarrow O(n)$ time

① Recursive specification

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

② Implement spec

thought experiment: suppose O(n)-time subroutine

$$\texttt{median}(A[1..n]) = \texttt{select}(\lceil n/2 \rceil, A[1..n])$$
$$= \text{the } \lceil n/2 \rceil \text{th smallest element out of } A[1..n].$$

Problem: don't have O(n)-time `median`$(A[1..n])$

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.)* */

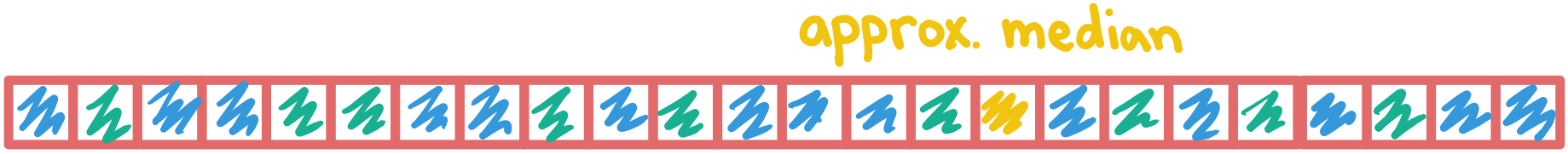
1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). // *We assume a linear time subroutine for* `median`$(A[1..n])$.
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$ then return `select`(k, B), where $B[1..\lceil n/2 \rceil - 1]$ is an array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
5. If $k > \lceil n/2 \rceil$ then return `select`$(k - \lceil n/2 \rceil, B)$, where $B[1..n - \lceil n/2 \rceil]$ is an array of the $n - \lceil n/2 \rceil$ elements larger than p.

O(n) time

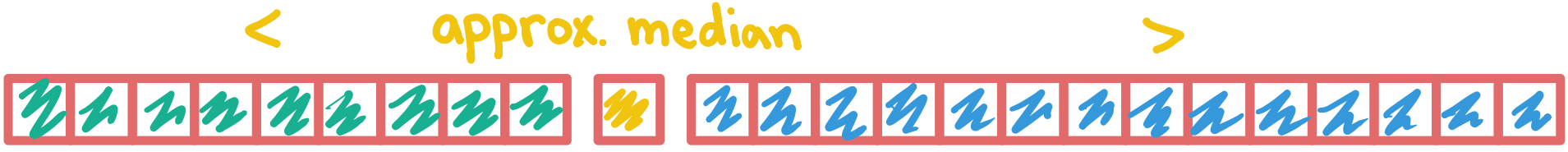
New thought experiment: suppose O(n)-time subroutine

`apx-median`$(A[1..n])$ returns an element $p \in A[1..n]$ with rank between $n/10$ and $9n/10$, in $O(n)$ time.

① find approx. median



② divide input



③ recurse on appropriate half
(somewhat unbalanced)

① Recursive specification

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

② Implement spec

thought experiment: suppose O(n)-time subroutine

`apx-median`$(A[1..n])$ returns an element $p \in A[1..n]$ with rank between $n/10$ and $9n/10$, in $O(n)$ time.

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.)* */

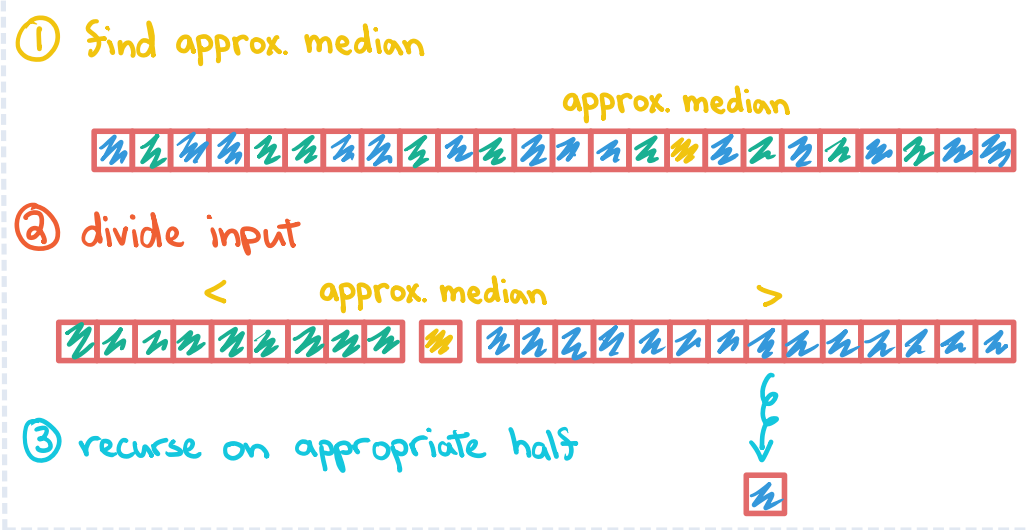
1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `apx-median`(A). // *$O(n)$ time by assumption.*
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ. // $O(n)$.
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select`$(k, A[1..\ell - 1])$, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

Running time

`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `apx-median`(A). // *$O(n)$ time by assumption.*
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ. // *$O(n)$.*
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select`$(k, A[1..\ell - 1])$, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

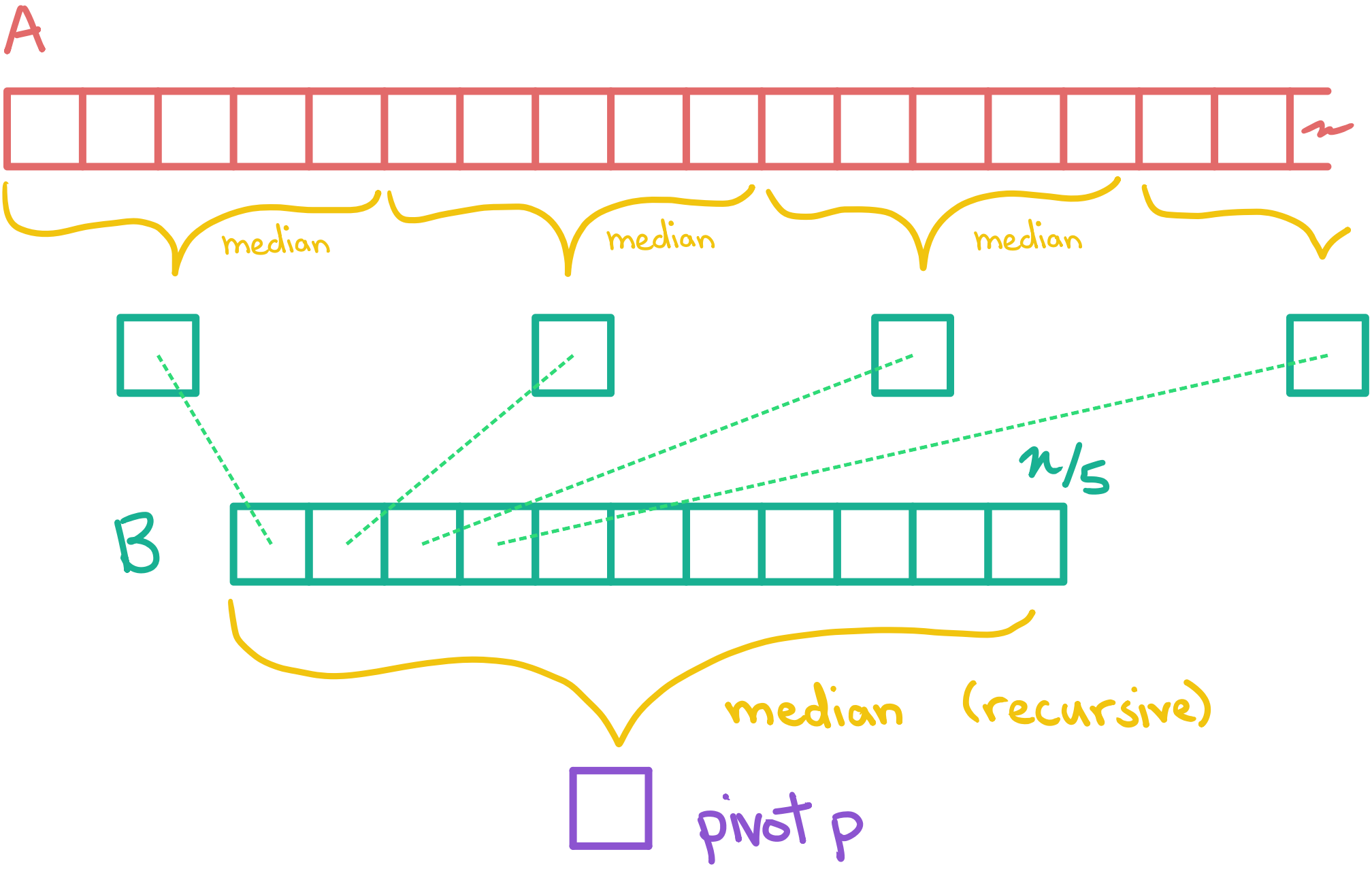


Remains to implement

`apx-median`$(A[1..n])$ returns an element $p \in A[1..n]$ with rank between $n/10$ and $9n/10$, in $O(n)$ time.

"Median of Medians"

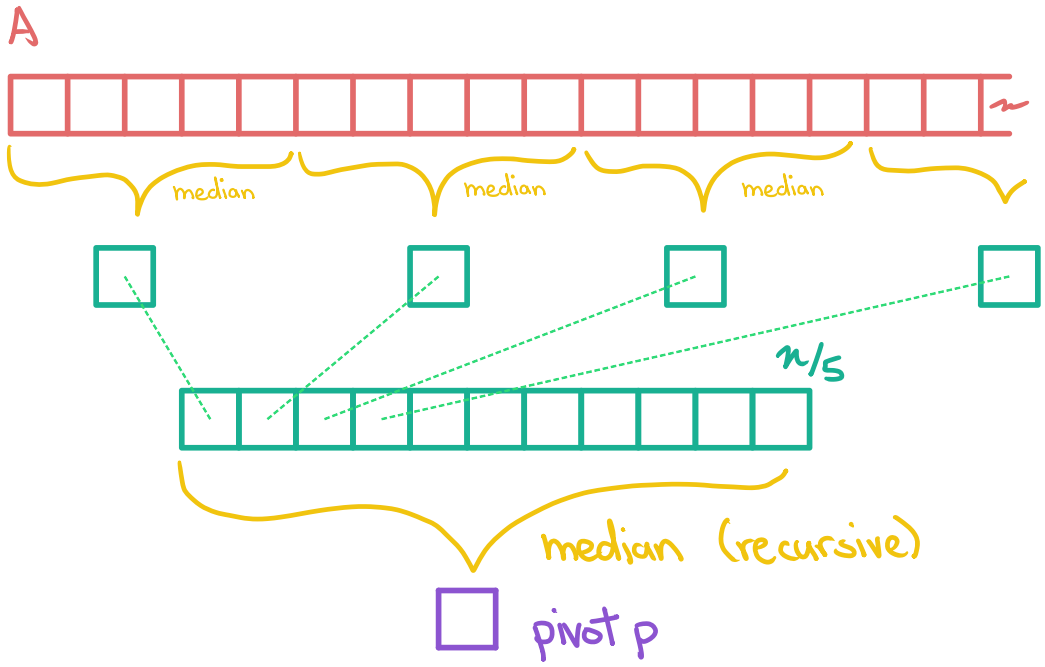
Blum, Floyd, Pratt, Rivest, Tarjan



claim:

M-of-M pivot p has rank

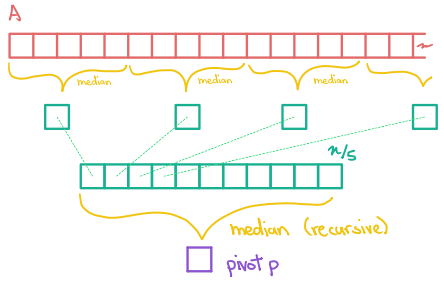
$$\frac{3n}{10} \leq \text{rank}(p) \leq \frac{7n}{10} + 5$$



claim:

