Divide & Conquer

(high-level comment)

*(mostly; we also had a little divide+conquer extend. merge-sort)

"only two algorithms"*

① identify subproblems

② change the input

① identify subproblems

code for induction / recursion

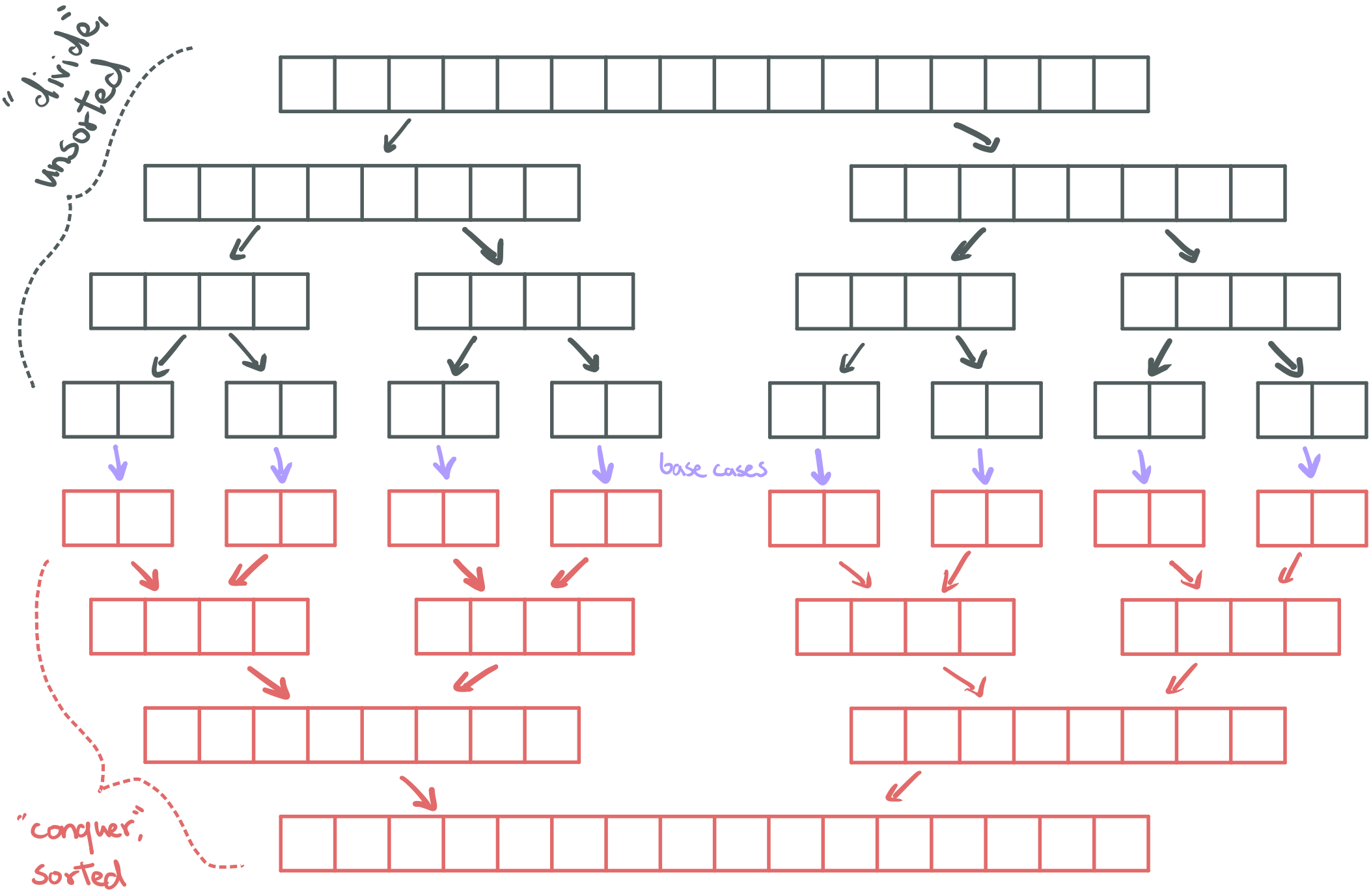
Divide + Conquer

`merge-sort`$(A[1..n])$ = an array consisting of the elements of A sorted in increasing order.