M-of-M pivot p has rank

$$\frac{3n}{10} \le \text{rank}(p) \le \frac{7n}{10} + 5$$



B sorted →

B

n/5

each column sorted ↓

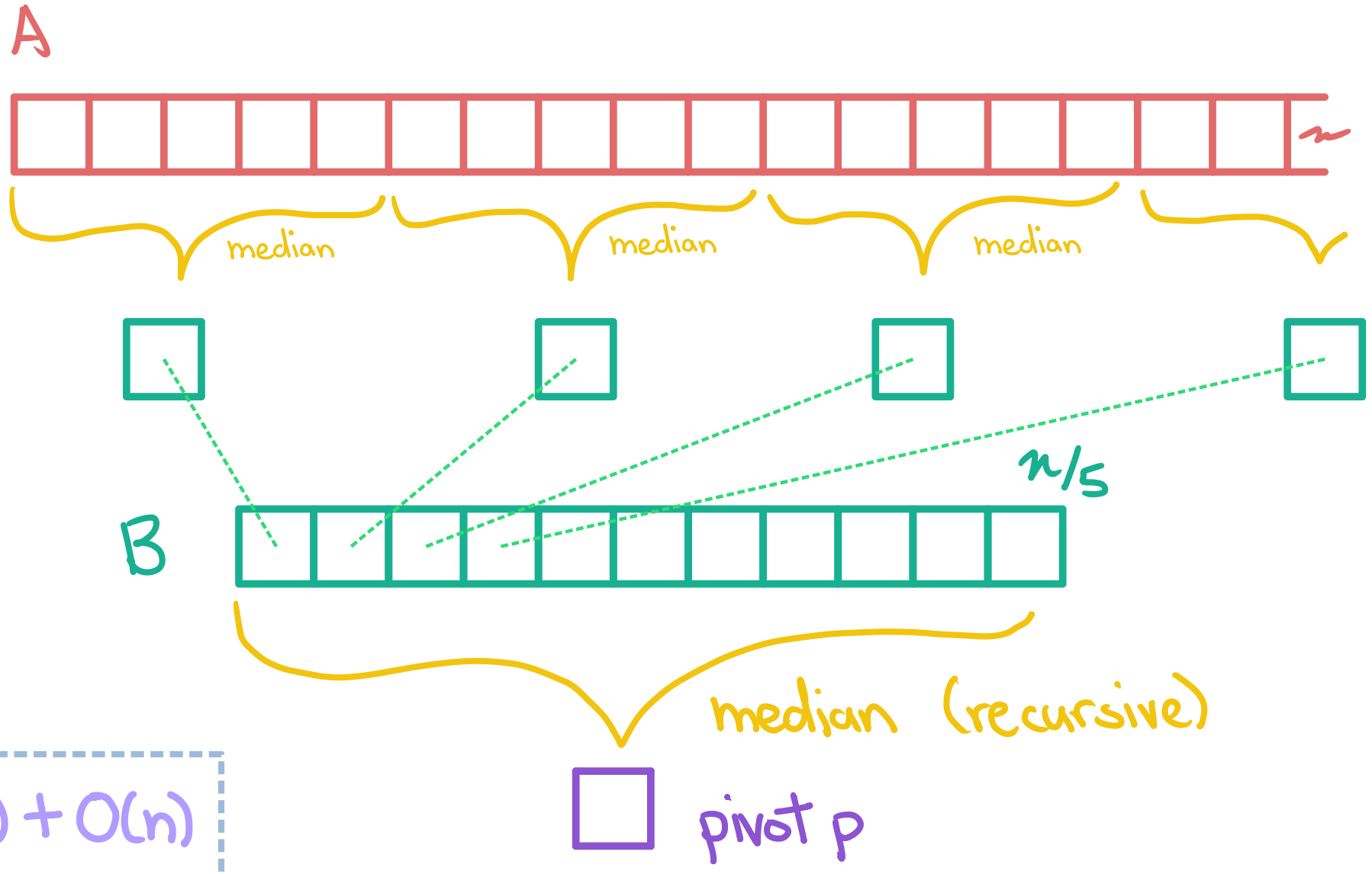
= def. smaller

= def. bigger

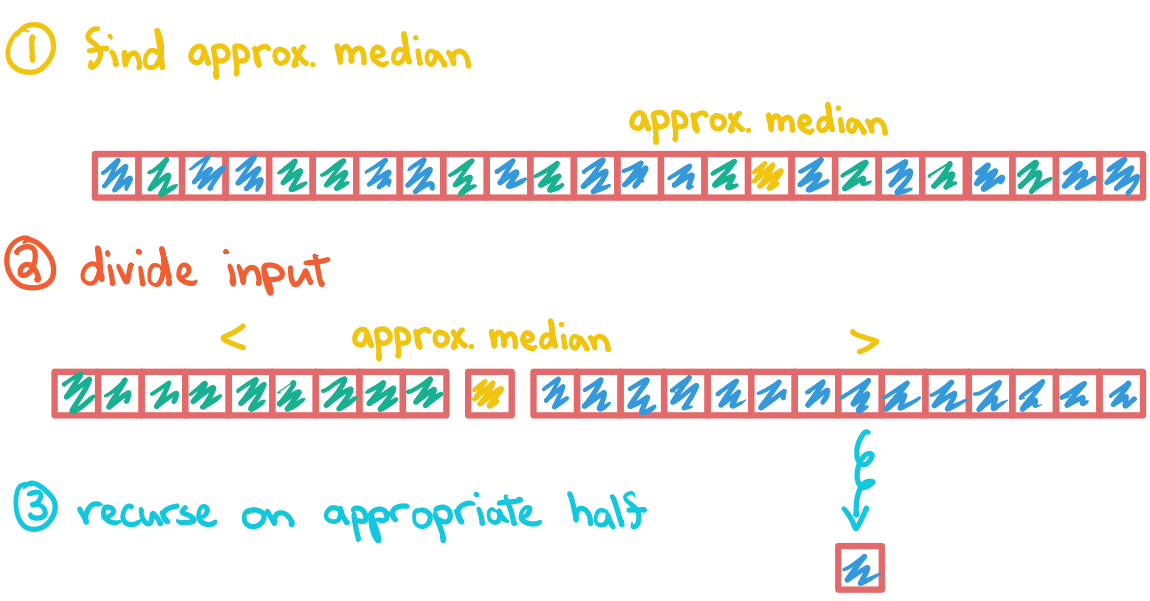
$$\frac{n}{5} \times \frac{1}{2} \times 3 = \frac{3n}{10}$$

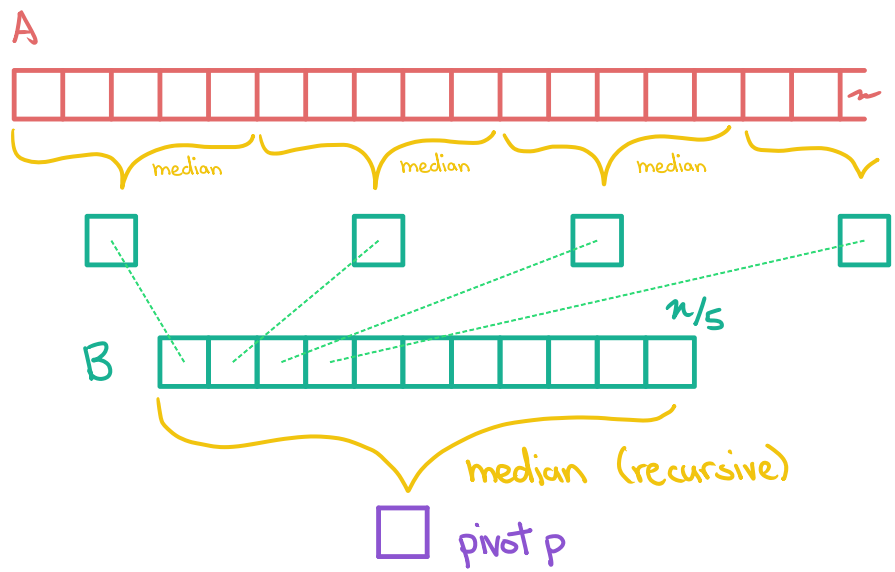
"Median of medians"

Blum, Floyd, Pratt, Rivest, Tarjan



$T(n/5) + O(n)$ time



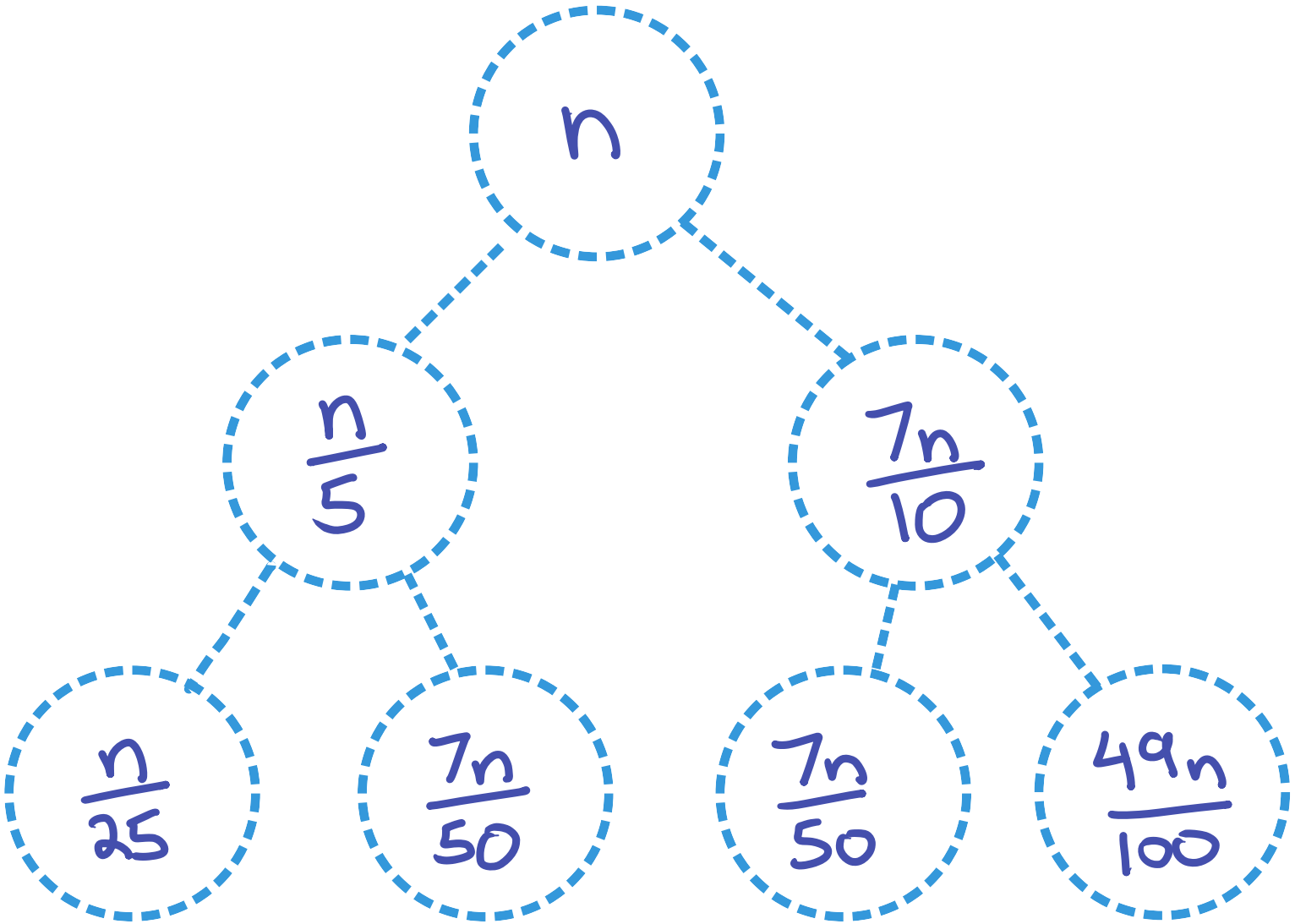


`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median-of-medians`(A).
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ.
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select`(k, B), where $B[1..\ell-1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n-\ell]$ is an array of the $n - \ell$ elements larger than p.

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



n

$\left(\frac{1}{5}+\frac{7}{10}\right)n$

$\left(\frac{1}{5}+\frac{7}{10}\right)^2 n$

$\left(\frac{1}{5}+\frac{7}{10}\right)^3 n$

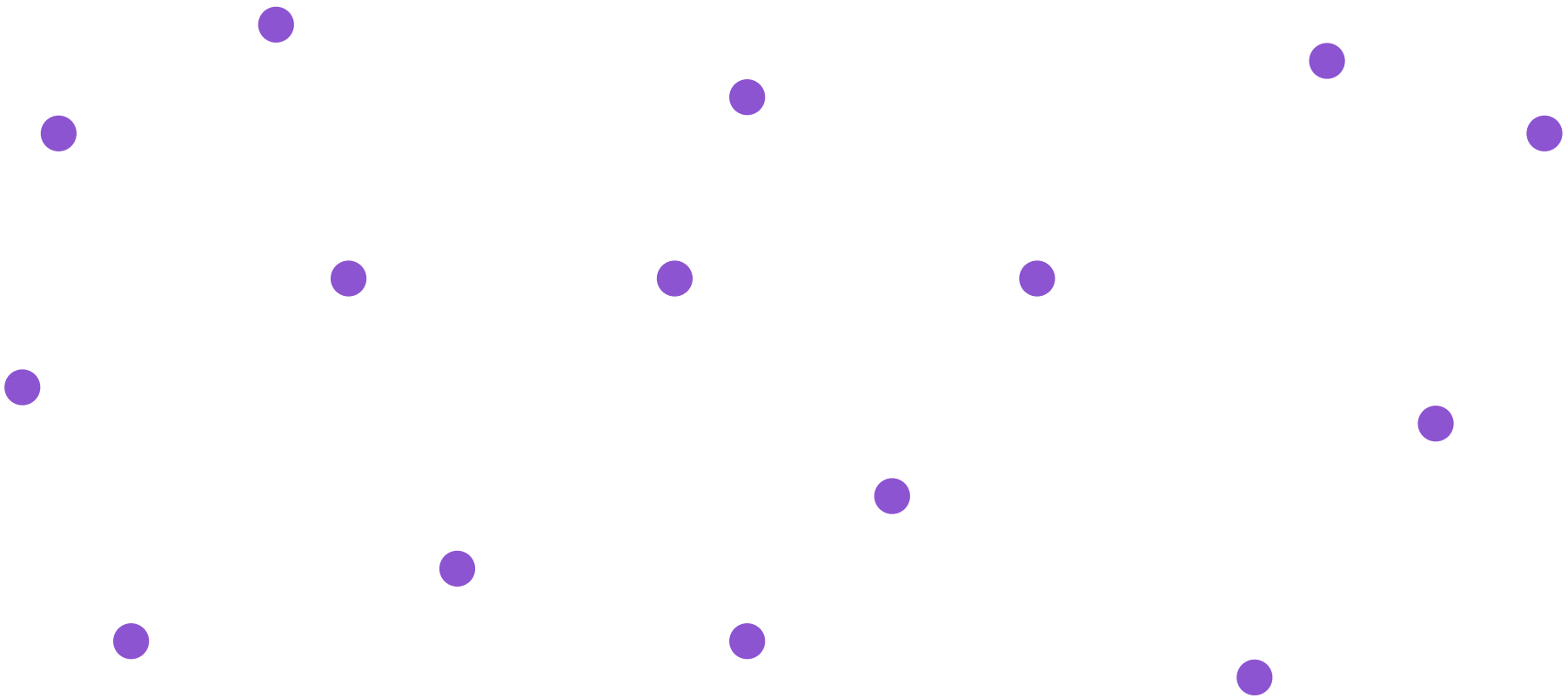
$$\sum_{i=0}^{\infty} \left(\frac{1}{5}+\frac{7}{10}\right)^i n = O(n)$$

Minimum Euclidean distance

Input: n points in $\mathbb{R}^2$

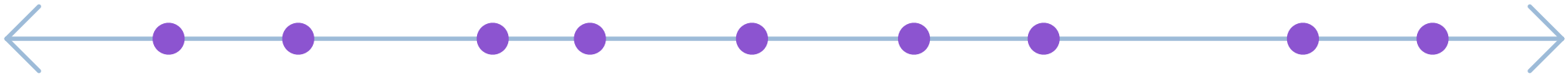
Goal: min distance over all pairs

$\|x-y\| = \sqrt{(x_1-y_1)^2+(x_2-y_2)^2}$ for $x, y \in \mathbb{R}^2$

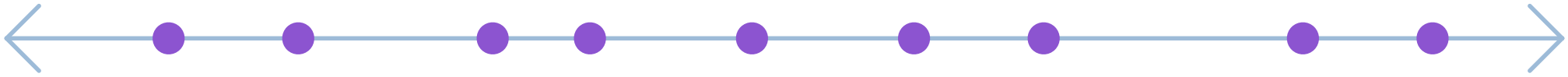


$O(n^2)$ obvious. Better?

Warmup: 1 dimension (all y-coordinates = 0)



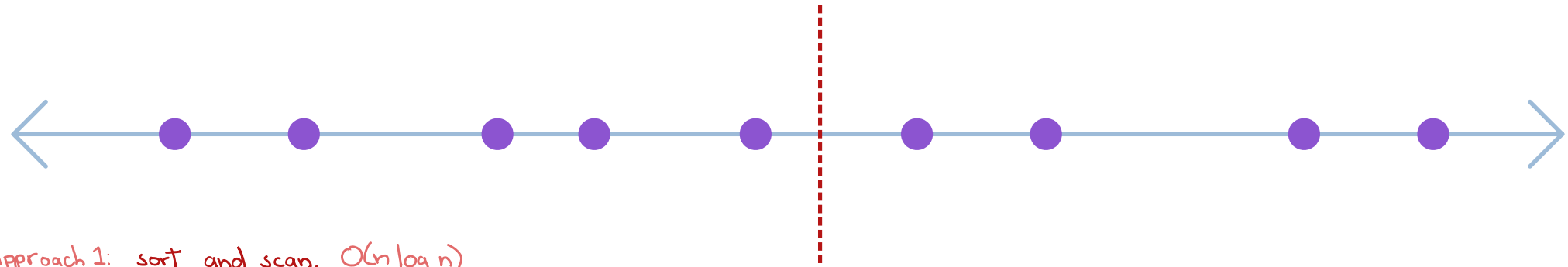
Warmup: 1 dimension (all y-coordinates = 0)



Approach 1: sort and scan. $O(n \log n)$

(hard to generalize to $\mathbb{R}^2$)

Warmup: 1 dimension (all y-coordinates = 0)



Approach 1: sort and scan. $O(n \log n)$
(hard to generalize to $\mathbb{R}^2$)

Approach 2.

Split at median, recurse

Compare w/ min pair "across" split

$O(n \log n)$ (like merge sort)

seems to generalize better

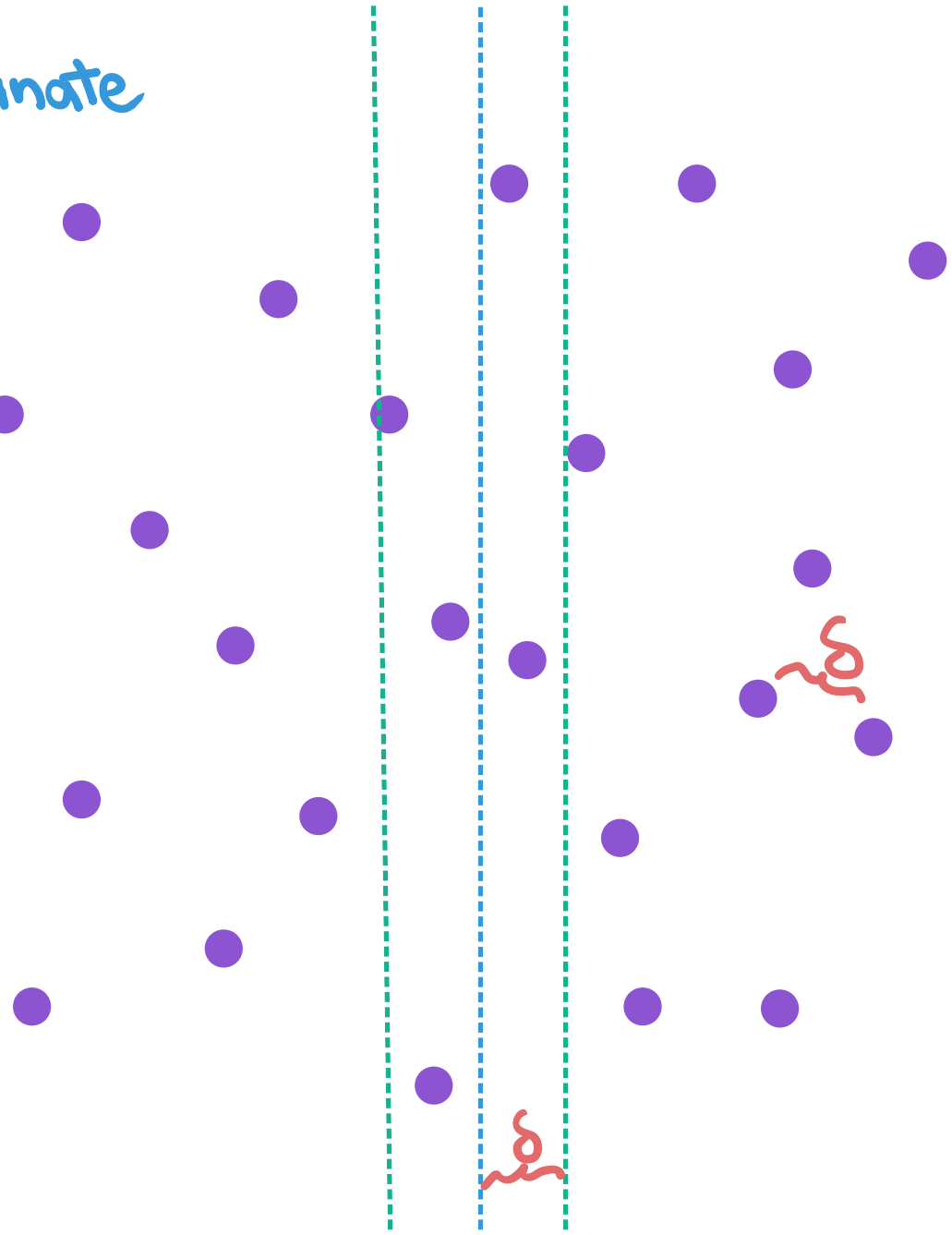
Generalizing to $\mathbb{R}^2$

Approach 2.

Split at median, recurse

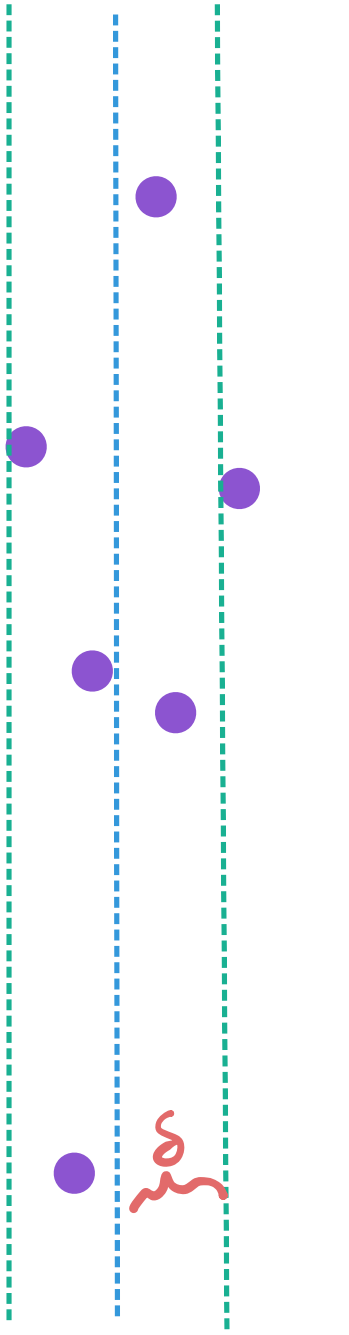
Compare w/ min pair "across" split

1. split at median x-coordinate
2. recurse on each half
   let δ = min distance from the two halves
3. look at "vertical band" of width 2δ around median
4. compute min distance inside band



1. split at median x-coordinate
2. recurse on each half,
   let δ = min dist. from two halves
3. look at "vertical band" of width 2δ around median line
4. compute min distance inside band

Approach 2.
Split at median, recurse
Compare w/ min pair "across" split



Observations

- very narrow
- points on left side have min distance $\geq \delta$
- points on right side have min distance $\geq \delta$

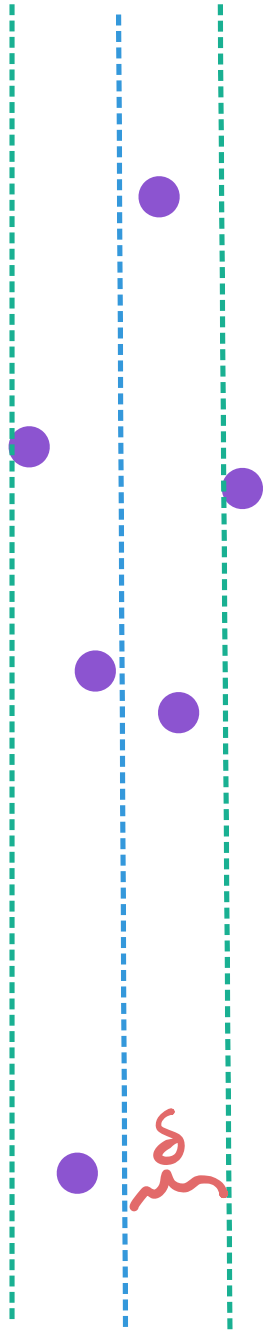
Intuitively: not dense,
fewer comparisons needed

1. split at median x-coordinate
2. recurse on each half,
   let δ = min dist. from two halves
3. look at "vertical band" of width 2δ around median
4. compute min distance inside band

Observations
- very narrow
- points on left side have min distance $\geq \delta$
- points on right side have min distance $\geq \delta$

Intuitively: not dense, fewer comparisons needed

Approach 2.
Split at median, recurse
Compare w/ min pair "across" split



`min-distance(`P`)`

/ We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.) */*

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/ Divide P along the vertical line $x = a$ */*

4. Let $P_1 = \{(x, y) \in P : x < a\}$ and $P_2 = \{(x, y) \in P : x \geq a\}$ // $O(n)$

/ Find the minimum distances within each half P_1 and P_2 recursively. */*

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

/ It remains to find the minimum distance across P_1 and P_2. */*

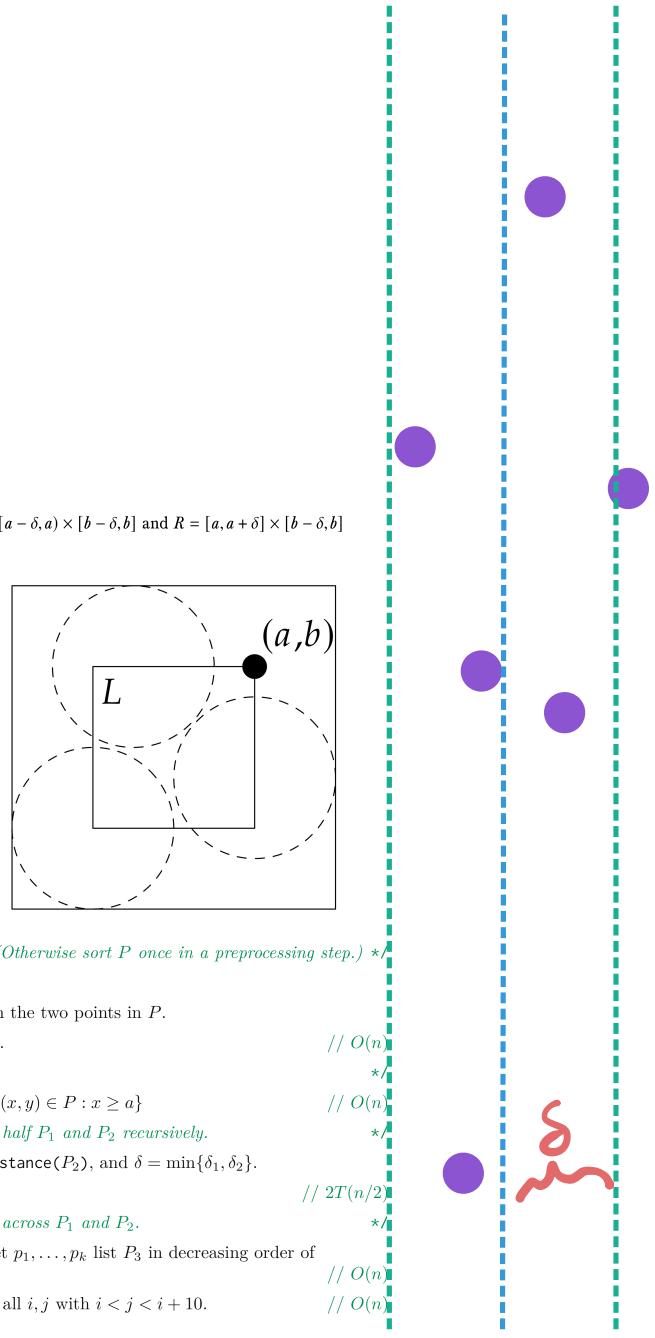
6. Let $P_3 = \{(x, y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

(a = median x-coordinate)

Lemma 11.3. *Let $b \in \mathbb{R}$. There are at most 10 points in P with coordinate in the rectangle $[a-\delta, a+\delta] \times [b-\delta, b]$,*

Observations
- very narrow
- points on left side have min distance $\geq \delta$
- points on right side have min distance $\geq \delta$

Intuitively: not dense, fewer comparisons needed



$L = [a-\delta, a) \times [b-\delta, b]$ and $R = [a, a+\delta] \times [b-\delta, b]$

`min-distance(`P`)`

/* *We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.)* */

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/* *Divide P along the vertical line $x = a$* */

4. Let $P_1 = \{(x,y) \in P : x < a\}$ and $P_2 = \{(x,y) \in P : x \geq a\}$ // $O(n)$

/* *Find the minimum distances within each half P_1 and P_2 recursively.* */

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

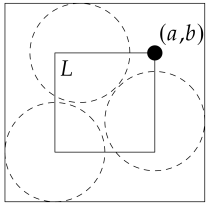
/* *It remains to find the minimum distance across P_1 and P_2.* */

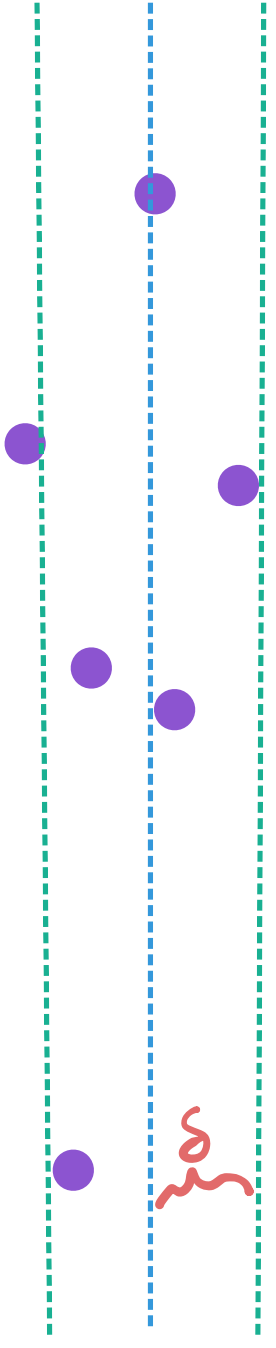
6. Let $P_3 = \{(x,y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