Dynamic Programming

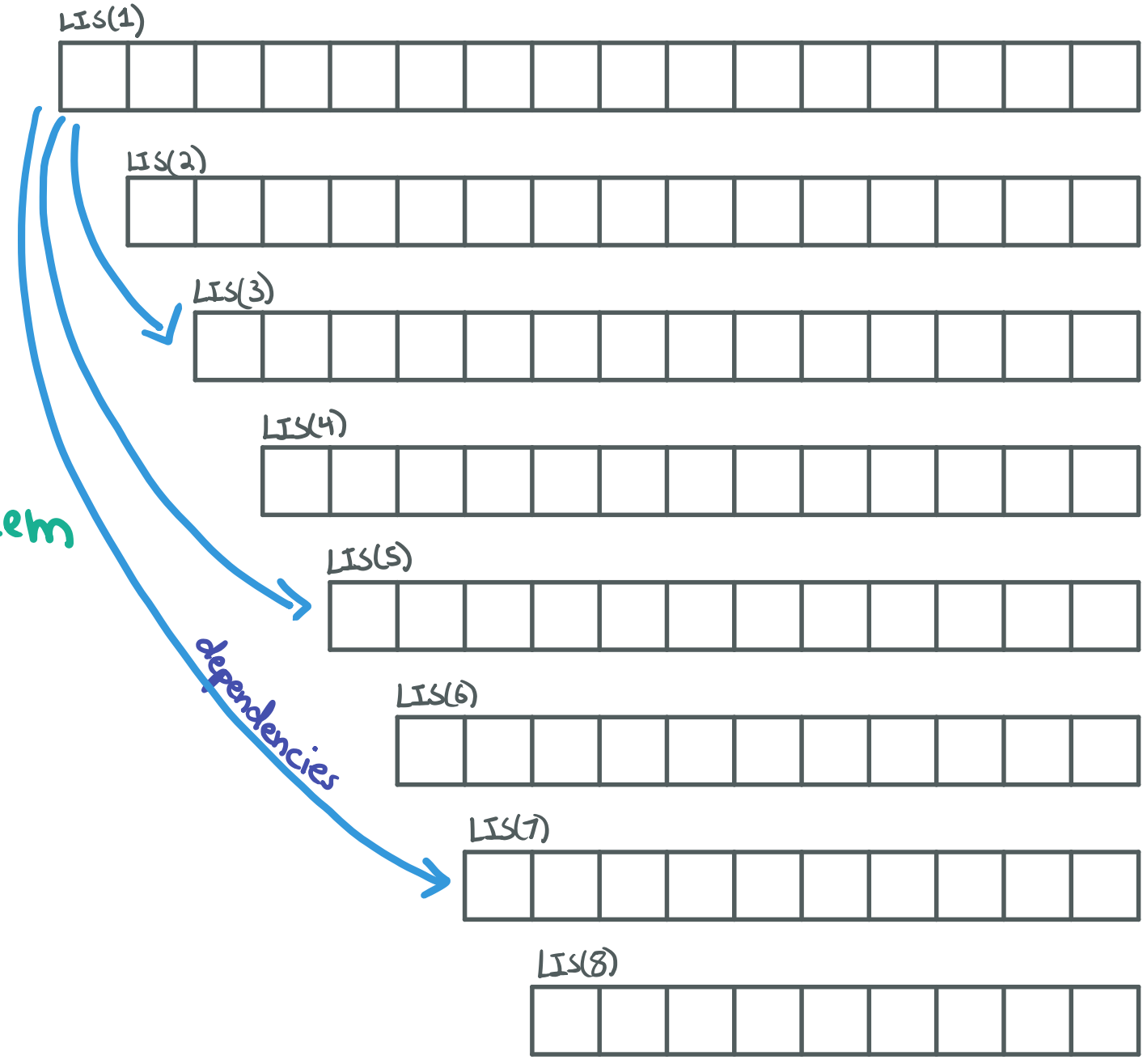
`LIS`(i) = the length of the longest increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.

merge-sort($A[1..n]$) = an array consisting of the elements of A sorted in increasing order.



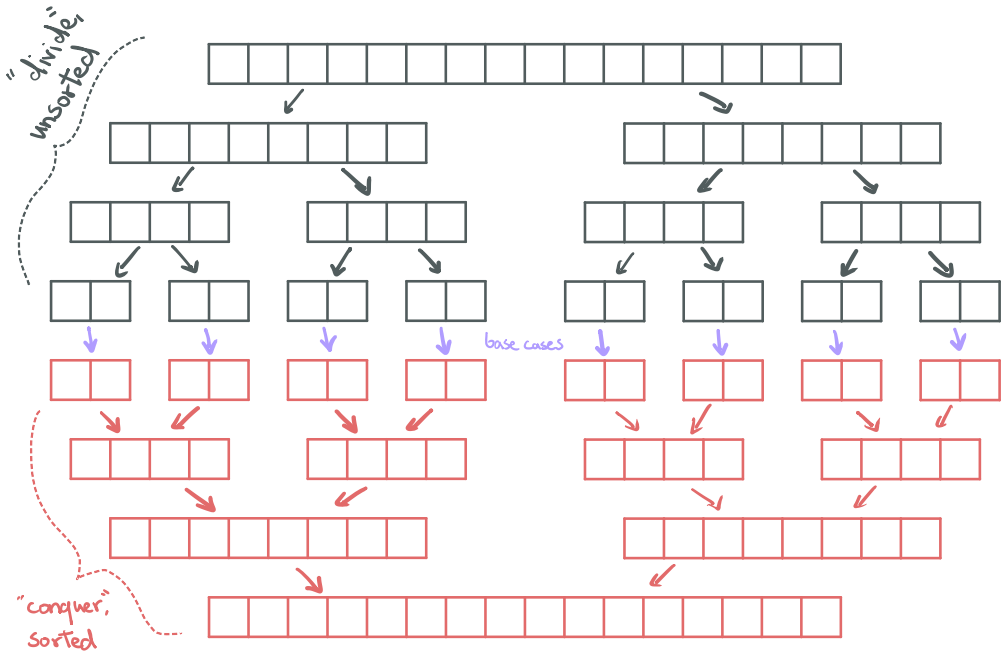
$\texttt{LIS}(i)$ = the length of the longest increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.

- overlapping subproblems
- many subproblem dependencies



Divide + Conquer

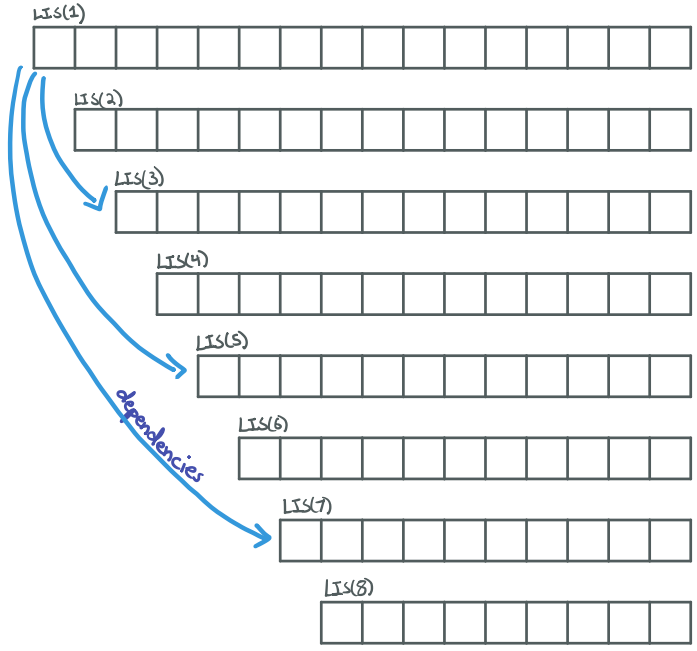
`merge-sort`$(A[1..n])$ = an array consisting of the elements of A sorted in increasing order.