Lemma 11.3. *Let $b \in \mathbb{R}$. There are at most 10 points in P with coordinate in the rectangle $[a-\delta, a+\delta] \times [b-\delta, b]$,*

Observations
- very narrow
- points on left side have min distance $\geq \delta$
- points on right side have min distance $\geq \delta$

Intuitively: not dense, fewer comparisons needed





`min-distance(`P`)`

/ We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.) */*

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/ Divide P along the vertical line $x = a$ */*

4. Let $P_1 = \{(x,y) \in P : x < a\}$ and $P_2 = \{(x,y) \in P : x \geq a\}$ // $O(n)$

/ Find the minimum distances within each half P_1 and P_2 recursively. */*

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

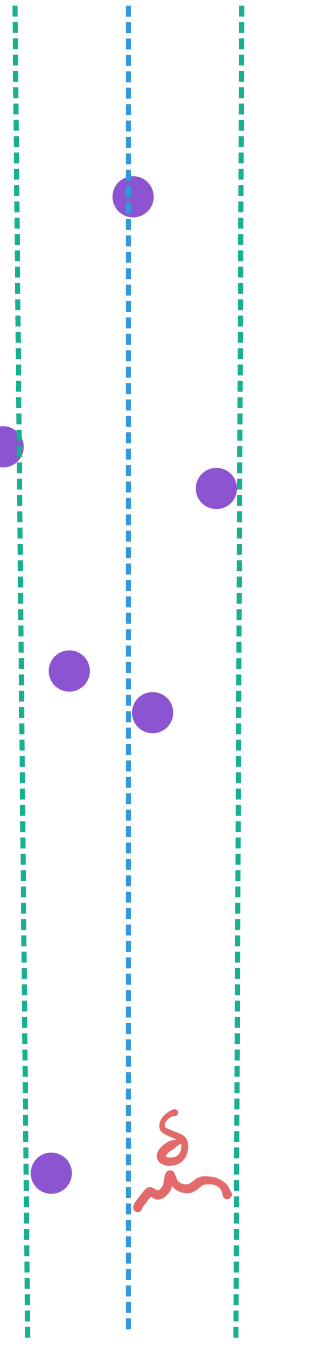
/ It remains to find the minimum distance across P_1 and P_2. */*

6. Let $P_3 = \{(x,y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

Running time

Observations
- very narrow
- points on left side have min distance $\geq \delta$
- points on right side have min distance $\geq \delta$

Intuitively: not dense, fewer comparisons needed



`min-distance(`P`)`

/ We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.) */*

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/ Divide P along the vertical line $x = a$ */*

4. Let $P_1 = \{(x, y) \in P : x < a\}$ and $P_2 = \{(x, y) \in P : x \geq a\}$ // $O(n)$

/ Find the minimum distances within each half P_1 and P_2 recursively. */*

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

/ It remains to find the minimum distance across P_1 and P_2. */*

6. Let $P_3 = \{(x, y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

$T(n) = 2T(n/2) + O(n) = O(n \log n)$

Multiplying n-bit integers

Karatsuba

Input: $x, y \in \{0,1\}^n$

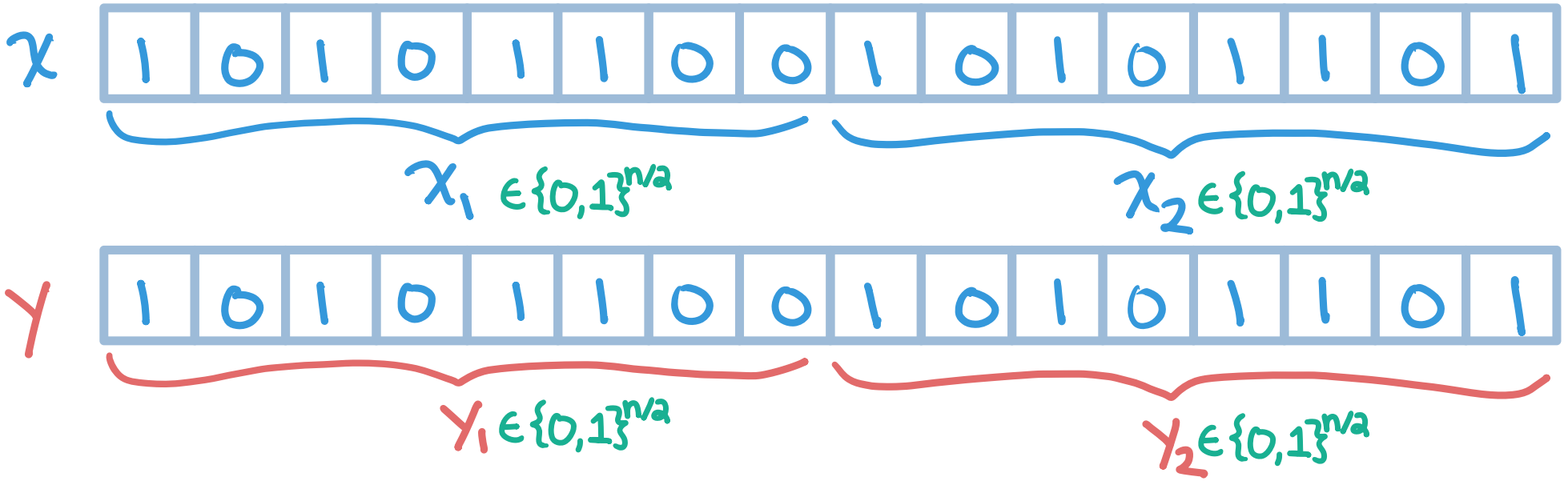
Output: product $(xy) \in \{0,1\}^{2n}$

$$
\begin{array}{r}
1234 \\
\times\ 5678 \\
\hline
9872 \\
8638 \\
+\ \ 6 \\
\hline
\end{array}
$$

$O(n^2)$ time

Better?

Divide bits in half



$$x = x_1 2^{n/2} + x_2, \quad y = y_1 2^{n/2} + y_2$$

$$x \cdot y = x_1 y_1 2^n + (x_1 y_2 + y_1 x_2) 2^{n/2} + x_2 y_2$$

which multiplies 4 pairs of $(n/2)$-bit numbers

$$x \cdot y = x_1 y_1 2^n + (x_1 y_2 + y_1 x_2) 2^{n/2} + x_2 y_2$$

Karatsuba's Trick

already computed

$$(x_1 + x_2)(y_1 + y_2) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2$$

one multiply

middle term

$$x_1 y_2 + x_2 y_1 = x_1 y_1 + x_2 y_2 - (x_1 + x_2)(y_1 + y_2)$$

2 multiplies for the price of 1 (more)!

① Recursive specification

Designing a recursive algorithm

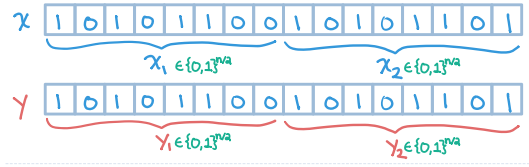
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

multiply($x[1..n], y[1..n]$) = the ($2n$-bit integer) product of x and y (as n-bit integers).

Divide bits in half



$x = x_1 2^{n/2} + x_2, \quad y = y_1 2^{n/2} + y_2$

$x \cdot y = x_1 y_1 2^n + (x_1 y_2 + y_1 x_2) 2^{n/2} + x_2 y_2$

which multiplies 4 pairs of $(n/2)$-bit numbers

Karatsuba's Trick

already computed

$(x_1 + x_2)(y_1 + y_2) = x_1 y_1 + x_1 y_2 + x_2 y_1 + x_2 y_2$

one multiply — middle term

$x_1 y_2 + x_2 y_1 = x_1 y_1 + x_2 y_2 - (x_1 + x_2)(y_1 + y_2)$

2 multiplies for the price of 1 (more)!

① Recursive specification

`select`$(k, A[1..n])$: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \ldots, n\}$, returns the kth smallest element out of $A[1..n]$.

Designing a recursive algorithm
① Recursive specification
- declares input/output of algo
- "contract" algo must fulfill
- induction hypothesis for correctness

② Implement spec

③ Prove correctness
argue algo satisfies recursive spec for all n by induction on n

② Implement spec

`multiply`$(x[1..n], y[1..n])$

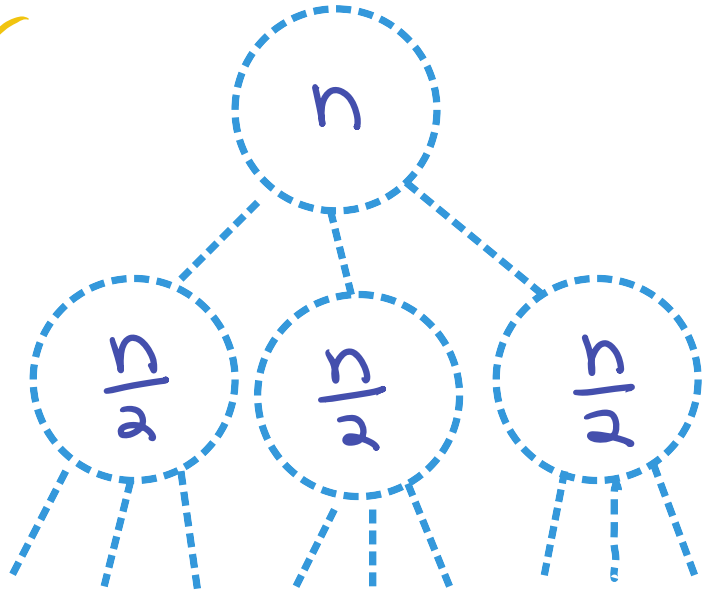
1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.
3. If n is not even then pad x and y with a leading 0, making n even. // $O(n)$.
4. Let $x_1 = x[1..n/2]$, $x_2 = [n/2+1..n]$, $y_1 = y[1..n/2]$, and $y_2 = y[n/2+1..n]$ (as $n/2$-bit integers). // $O(n)$.
5. Let $x_3[1..n/2+1] = x_1 + x_2$ and $y_3[1..n/2+1] = (y_1 + y_2)$ (as $(n/2+1)$-bit integers). // $O(n)$.
6. Let $Z_1 =$ `multiply`(x_1, y_1), $Z_2 =$ `multiply`(x_2, y_2), and $Z_3 =$ `multiply`(x_3, y_3). // $3T((n+2)/2)$.

/* *Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time.* */

7. Return $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$

4. Running time

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



$h = \log_2 n$

n

$3 \times \frac{n}{2} = \frac{3}{2}n$

$(\frac{3}{2})^2 n$

$\vdots$

$(\frac{3}{2})^h n$

`multiply`$(x[1..n], y[1..n])$

1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.
3. If n is not even then pad x and y with a leading 0, making n even. // $O(n)$.
4. Let $x_1 = x[1..n/2]$, $x_2 = [n/2+1..n]$, $y_1 = y[1..n/2]$, and $y_2 = y[n/2+1..n]$ (as $n/2$-bit integers). // $O(n)$.
5. Let $x_3[1..n/2+1] = x_1 + x_2$ and $y_3[1..n/2+1] = (y_1 + y_2)$ (as $(n/2+1)$-bit integers). // $O(n)$.
6. Let $Z_1 =$ `multiply`(x_1, y_1), $Z_2 =$ `multiply`(x_2, y_2), and $Z_3 =$ `multiply`(x_3, y_3). // $3T((n+2)/2)$.

/* *Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time.* */

7. Return $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$

Running time

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

$$\sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i n = O\left(\left(\frac{3}{2}\right)^{\log_2 n} \cdot n\right)$$

$$= O\left(n^{\log_2(3/2)+1}\right)$$

$$= O\left(n^{\log_2(3)}\right)$$

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O\left(n^{\log_2 3}\right)$$

$$\approx n^{1.585}$$

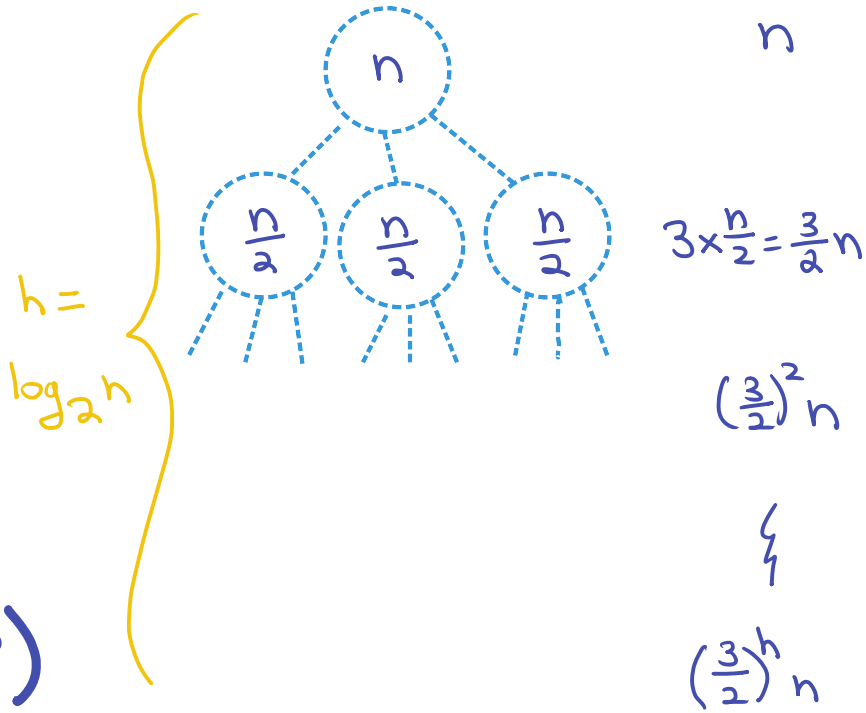
4. Running time

`multiply`$(x[1..n], y[1..n])$

1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.
3. If n is not even then pad x and y with a leading 0, making n even. // $O(n)$.
4. Let $x_1 = x[1..n/2]$, $x_2 = [n/2+1..n]$, $y_1 = y[1..n/2]$, and $y_2 = y[n/2+1..n]$ (as $n/2$-bit integers). // $O(n)$.
5. Let $x_3[1..n/2+1] = x_1 + x_2$ and $y_3[1..n/2+1] = (y_1 + y_2)$ (as $(n/2+1)$-bit integers). // $O(n)$.
6. Let $Z_1 =$ `multiply`(x_1, y_1), $Z_2 =$ `multiply`(x_2, y_2), and $Z_3 =$ `multiply`(x_3, y_3). // $3T((n+2)/2)$.

/* *Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time.* */

7. Return $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$



Chapter 3

Faster algorithms by divide and conquer

This chapter is also about recursive algorithms that rely on induction for correctness. For the problems discussed here, it is relatively easy to see that they have correct polynomial time algorithms. The focus instead is on *faster* polynomial running times. We will use recursion to more aggressively *divide* the input and make the subproblems significantly smaller.

This paradigm is called *divide-and-conquer*. We have seen it before in `merge-sort`. In `merge-sort`, we divided the input in half, and recursively sorted each half separately. The two sorted halves could then be combined very efficiently by merging. `merge-sort` was also used as a vehicle to introduce the recursion tree method for analyzing running times. We will see several more algorithms that leverage insights from recursion trees - introducing clever ideas that do just enough to establish a better recurrence, hence a better running time. In general, the "divide" step generally matches a running theme of "identifying subproblems". The second "conquer" component, so to speak, highlights an additional angle of "combining subproblem solutions".

3.1 Selection

Given a set of n numbers $A[1..n]$, which is the median? The obvious route is to sort all the numbers, and then pick out the $\lceil n/2 \rceil$th number from the sorted order. This takes $O(n \log n)$ time. Can one do better?

We will develop a linear time algorithm for the more general problem of *selection*. Given a set of n numbers $A[1..n]$ in arbitrary order, and an index $k \in [n]$, the goal is to find the kth largest number in A. Again, we can solve this by sorting A, but our goal is to do this faster.

We will develop a recursive algorithm for selection. The first step is define the intent of our recursive algorithm, which establishes the induction hypothesis for the sake of correctness. We do this before developing any algorithm details.

> `select(`$k, A[1..n]$`)`: Given an array $A[1..n]$ of comparable elements and an integer $k \in \{1, \dots, n\}$, returns the kth smallest element out of $A[1..n]$.

We now focus on implementing `select(`$k, A[1..n]$`)` exactly as specified. The high-level idea is to aggressively cutting down the search space — very much like binary search.

Selection via medians. The strategy we pursue is based on the idea of a *pivot*. As a thought experiment to motivate this idea, suppose we have a subroutine `median(`$A[1..n]$`)` that can find (specifically) the median of A in linear time. To formalize this, we define

$$\begin{aligned}\texttt{median}(A[1..n]) &= \texttt{select}(\lceil n/2 \rceil, A[1..n]) \\ &= \text{the } \lceil n/2 \rceil \text{th smallest element out of } A[1..n].\end{aligned}$$

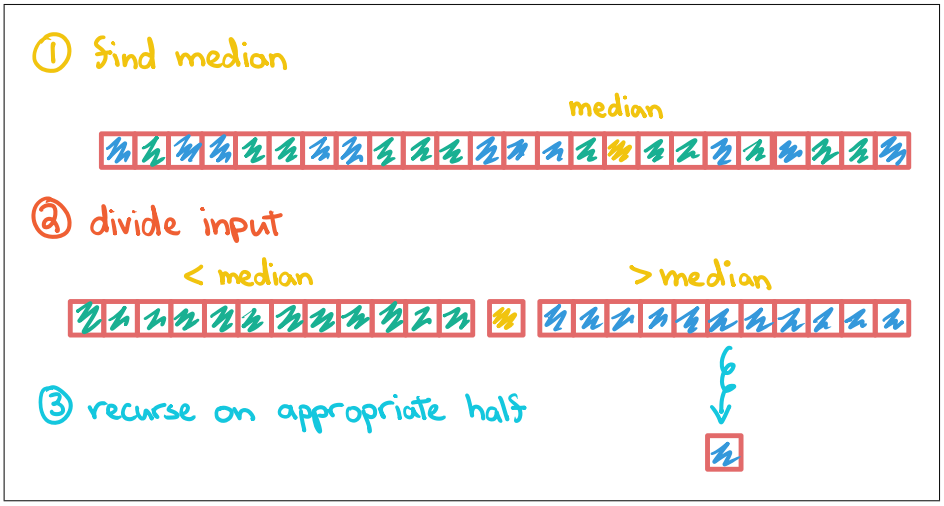
We assume access to a $O(n)$-time subroutine for `median(`$A[1..n]$`)`. `median(`$A[1..n]$`)` is a special case of `select(`$k, A[1..n]$`)`, and it is interesting to ask if the special can be leveraged to accelerate the general case.

Consider an instance of `select(`$k, A[1..n]$`)`. Let p be the median of $A[1..n]$, produced by `median`. We call p the "pivot". Of course if $k = \lceil n/2 \rceil$ then we return p. Otherwise, either $k < \lceil n/2 \rceil$ or $k > \lceil n/2 \rceil$. If $k < \lceil n/2 \rceil$, then we know the rank k element is among the $\lceil n/2 \rceil - 1$ elements smaller than p. We assemble these elements in an array $B[1..\lceil n/2 \rceil - 1]$, and call `select(`$k, B$`)`. Similarly we recurse on the elements bigger than p if $k > \lceil n/2 \rceil$, though now we are looking for the $(k - \lceil n/2 \rceil)$-rank element remaining. By induction we assume that the recursive calls to `select` on smaller input correctly returns the desired element. See fig. 3.1 on the following page for pseudocode.

Consider the running time of `select(`$k, A[1..n]$`)`. Let $T(n)$ denote the maximum running time of `select(`$k, A[1..n]$`)` on any input of size n. `select(`$k, A[1..n]$`)` spends $O(n)$ time to compute the median, and then makes at most one recursive subcall. Because p is always the median, the subcall is a selection over at most $\lfloor n/2 \rfloor$ elements. So we model T recursively as follows:

$$\begin{aligned}T(1) &= 1 \\ T(n) &= T(\lfloor n/2 \rfloor) + O(n).\end{aligned}$$

If we draw out the recursion tree for T, it hardly looks like a tree. Each problem begets one subproblem of half the size. The total work per level is the number of elements in the lone subproblem at that level. The work per level, over all levels, gives a geometrically decreasing sum of the form $n + n/2 + n/4 + \cdots$. As we know



`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties arbitrarily.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). // *We assume a linear time subroutine for* `median`$(A[1..n])$.
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$ then return `select`(k, B), where $B[1..\lceil n/2 \rceil - 1]$ is an array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
5. If $k > \lceil n/2 \rceil$ then return `select`$(k - \lceil n/2 \rceil, B)$, where $B[1..n - \lceil n/2 \rceil]$ is an array of the $n - \lceil n/2 \rceil$ elements larger than p.

Figure 3.1: A (conceptual) divide-and-conquer algorithm for `select`$(k, A[1..n])$ using `median`$(A[1..n])$ to generate a pivot.

(cf. appendix C.2.3), the geometrically decreasing coefficients sum to a constant, and we have

$$T(n) = O(n).$$

That is, assuming a linear time algorithm for finding the median, we can select any item in linear time.

Selection via approximate medians. Of course, we do not know how to implement `median(`$A[1..n]$`)` in $O(n)$ time. Suppose we weakened our assumption to assume we have access to an *approximate* median instead. Formally we assume:

> `apx-median(`$A[1..n]$`)` returns an element $p \in A[1..n]$ with rank between $n/10$ and $9n/10$, in $O(n)$ time.

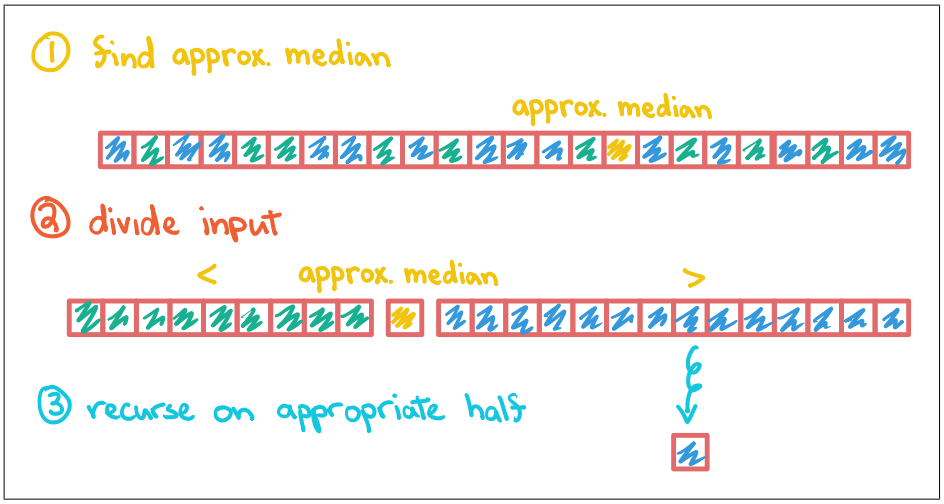
That is, p is always in the middle (4/5)ths of the input. (The choice of 1/10 was arbitrary; the reader may verify afterwards that any constant factor would lead to the same analysis.)

We can use `apx-median(`$A[1..n]$`)` to generate a pivot p in the exact same way as we used `median(`$A[1..n]$`)`. One difference is that after receiving p, we have to calculate its rank ℓ among A, which takes $O(n)$ time. Thereafter we split the input by p, and recurse on the appropriate subproblem. See fig. 3.2 on the next page.

We now have

$$T(n) \leq T(9n/10) + O(n).$$

This recurrence might appear worse than the previous one, since we are only removing 10% of the input rather than half. But when we sketch out a recursion tree, we again find a geometric series except now the coefficient is (9/10) rather than 1/2. A larger coefficient (as long as it remains below 1) increases the running time by only a constant factor. So this second version of `select(`k`,`$A[1..n]$`)`, using a much weaker subroutine to select the pivot, is still a $O(n)$-time algorithm.



`select`$(k, A[1..n])$

/* `select`$(k, A[1..n])$ = *the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.)* */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `apx-median`(A). // *$O(n)$ time by assumption.*
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ. // *$O(n)$.*
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select`$(k, A[1..\ell - 1])$, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

Figure 3.2: A (second, conceptual) divide-and-conquer algorithm for `select`$(k, A[1..n])$ using `apx-median`$(A[1..n])$ to generate a pivot that is approximately the median.

Median-of-medians. We have now reduced linear time selection to approximating the median in linear time. We present such a method, called the *median-of-medians*, invented by Blum, Floyd, Pratt, Rivest, and Tarjan [BFPRT72].[1]

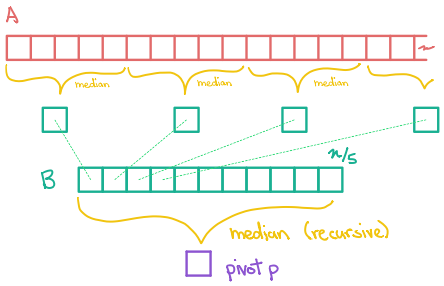
At a high level, [BFPRT72] carefully selects a subset $B[1..m]$ of $A[1..n]$ of size $m = n/5$, and recursively computes the median of B. The process to select B (which we will describe momentarily) takes $O(n)$ time, and is carefully designed to ensure that the median of B is a $(7/10)$-approximate median overall. This gives a recurrence of the form

$$T(n) = T(7n/10) + T(n/5) + O(n). \tag{3.1}$$

Applying the recursion tree method to this recurrence gives $T(n) = O(n)$ – (why?) – linear time!

Let us now explain how [BFPRT72] selects the approximate median. It takes A and partitions it into subintervals of 5 elements each:

$$\{A[1], \dots, A[5]\}, \{A[6], \dots, A[10]\},$$
$$\{A[11], \dots, A[15]\}, \{A[16], \dots, A[20]\}, \dots$$



(The last interval may have fewer than 5.) From each set of 5, we compute the median of those 5 elements in constant time. This takes $O(n)$ time overall and assembles a set $B[1..m]$ of medians, where $m = \lceil n/5 \rceil$. Observing that B is smaller than A by a factor of 5, we recurse on B, computing the "median-of-medians" p. We choose p to be the pivot.

While p is not an exact median, one can show that p has rank roughly inside the range $3n/10$ and $7n/10$ - which is enough to justify the recurrence (3.1) above. Pseudocode is given in fig. 3.3 on the following page. Figure 3.4 on the next page implements `select(`k`,`$A[1..n]$`)` with `median-of-medians(`$A[1..n]$`)`.

We now formally analyze `median-of-medians(`$A[1..n]$`)`.

Lemma 3.1. `median-of-medians(`$A[1..n]$`)` *returns an element p with rank ℓ in the range*

$$3n/10 - 2 \leq \ell \leq 7n/10 + 2.$$

[1] An old joke:

> *Question:* How many Turing Awards does it take to find the median in $O(n)$?
> *Answer:* 4, plus an unofficial founder of Sun Microsystems!

`median-of-medians(`$A[1..n]$`)`

/ Assemble medians of subsets of 5 elements each. */*

1. Divide $A[1..n]$ into $m = \lceil n/5 \rceil$ subintervals of 5 numbers each, and compute the median of each, producing a list $B[1..m]$ for $m = \lceil n/5 \rceil$. (The last interval may have less than 5.)

/ Recursively compute the median of these medians. */*

2. Return `select(`$B, \lceil m/2 \rceil, A[1..n]$`)`

Figure 3.3: The median-of-medians subroutine for approximating the median.

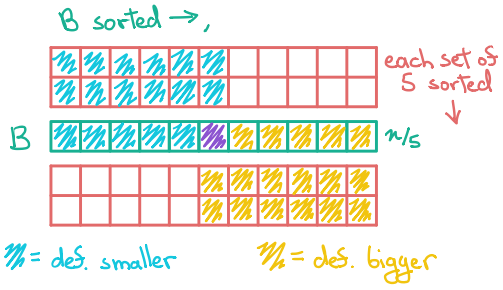
`select(`$k, A[1..n]$`)`

/ `select(`$k, A[1..n]$`)` = the kth smallest element in $A[1..n]$. We assume the elements in A are distinct for simplicity. (Otherwise, break ties consistently.) */*

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median-of-medians(`A`)`.
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ.
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select(`k, B`)`, where $B[1..\ell - 1]$ is an array of the $\ell - 1$ elements smaller than p.
6. If $k > \ell$ then return `select(`$k - \ell, B$`)`, where $B[1..n - \ell]$ is an array of the $n - \ell$ elements larger than p.

Figure 3.4: The linear time divide-and-conquer algorithm for `select(`$k, A[1..n]$`)` by [BFPRT72] using the median-of-medians as a pivot.

Proof. Let us first assume that n is divisible by 5 for simplicity. Consider the array B of "intermediate medians" of size $m = \lceil n/5 \rceil$, of which p is the median. p is greater than or equal to at least $\lceil m/2 \rceil \geq \lceil n/10 \rceil$ of these intermediate medians. Moreover, each intermediate median, as the median of its own set of 5 elements, is greater than 2 more elements from its own set up 5. This $\ell \geq 3\lceil n/10 \rceil \geq 3n/10$.



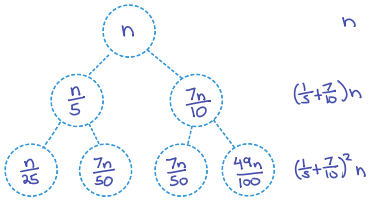
In general when n is not divisible by 5, there may be one set with less than 5 elements. For that deficient subset, we can only guarantee that the intermediate median is at least as big as one element (namely, itself) from its set. So instead we get $\ell \geq 3n/10 - 2$.

Similarly one can show that p is less than or equal to at least $3n/10-2$ elements. □

Running time. Above we asked the reader to try to analyze the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

themselves. We now provide the details for the sake of completeness. A quick recursion tree reveals that the total amount of work at the kth level is



$$(1/5 + 7/10)^k n = (9/10)^k n.$$

Thus the total running time is bounded above by

$$O\left(\sum_{k=0}^{\infty} (9/10)^k n\right) = O(n).$$

Theorem 3.2. *[BFPRT72] Given an array of n comparable elements $A[1..n]$ and an index $k \in [n]$, the kth smallest number of A can be computed in $O(n)$ time.*

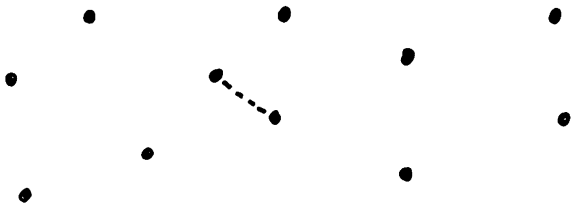
Let us briefly mention a simpler, *randomized* variant that also runs in linear time *on average*. We haven't yet developed the tools to analyze it but the algorithm is simple enough (and perhaps obvious). Rather than by "median-of-medians", simply pick a number at random as a pivot. Divide A into the set of numbers smaller and bigger than the pivot. Recurse appropriately.

The intuition to the randomized analysis is also very simple. With probability 1/2, the pivot is one of the middle $(n/2)$-numbers, and we eliminate at least $n/4$ numbers

from the search. So imagine flipping a bunch of coins. Each coin toss represents one iteration, which takes time linear in the number of elements remaining. Each heads cuts down the input by half. How many coin tosses until the input halves? In expectation, at most a constant. This leads to a $O(n)$ running time and we will formalize the argument later.

The randomized algorithm works better in practice.

3.2 Minimum distance in $\mathbb{R}^2$



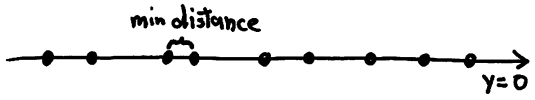
We consider the problem of computing the minimum pairwise distance among a set of points in the plane. The input consists of n points P in $\mathbb{R}^2$. The goal is to find the minimum Euclidean distance between any two points $p, q \in P$. Here we recall that for two points $p = (p_1, p_2)$ and $q = (q_1, q_2)$, the Euclidean distance $\|p - q\|$ is given by

$$\|p - q\| = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

We assume for simplicity that all the coordinates of all the points are distinct.[2] Of course there is an $O(n^2)$-time algorithm where we simply calculate the distance of every pair. The question is whether one could do better. In anticipation of a recursive algorithm, we first declare a recursive spec.

> `min-distance(`P`)` = the minimum Euclidean distance between any pair of points in P (or $+\infty$ if there are less than 2 points in P).

The goal is to implement `min-distance(`P`)` exactly as described above.

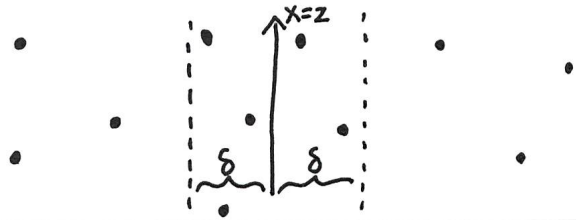


[2]The fancy way to state this assumption is to say that the points are in "general position". In practice, we can often assume general position by first perturbing the points by a sufficiently small amount. We also point out that the algorithms here are simple enough to adapt to duplicate coordinates. We assume general position to keep the algorithm clean and focus on the high-level ideas.

To build some intuition, it is helpful to consider the 1-dimensional case; or equivalently, assume that all points have the same y-coordinate, (say) $y = 0$. (We continue to assume the x-coordinates are distinct.) The most obvious approach is to sort the points by x-coordinate, and then scan and check all the consecutive pairs in sorted order. This takes $O(n \log n)$ time. This is very good but it does not seem to generalize to general y-coordinates. In $\mathbb{R}^2$, the minimum distance is not necessarily attained by a consecutive pair of points when listed in increasing order of x.

A second approach to the one-dimensional ($y = 0$) setting is to first find the median x-coordinate a. (We can compute a in $O(n)$ time via `select` from section 3.1). Let P_1 be the subset of points with x-coordinate $< a$ and P_2 be the subset of points with x-coordinate $\geq a$. We recursively compute the minimum distance in P_1 and P_2. Let δ be the minimum of the two values returned. δ is the minimum distance among pairs of points within P_1 and within P_2; it remains to check the minimum distance of pairs *across* the two halves. To this end, we calculate the distance between the right-most-point on the left half P_1, and the left-most-point on the right half P_2. We return the minimum of this distance and δ.

The second approach almost generalizes to $\mathbb{R}^2$, except for the last step where we get the minimum distance across the two halves. The minimum distance across P_1 and P_2 is not necessarily between the two points closest to the dividing line $x = a$.



We do know that that any point in the minimum cross pair has x-coordinate within δ of a. (Otherwise the x-coordinate alone makes the distance of the cross-pair $> \delta$). Let

$$P_3 = \{(x, y) \in P : |x - a| \leq \delta\}$$

gather up all the points in P whose x-coordinates are within δ of a. Now consider the following geometric facts.

1. P_3 lies in a narrow, vertical band of width 2δ.
2. Any two points in $P_1 \cap P_3$ have distance at least δ, and any two points in $P_2 \cap P_3$ have distance at least δ. This generally forces P_3 to be spread apart.

`min-distance(`P`)`

/ We assume P is sorted by y-coordinate. (Otherwise sort P once in a preprocessing step.) */*

1. If $n \leq 1$ then return $+\infty$.
2. If $n = 2$ then return the distance between the two points in P.
3. Let a be the median of the x-coordinates. // $O(n)$

/ Divide P along the vertical line $x = a$ */*

4. Let $P_1 = \{(x, y) \in P : x < a\}$ and $P_2 = \{(x, y) \in P : x \geq a\}$ // $O(n)$

/ Find the minimum distances within each half P_1 and P_2 recursively. */*

5. Let $\delta_1 =$ `min-distance(`P_1`)`, $\delta_2 =$ `min-distance(`P_2`)`, and $\delta = \min\{\delta_1, \delta_2\}$. // $2T(n/2)$

/ It remains to find the minimum distance across P_1 and P_2. */*

6. Let $P_3 = \{(x, y) \in P : |x - a| < \delta\}$ and let $p_1, \ldots, p_k$ list P_3 in decreasing order of y-coordinate. // $O(n)$
7. Let δ_3 be the minimum of $\|p_i - p_j\|$ over all i, j with $i < j < i + 10$. // $O(n)$
8. Return $\min\{\delta, \delta_3\}$.

Figure 3.5: Computing the minimum distance in $\mathbb{R}^2$ by divide-and-conquer along the horizontal axis.

Combining the two facts leads to the following observation (formalized in lemma 3.3):

> For any point $p \in P_3$, there are at most a constant number of other points with y-coordinate within δ of p.

We take advantage of this by sorting P_3 by y-coordinate, and only compute the distance between pairs of points that are within 10 ranks of one another in the sorted list. (By lemma 3.3, 10 is a sufficiently large constant.)

Pseudocode of the divide-and-conquer algorithm described above is given in fig. 3.5. As a relatively minor point, instead of sorting by y-coordinate in each recursive call, we can sort all of P once in a preprocessing step. We assume that P is already sorted by y-coordinate in the implementation in fig. 3.5.

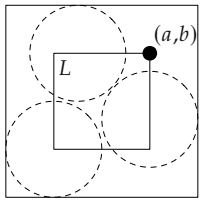
Lemma 3.3. *Let $b \in \mathbb{R}$. There are at most 10 points in P with coordinate in the rectangle $[a - \delta, a + \delta] \times [b - \delta, b]$,*

Proof. The following is an example of a "packing argument", which is common in computational geometry.

We split the rectangle into two squares

$$L = [a - \delta, a) \times [b - \delta, b] \text{ and } R = [a, a + \delta] \times [b - \delta, b].$$

For each half we show that there are at most 5 points.



Consider the left half L. Any pair of points in L, being both in P_1, has distance at least δ. For each point $p \in P \cap L$, draw a ball B_p with radius $\delta/2$ centered at p. Each ball B_p has area $\pi\delta^2/4$, and lies in the "padded" rectangle $[a - 1.5\delta, a + .5\delta] \times [b - 1.5\delta, b + .5\delta]$ of area $2\delta \times 2\delta = 4\delta^2$. These balls are also interior disjoint because the minimum distance between their centers is δ. The number of these balls that can fit in the padded rectangle, by comparing areas, is at most

$$\frac{4\delta^2}{\pi\delta^2/4} = \frac{16}{\pi} \approx 5.09.$$

Since the maximum number is also integer, we conclude there can be at most 5 balls, hence 5 points in $P \cap L$.

By the same argument (now applied to P_2, instead of P_1) there are at most 5 points in $R \cap P$. □

The overall proof of correctness is as follows. We will prove that fig. 3.5 correctly implements `min-distance(`P`)` as specified above by induction on $|P|$. The base cases are when $|P| \leq 2$; if $|P| < 1$ then the distance is automatically $+\infty$; if $|P| = 2$, there is only one pairwise distance to consider and that is the value returned. Consider the general case where $|P| > 2$. We assume by induction that the implementation is correct for all strictly smaller point sets.

After dividing the input size into P_1 and P_2, we know that the minimum distance is either within P_1, within P_2, or across P_1 and P_2. By induction we assume that `min-distance(`P_1`)` and `min-distance(`P_2`)` return the (true) minimum distance within P_1 and P_2, respectively. For pairs going across P_1 and P_2, it suffices to compute the minimum distance in P_3 since otherwise the difference in x-coordinate is at least δ. Within P_3, by lemma 3.3, it suffices to compare each point $p \in P_3$ with the 10 points with closest y-value, which is exactly what the algorithm does. So the algorithm also

finds the minimum distance within P_3 correctly. The minimum distance from the three sets P_1, P_2, and P_3 gives the minimum distance of all of P, as claimed.

As for the running time, we have two recursive calls to point sets of size $n/2$, plus $O(n)$ additional work (as annotated in the code). This gives a recursion of the form

$$T(n) = 2T(n/2) + O(n).$$

This is the same recurrence as `merge-sort` and is solved by $T(n) = O(n \log n)$.

3.3 Multiplication

Our first example (excluding merge-sort) is integer multiplication. Let $x, y \in \{0,1\}^n$ be two n-bit integers, we want to compute the product $x \times y$. The standard approach to multiplication I learned in grade school is an $O(n^2)$-time algorithm, that multiplies every bit in x with every bit in y (along with some addition and carrying). In this section we will use divide-and-conquer to obtain a faster algorithm running in roughly $O(n^{1.585})$.

Improving the polynomial dependency on the number of bits may not seem very practical, since most programs work with fixed bit widths, and modern CPU's have specialized hardware dedicated to multiplying fixed width integers very quickly. Our algorithm will mostly be useful for extremely large integers. (Maybe our tricks could also help the CPU designers). Still we present integer multiplication first for the following reasons (besides the usual punt that "this is theory"). First, both the problem and the algorithm are (in our opinion) technically simpler than the other examples that follow. We understand all of the mathematics concerning integer multiplication and the only novelties here are algorithmic.

Second, the algorithm we present has some historical significance. Integer multiplication is so commonplace that we would seem to know everything about it, and one might not suspect any deficiencies in the $O(n^2)$ approach. So in the 1950's the famous mathematician Andrey Kolmogorov conjectured that it was the best possible algorithm; that is, he conjectured a $\Omega(n^2)$ lower bound for this problem. But in 1960 Anatoly Karatsuba found a substantially faster algorithm and this is the algorithm that we discuss. It was a surprising revelation that more broadly suggests that the "obvious" approaches to many other well-understood problems can be improved.

We first explain Karatsuba's algorithm at a high level. Let $x, y \in \{0,1\}^n$ be two n-bit numbers we want to multiply. We assume n is even for simplicity (and otherwise pad both x and y with a leading 0). Let us break the bits in two halves $x = (x_1, x_2)$ and $y = (y_1, y_2)$, where $x_1 \in \{0,1\}^{n/2}$ (resp. $y_1 \in \{0,1\}^{n/2}$) are the

$n/2$ most significant bits of x (resp. of y), and $x_2 \in \{0,1\}^{n/2}$ (resp. $y_2 \in \{0,1\}^{n/2}$) represent the remaining, $n/2$ least significant bits of x, (resp. y). That is,

$$x = x_1 2^{n/2} + x_2 \text{ and } y = y_1 2^{n/2} + y_2.$$

Nothing special. We have

$$xy = \Big(x_1 2^{n/2} + x_2\Big)\Big(y_1 2^{n/2} + y_2\Big) = x_1 y_1 2^n + (x_1 y_2 + x_2 y_1) 2^{n/2} + x_2 y_2.$$

Again, nothing special. We have broken one multiplication of n-bit numbers into 4 multiplications of $n/2$-bit numbers which has no effect on the running time. But consider the middle term $x_1 y_2 + x_2 y_1$. Observe that

$$x_1 y_2 + x_2 y_1 = (x_1 + x_2)(y_1 + y_2) - x_1 y_1 - x_2 y_2.$$

The equation above rewrites two products as three, which seems like a step backwards. *Except two of them, $x_1 y_1$ and $x_2 y_2$, are already being computed.* With only *one more multiplication* (of $(x_1 + x_2)(y_1 + y_2)$), plus some $O(n)$-time addition and subtraction, we replace *two multiplications* in $x_1 y_2 + x_2 y_1$.[3]

Let's make this concrete in an algorithm. We are designing a recursive algorithm so first we declare our recursive spec / induction hypothesis.

> `multiply(`$x[1..n]$`,`$y[1..n]$`)` = the ($2n$-bit integer) product of x and y (as n-bit integers).

In fig. 3.6, we implement `multiply(`$x[1..n]$`,`$y[1..n]$`)` as a divide-and-conquer algorithm leveraging the reduction to 3 integer multiplications of (roughly) half the bits.

3.3.1 Running time analysis

Let $T(n)$ denote the running time of `multiply(`$x[1..n]$`,`$y[1..n]$`)` on two n-bit integers. As annotated in the code, the algorithm consists of three recursive calls to multiply integers with at most $n/2+2$ bits, plus $O(n)$ additional work. This gives the recurrence

$$T(n) \leq 3T(n/2 + 2) + O(n) \tag{3.2}$$

for n larger than a constant (e.g., 5), and $O(1)$ time for n below this constant. This is would be easy for a recursion tree were it not for the extra "+2" in the recursive call. Consider instead the cleaner recurrence

$$S(n) \leq 3S(n/2) + O(n) \tag{3.3}$$

$S(n)$ is clearly similar to $T(n)$; morally, we are interested in large values of n, and the omitted additive factor of 2 is diminutive compared to $n/2$ for large n.

[3] Note that $(x_1 + x_2)(y_1 + y_2)$ multiples two $(n/2 + 1)$-bit integers, instead of $n/2$, which makes our analysis a little messier.

`multiply`$(x[1..n], y[1..n])$

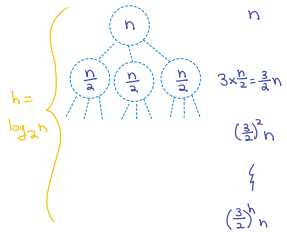
1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.
3. If n is not even then pad x and y with a leading 0, making n even. // $O(n)$.
4. Let $x_1 = x[1..n/2]$, $x_2 = [n/2+1..n]$, $y_1 = y[1..n/2]$, and $y_2 = y[n/2+1..n]$ (as $n/2$-bit integers). // $O(n)$.
5. Let $x_3[1..n/2+1] = x_1 + x_2$ and $y_3[1..n/2+1] = (y_1 + y_2)$ (as $(n/2+1)$-bit integers). // $O(n)$.
6. Let $Z_1 =$ `multiply`(x_1, y_1), $Z_2 =$ `multiply`(x_2, y_2), and $Z_3 =$ `multiply`(x_3, y_3). // $3T((n+2)/2)$.

/* *Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time.* */

7. Return $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$

Figure 3.6: Karatsuba's algorithm for integer multiplication.

A clean recursion tree. Let us suppose for the moment that $S(n)$ does model Karatsuba's algorithm. A quick sketch of a recursion tree for $S(n)$ shows that:



1. There are 3^k problems on the kth level each contributing $n/2^k$ work. This gives $(3/2)^k n$ work per level.
2. The height h is bounded above by $h \le \log_2(n) + O(1)$.

We have

$$S(n) \le \sum_{i=1}^{h} \left(\begin{matrix} \text{total work} \\ \text{on level } i \end{matrix} \right) = \sum_{i=1}^{h} \left(\frac{3}{2}\right)^i n \overset{\text{(a)}}{=} O\left(\left(\frac{3}{2}\right)^h n\right)$$
$$= O\left(\left(\frac{3}{2}\right)^{\log_2(n)+1}\right) = O\left(n^{\log_2(3)}\right) \approx O\left(n^{1.585}\right).$$

(a) is because the sum of coefficients is a geometrically increasing sequence (cf. appendix C.2.3).

Justifying the clean recurrence. Now we have analyzed a "nice" recurrence for $S(n)$ and suggested without proof that this is representative of $T(n)$. Let us now furnish the details.

We define $S(n) = T(n + \alpha)$ for a constant $\alpha > 0$ TBD. We want to choose α so that $S(n) = T(n+\alpha)$ satisfies the nice recurrence (3.3). This can be done by choosing $\alpha = 4/3$ but let us explain how to deduce this value of α.

On one hand, the given recurrence for T gives

$$S(n) = T(n - \alpha) \leq 3T\Big(\frac{n - \alpha + 2}{2}\Big) + O(n).$$

Meanwhile substituting $S(n) = T(n + \alpha)$ into our target recurrence (3.3) gives

$$S(n) \leq 3S(n/2) + O(n) = 3T(n/2 + \alpha) + O(n).$$

To try to make the recurrences equal, let us choose α so that

$$\frac{n - \alpha + 2}{2} = \frac{n}{2} + \alpha;$$

This is solved by $\alpha = 4/3$. Now substituting $\alpha = 4/3$ into the given recurrence (3.2) gives

$$S(n) = T(n - 4/3) \leq 3T(n/2 - 4/3) + O(n) = 3S(n/2) + O(n),$$

as desired. We already know that for this recurrence, $S(n) = O\Big(n^{\log_2(3)}\Big)$. To transfer this to $T(n)$, we have

$$T(n) = S(n + 4/3) = O\Big(n^{\log_2(3)}\Big),$$

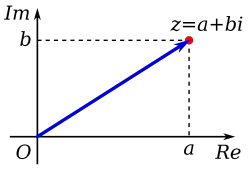
as desired.[4]

3.4 Fourier transforms

[4]In practice, it is fine to just analyze the nicer recurrence $S(n)$, perhaps with some comment such as, "we omit a constant additive factor in the recursive call which does not effect the runtime".

Complex numbers. The discrete Fourier transform is an operation applied to polynomials that takes place in the complex plane $\mathbb{C}$. Here we will present the minimal algebraic facts regarding $\mathbb{C}$. Recall that a complex number $x \in \mathbb{C}$ has the form

$$x = a + \imath b,$$



The complex plane [IW].

where $a, b \in \mathbb{R}$ are real-valued numbers. a is called the "real part" of x and b is called the "complex part" of b. $\imath$ denotes "imaginary i", defined by $\imath^2 = -1$. We have

$$(a + \imath b)(c + \imath d) = ac + (bc + ad)\imath + bd\imath^2 \overset{(a)}{=} ac + (bc + ad)\imath - bd;$$

here (a) applies the identity $\imath^2 = -1$. With this, polynomials

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

are well defined for $x \in \mathbb{C}$. So are functions that can be expressed as convergent power series, such as the exponential function:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Euler's formula gives a geometric interpretation of the exponential function:

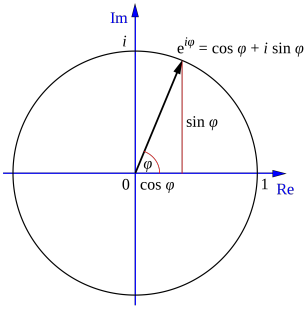
$$e^{\theta\imath} = \cos\theta + \imath \sin\theta.$$

From this we have Euler's identity,

$$e^{\pi\imath} = -1,$$

as well as the identity

$$e^{2\pi\imath} = 1.$$



Euler's formula [GW].

Roots of unity. Now, let $n \in \mathbb{N}$ be fixed. An *nth root of unity* is any (complex) number ω satisfying

$$\omega^n = 1.$$

For example, 1 is an nth root of unity for any n, and -1 is an nth root of unity for any even n. $\imath$ is a 4th root of unity. More generally,

$$\omega_n \overset{\text{def}}{=} e^{2\pi\imath/n}$$

is an nth root of unity, as well as ω_n^j for any $j \in \mathbb{Z}$. In fact, the n distinct powers of ω_n,

$$1 = \omega_n^0, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$$

give all the nth roots of unity.

The discrete Fourier transform. The *(nth) discrete Fourier transform*, denoted F_n, takes as input a polynomial

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1}x^{n-1}$$

represented by its coefficients

$$a_0, \ldots, a_{n-1} \in \mathbb{C},$$

and produces the vector obtained by evaluating $p(x)$ at each of the n roots of unity[5]. Here, identifying p with its coefficients $a \in \mathbb{C}^n$, we write

$$F_n a \stackrel{\text{def}}{=} \Big(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1})\Big).$$

The *conjugate (nth) discrete Fourier transform*, denoted $F_n^* a$, instead lists the values for the powers of ω_n^{-1}:

$$F_n^* a \stackrel{\text{def}}{=} \Big(p(1), p(\omega_n^{-1}), p(\omega_n^{-2}), \ldots, p(\omega_n^{1-n})\Big)$$

Since $\omega_n^{-1} = \omega_n^{n-1}$, F_n^* actually evaluates the same points as $F_n a$, but the order of values is partly reversed.

Later we will show that F_n and F_n^* are nearly inverse to one another, in the following sense. In particular, they can each be used to "undo" the other.

Fact 3.4. *For all $a \in \mathbb{C}^n$, $F_n^* F_n a = na$.*

What is the fastest way to compute F_n? The most direct way to compute the discrete Fourier transform is to just evaluate each $p(\omega_n^i)$ separately, in $O(n^2)$ time overall. The *fast Fourier transform* algorithm applies divide and conquer to give the following improved running time.

Theorem 3.5. *F_n and F_n^* can be computed in $O(n \log n)$ time.*

3.4.1 Application: multiplying polynomials.

Below we will have a more technical discussion that proves both fact 3.4 and theorem 3.5. But to motivate that discussion we first ask: why is F_n useful? We first mention that it is incredibly useful for signal processing and refer to the wikipedia article for more on this application. But as another basic application that requires no

[5]We mention that there are other "continuous" variants for Fourier transforms that we do not discuss.

new definitions, consider the problem of multiplying two degree-$(n-1)$ polynomials $p(x)$ and $q(x)$, represented by coefficients $a, b \in \mathbb{C}^n$ in respectively. Let $r(x) = p(x)q(x)$ be their product, with coefficients $c \in \mathbb{C}^{2n}$. We can compute the coefficients of $r(x)$ explicitly in $O(n^2)$ time, by take the appropriate sums of products of coefficients of p and q. For example, the coefficient c_k corresponding to x^k in r is given by the *convolution*

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}.$$

This is not a complicated formula but computationally it does represent a loop of size k, where k can be as large as $2n$. Using the convolution above to compute $r = pq$, we take $O(n)$ time per coefficient c_k, and $O(n^2)$ time total.

Instead of focusing on the coefficients of r, let us consider its Fourier transform $F_{2n}r$. The values $r(x) = p(x)q(x)$ at the $(2n)$th roots of unity ω_{2n}^k are given by

$$r(\omega_{2n}^k) = p(\omega_{2n}^k)q(\omega_{2n}^k)$$

for each i. In contrast to convolutions, each value takes $O(1)$ time given the values on the RHS. Thus, given $F_{2n}p$ and $F_{2n}q$, computing $F_{2n}r$ is very quick, taking only $O(n)$ time.

Now recall that F_{2n} computes all $2n$ roots of unity, and F_{2n}^* effectively reverses the process and recovers the polynomial (times a factor of n) from the values at the roots of unity. We can compute $r = pq$ much more quickly by (a) taking the Fourier transforms of p and q, (b) multiplying the values together to get the Fourier transform of r, and then using the conjugate Fourier transform (and dividing by $2n$) to get the coefficients of r. By the fast Fourier transform (theorem 3.5), the whole procedure takes only $O(n \log n)$ time!

$$(p, q) \xrightarrow{O(n \log n)} (F_{2n}p, F_{2n}q) \xrightarrow{O(n)} F_{2n}r \xrightarrow{O(n \log n)} F_{2n}^* F_{2n} r = 2nr$$

Corollary 3.6. *Two polynomials of degree $n-1$ can be multiplied in $O(n \log n)$ time.*

We also point out that the fastest integer multiplication algorithms are based on the Fourier transform. Indeed, multiplying integers is not so different from multiplying polynomials – think of the bits of an integer in binary notation as the n coefficients of a degree-n polynomial.

3.4.2 The fast Fourier transform

We defer the proof of fact 3.4 until the end, as it is a purely mathematical fact. Let us first discuss the divide-and-conquer algorithms for F_n and F_n^* which is the main pedagogical point. As noted above, F_n^* is a rearrangement of the coordinates produced by F_n, so it suffices to only present an algorithm for F_n.

Let $p(x)$ be a degree $(n-1)$-degree with coefficients $a_0, \ldots, a_{n-1} \in \mathbb{C}$:

$$p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}.$$

For ease of discussion let us assume that n is a power of 2. Our goal is to compute the vector of values

$$F_n a \stackrel{\text{def}}{=} \Big(p(1),\, p(\omega_n),\, ,\, p(\omega_n^2),\, p(\omega_n^2)\Big)$$

To avoid ambiguity we first give a recursive spec for our algorithm. Given $n \in \mathbb{N}$ and a vector $a \in \mathbb{C}^n$ (indexed as $a_0, a_1, \ldots, a_{n-1} \in \mathbb{C}$ by convention), we define

> `Fourier(`$a_0, \ldots, a_{n-1}$`)` $=$ the nth discrete Fourier transform
>
> $$F_n a = \Big(p(1), p(\omega_n), p(\omega_n^2), \ldots, p(\omega_n^{n-1})\Big)$$
>
> where $p(x)$ is the polynomial $p(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$.

The divide and conquer argument exploits the following identity for ω_n (for all $n \in \mathbb{N}$):

$$\omega_{2n} = \omega_n^2. \tag{3.4}$$

Indeed, recalling that $\omega_n \stackrel{\text{def}}{=} e^{2\pi\imath/n}$, we have

$$\omega_{2n}^2 = e^{(2\pi\imath/2n)^2} = e^{2\pi\imath/n} = \omega_n.$$

Now, consider $(F_{2n}a)_k$ for fixed $n \in \mathbb{N}$ and $a \in \mathbb{C}^{2n}$. We have

$$\begin{aligned}(F_{2n}a)_k &\stackrel{\text{(a)}}{=} a_0 + a_1\omega_{2n}^k + a_2(\omega_{2n}^k)^2 + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{2n-1} \\ &\stackrel{\text{(b)}}{=} \Big(a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{(2n-2)}\Big) \\ &\quad + \Big(a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{(2n-1)}\Big).\end{aligned} \tag{3.5}$$

Here (a) is by definition of F_{2n}, and (b) groups the sum into one sum for the even-index coordinates, and another sum for the odd-index coordinates.

Consider the first group with the even-index coefficients. All the powers of ω_{2n} are even, so by applying the identity (3.4), we have

$$a_0 + a_2(\omega_{2n}^k)^2 + \cdots + a_{2n-2}(\omega_{2n}^k)^{(2n-2)} = a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1}.$$

Now the RHS can be interpreted as evaluating a degree $n-1$ polynomial with coefficients $a_0, a_2, \ldots, a_{2n-2}$ at ω_n^k. More specifically, if we let $a_{\text{even}} = (a_0, a_2, \ldots, a_{2n-2})$, then we have

$$a_0 + a_2\omega_n^k + \cdots + a_{2n-2}(\omega_n^k)^{n-1} = F_{a_{\text{even}}}$$

– the nth Fourier transform of the even-index coefficients!

We can do something similar for the odd coefficients. Let $a_{\text{odd}} = (a_1, a_3, \ldots, a_{n-1})$ be the vector of odd coefficients, and consider the second sum in (3.5). We have

$$\begin{aligned}
&a_1\omega_{2n}^k + a_3(\omega_{2n}^k)^3 + \cdots + a_{2n-1}(\omega_{2n}^k)^{(2n-1)} \\
&= \omega_{2n}^k\Big(a_1 + a_3(\omega_n^k)^2 + a_5(\omega_n^k)^4 + \cdots + a_{2n-2}(\omega_n^k)^{n-1}\Big) \\
&= \omega_{2n}^k\Big(a_1 + a_3(\omega_n^k)^2 + a_5(\omega_n^k)^4 + \cdots + a_{2n-2}(\omega_n^k)^{n-1}\Big) \\
&= \omega_{2n}^k(F_n a_{\text{odd}}).
\end{aligned}$$

Plugging all this back into (3.5), we have the clean identity

$$(F_{2n}a)_k = (F_n a_{\text{even}})_k + \omega_{2n}^k(F_n a_{\text{odd}})_k$$

for $k = 0, \ldots, n-1$. This equation tells us that computing $F_{2n}a$ is largely reduced to computing $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$. More specifically, given $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$, we need only $O(2n)$ time to compute $F_{2n}a$. Meanwhile $F_n a_{\text{even}}$ and $F_n a_{\text{odd}}$ can be computed recursively, where each subproblem has half the size / dimension.

In fig. 3.7 on page 68, we give pseudocode for the recursive algorithm `Fourier(`$a_0, \ldots, a_{n-1}$`)`, based on the divide-and-conquer approach outlined above. The code in fig. 3.7 is modified slightly for mathematics above to remove the assumption that n is even. (The argument for odd n is the exact same, except there is one more even-index coefficient than odd-index coefficient.)

We now analyze the running time of `Fourier(`$a_0, \ldots, a_{n-1}$`)`. As annotated in the pseudocode, `Fourier(`$a_0, \ldots, a_{n-1}$`)` consists of 2 recursive calls with half the number of coefficients, plus $O(n)$ work. Thus the running time obeys the recurrence

$$\begin{aligned}
T(1) &= O(1) \\
T(n) &= 2T(n/2) + O(n).
\end{aligned}$$

`Fourier`$(a_0, \dots, a_{n-1})$

/* *We assume for simplicity that n is even.* */

1. If $n = 0$ then return the empty vector.
2. If $n = 1$ then return a_0.
3. Let $a_{\text{even}} = (a_0, a_2, a_4, \dots)$ be the vector of $\lceil n/2 \rceil$ even-index coefficients and let $a_{\text{odd}} = (a_1, a_3, a_5, \dots)$ be the vector of $\lfloor n/2 \rfloor$ odd-index coefficients. // $O(n)$
4. Let $(y_0, \dots, y_{\lceil n/2 \rceil - 1}) =$ `Fourier`(a_{even}) // $T(\lceil n/2 \rceil)$
5. Let $(z_0, \dots, z_{\lfloor n/2 \rfloor - 1}) =$ `Fourier`(a_{odd}). // $T(\lfloor n/2 \rfloor)$
6. For $k = 0, 1, \dots, n-1$, let $x_k = y_k + \omega_n^k z_k$ // $O(n)$
7. Return $(x_0, \dots, x_{n-1})$.

Figure 3.7: The fast Fourier transform.

This is the same recurrence as in `merge-sort` (cf. section 1.2) and is solved by

$$T(n) = O(n \log n).$$

This completes the proof of theorem 3.5.

3.4.3 Inverting F_n and F_n^*

Lastly we prove fact 3.4, which described a near-inverse relationship between F_n and F_n^*. Fact 3.4 is just as critical to applications as the $O(n \log n)$ algorithm above, and this discussion was deferred to the end only for pedagogical reasons. (In particular the discussion is purely mathematical.) We first require the following identity.

Lemma 3.7. *For any nth root of unity ω except for $\omega = 1$, we have*

$$1 + \omega + \omega^2 + \cdots + \omega^{n-1} = 0.$$

For $\omega = 1$, the LHS is n.

Proof. We have

$$(1 - \omega)\left(1 + \omega + \omega^2 + \cdots + \omega^{n-1}\right) = 1 - \omega^n = 0,$$

so either $\omega = 1$ or $1 + \omega + \cdots + \omega^{n-1} = 0$. □

Now we prove fact 3.4. We restate the claim below for the reader's convenience.

Fact 3.4. *For all $a \in \mathbb{C}^n$, $F_n^* F_n a = na$.*

Proof. Take any $a \in \mathbb{C}^n$; we want to show that $F_n^* F_n a = na$. Indeed, for each coordinate $h \in \{0, \dots, n-1\}$, we have

$$(F_n^* F_n a)_h \overset{\text{(a)}}{=} \sum_{j=0}^{n-1} (F_n a)_j (\omega_n^{-h})^j \overset{\text{(b)}}{=} \sum_{j=0}^{n-1} \left(\sum_{k=0}^{n-1} a_k (\omega_n^j)^k \right) (\omega_n^{-h})^j$$
$$\overset{\text{(c)}}{=} \sum_{k=0}^{n-1} a_k \sum_{j=0}^{n-1} (\omega_n^{k-h})^j \overset{\text{(d)}}{=} n a_h.$$

Here (a) and (b) expand out the definitions of F_n^* and F_n. (c) interchanges sums. (d) is by lemma 3.7: unless $k - h = 0$ (or some other multiple of n), we have $\sum_{j=0}^{n-1} (\omega_n^{k-h})^j = 0$. □

One implication of fact 3.4 is as follows.

Corollary 3.8. *A polynomial $p(x)$ of degree $n-1$ is uniquely identified by its value at the n roots of unity $p(1), p(\omega_n), \dots, p(\omega_n^{n-1})$.*

The above is a special case of the fundamental theorem of algebra.

3.5 Matrix multiplication

Let A and B be two $n \times n$ matrices. We can compute the product AB in $O(n^3)$ based on the formula

$$(AB)_{ij} = \sum_k A_{ik} B_{kj}.$$

In 1969, Strassen [Str69] showed that one compute AB faster by leveraging divide-and-conquer, which was very surprising. Write

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \text{ and } B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

where $A_{11}, \dots, B_{2,2} \in \mathbb{R}^{n/2}$ are all square matrices with half the dimension. Recall that

$$AB = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

This divides AB into 8 matrix multiplications of smaller matrices. Strassen showed that in fact 7 multiplications suffice. We describe the scheme momentarily. First we observe that this gives a recursion of the form

$$T(n) = 7T(n/2) + O(n).$$

A quick sketch of a recursion tree reveals:

1. The kth level has 7^k subproblems each with $O\Big(n/2^k\Big)$ work.
2. There are at most $\lceil \log n \rceil$ levels.

Thus the total running time is

$$O\left(\sum_{i=1}^{\log(n)} \left(\frac{7}{2}\right)^k n\right) = O\Big(n^{\log_2(7)}\Big) \approx O\Big(n^{2.8074}\Big).$$

We will describe Strassen's algorithm for the sake of completeness but by no means do we claim it is obvious, even in hindsight. Consider the following 7 products.

$$(A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{22}B_{11} + A_{11}B_{22} + A_{11}B_{22} \tag{3.6}$$
$$(A_{11} - A_{21})(B_{11} - B_{12}) = A_{11}B_{11} - A_{21}B_{11} - A_{11}B_{12} + A_{21}B_{12} \tag{3.7}$$
$$(A_{12} - A_{21})(B_{21} + B_{22}) = A_{12}B_{12} - A_{21}B_{21} - A_{21}B_{22} + A_{12}B_{22} \tag{3.8}$$
$$A_{11}(B_{12} - B_{22}) = A_{11}(B_{12} - B_{22}) \tag{3.9}$$
$$(A_{11} + A_{12})B_{22} = A_{11}B_{22} + A_{12}B_{22} \tag{3.10}$$
$$(A_{11} + A_{22})(B_{11} + B_{22}) = A_{11}B_{11} + A_{11}B_{11} + A_{22}B_{11} + A_{22}B_{22} \tag{3.11}$$
$$(A_{11} - A_{21})(B_{11} + B_{12}) = A_{11}B_{11} - A_{21}B_{11} + A_{11}B_{12} - A_{21}B_{12} \tag{3.12}$$

Then

$$\begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$
$$= \begin{pmatrix} (3.6) + (3.7) - (3.10) + (3.8) & (3.9) + (3.10) \\ (3.11) + (3.7) & (3.9) + (3.6) - (3.11) - (3.12) \end{pmatrix}$$

Even more clever combinations and strategies exist which has driven the exponent down further. Currently, the best running time is $O(n^\omega)$ for $\omega \approx 2.37$ [AW20]. It is conjectured that ω will eventually be driven down to 2 (a natural lower bound). We should point out that the theoretically best algorithms are so complicated that not necessarily preferred practice. (The hidden constant strikes back!)

3.6 Additional notes and references

Spring 2022 Lecture 11. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

Spring 2022 Lecture 12. Click on the links below for the following files:

- Handwritten notes prepared before the lecture.
- Handwritten notes annotated during the presentation.
- Recorded video lecture.

3.7 Exercises

Many exercises can be found in [KT06, Chapter 5], [DPV08, Chapter 2], and [Eri19, Chapter 1]. When designing and analyzing divide-and-conquer algorithms, make sure to address the following (unless specified otherwise):

1. A recursive spec for your algorithm.
2. A recursive implementation of your recursive spec.
3. An explanation of how to use the recursive algorithm to solve the original problem.
4. An analysis of the running time.
5. A proof by induction, where the induction hypothesis is based on the recursive spec.

Exercise 3.1. Let $A[1..n]$ be an array of elements. The *frequency* of an element is the number of times it appears in $A[1..n]$. An element is a *majority element* if it has frequency at least $n/2$.

If the elements are comparable then we can identify the majority element by sorting all the elements and scanning the sorted list. Here we do *not* assume that the elements are pairwise comparable; we can only test if two elements $A[i]$ and $A[j]$ are equal.

1. Write a recursive spec for the majority element problem.
2. Design a divide-and-conquer algorithm (implementing your recursive spec) for the majority element problem that divides the input in half, and runs in $O(n \log n)$.[6]

[6]There is also a well-known $O(n)$ time algorithm which we are not asking for.

3. Prove your algorithm is correct by induction, where the recursive spec is your induction hypothesis.

4. Analyze the running time of your algorithm.

Exercise 3.2. Here we assume the same model as exercise 3.1 – an array $A[1..n]$ of incomparable elements[7] – and generalize the notion of a majority element. For $k > 1$, an element is a $(1/k)$-*heavy-hitter* if it has frequency at least n/k. (e.g., majority elements are $(1/2)$-heavy hitters.) Note that there are at most k $(1/k)$-heavy-hitters in $A[1..n]$.

1. (2.5 pt.) Give a recursive spec for the $(1/k)$-heavy hitter problem.

2. (2.5 pt.) Give a divide-and-conquer algorithm for the $(1/k)$-heavy hitter problem implementing your recursive spec (the faster the better). (This algorithm should be similar to your majority element algorithm.)

3. (2.5 pt.) Analyze the running time of your algorithm.

4. (2.5 pt.) Prove your algorithm is correct by induction, where the recursive spec should act as your induction hypothesis.

Your solutions should generalize that of finding the majority elements. That is, plugging in $k = 2$ should recover the same results as for the majority problem.[8]

Exercise 3.3. Recall that given a set P of n points in $\mathbb{R}^2$, we can compute the minimum distance between any pair of points in P in $O(n \log n)$ time. Consider now the problem of computing the second smallest distance between any pair of points in P. Design and analyze an algorithm for this problem.

Exercise 3.4. Let X and Y be two sets of integers. We define the *unique sums* of X and Y as the set of integers of the form

$$z = x + y$$

where $x \in X$, $y \in Y$, and the choice of $(x, y) \in X \times Y$ is unique.[9] You may assume that all the integers in X are distinct (amongst themselves), and that all the integers in Y are distinct (amongst themselves).[10]

[7] In addition to disallowing sorting, no hash tables are allowed as well.

[8] Your algorithm should be faster than $O(n^2)$ (which is the trivial running time) for, e.g., $k = n^{1/3}$.

[9] That is, for fixed z, there is only one choice of $x \in X$ and one choice of $y \in Y$ such that $z = x + y$.

[10] For example, for $X = \{0, 1\}$ and $Y = \{0, 1\}$, the unique sums are $\{0, 2\}$. 1 can be obtained by $0 + 1$ or $1 + 0$ so it is not a unique sum.

1. (2 pt.) Suppose X and Y each have n integers. Design and analyze a $O(n^2 \log(n))$ time algorithm to compute the unique sums of X and Y.

2. (6 pt.) Suppose all the integers in X are Y are between 0 and M for a fixed value $M \in \mathbb{N}$. Design and analyze a $O(M \log M)$ time algorithm to compute the unique sums of X and Y.[11]

3. (2 pt.) Suppose we now that we had k sets $X_1, \ldots, X_k$, each consisting of integers between 1 and M. (You may again assume no duplicates within each X_i.) A unique sum of $X_1, \ldots, X_k$ is defined as an integer of the form

$$x_1 + \cdots + x_k,$$

where $x_1 \in X_1$, $x_2 \in X_2$, $\ldots$, $x_k \in X_k$, and the choice of $(x_1, \ldots, x_k) \in X_1 \times \cdots \times X_k$ is unique. Design and analyze an algorithm that computes the unique sums of $X_1, \ldots, X_k$ in $O(kM \log(Mk) \log(k))$ time. You may assume for simplicity that k is a power of 2.[12]

Hint: For part 2, try "changing the input" to an algorithm from section 3.4 (that is not the FFT, at least directly).

Exercise 3.5. Consider the following generic recursion.

$$T(n) = \alpha T(\beta n) + n^\gamma$$
$$T(1) = \delta$$

where $\alpha, \beta, \gamma, \delta > 0$ are fixed constants. Prove each of the following using the recursion tree method.

1. For any $\beta < 1$, $T(n)$ is at most a polynomial in n. (Even if, say, $\alpha = 2^{260}$. Here $\alpha, \beta, \gamma, \delta$) may appear in the exponent of the polynomial.) For simplicity, you may assume that α is an integer.

2. Show, using the recursion tree method, that if $\alpha\beta^\gamma < 1$, then $T(n) \leq O(n^\gamma)$.

3. Show, using the recursion tree method, that if $\alpha\beta^\gamma = 1$, and $\beta < 1$, $T(n) \leq O(n^\gamma \log(n))$.

[11] It might be helpful to work through very simple examples such as $X = Y = \{1\}$ and $X = Y = \{0, 1\}$.

[12] It might help to work through $k = 4$ to build your intuition.