- fewer, mostly disjoint subproblems
- correctness proof relatively easy
- very economical
- nearly linear time

Dynamic Programming

`LIS`(i) = the length of the longest increasing subsequence of $A[i..n]$ where the subsequence starts with $A[i]$.



- many highly overlapping subproblems
- correctness more subtle
- simulates brute force / exhaustive search
- polynomial time

Divide + Conquer

① identify subproblems

② change the input

I. Integer multiplication

II. Selection

III. Minimum Euclidean distance

IV. Fourier Transforms

} next time

Multiplying two n-bit integers

Karatsuba

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

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

123×456

$$
\begin{array}{r}
123 \\
\times\ 456 \\
\hline
738 \\
615 \\
492 \\
\hline
56088
\end{array}
$$

Better?

Multiplying two n-bit integers

Karatsuba

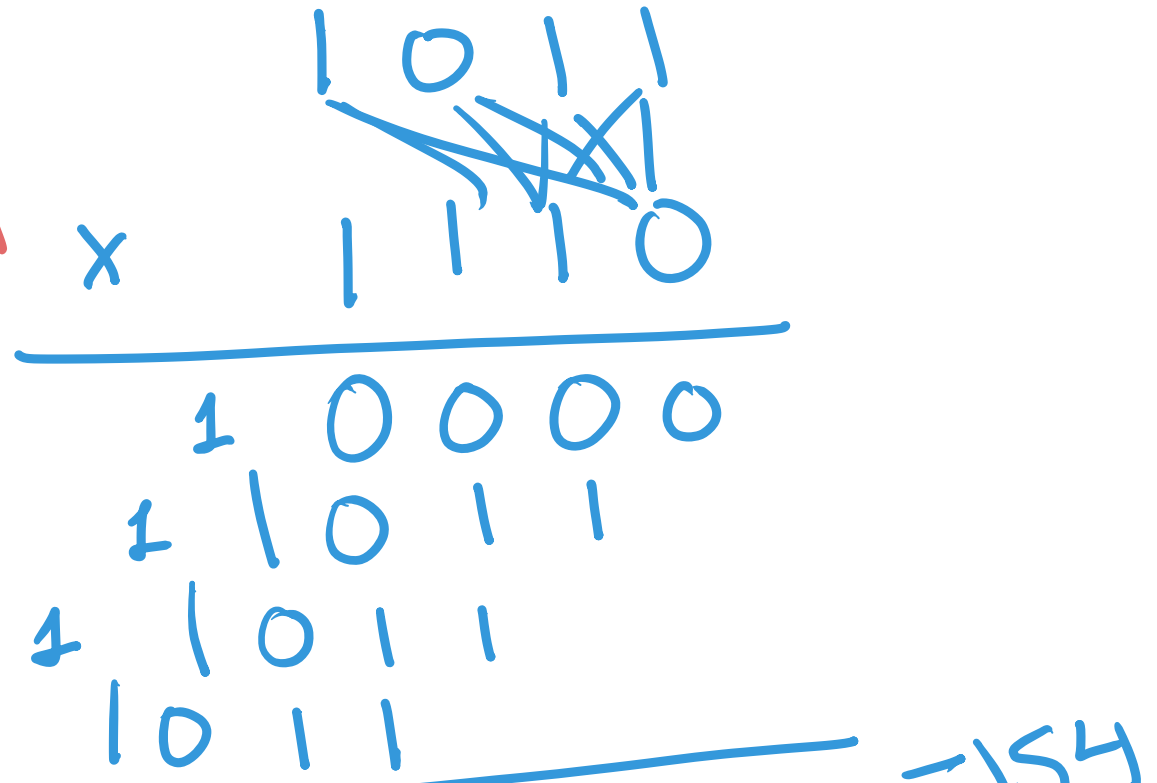
Input: $x, y \in \{0,1\}^n$

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

$O(n^2)$

Karatsuba

Better?



1 0 0 1 1 0 1 0 = ?

Divide bits in half

$$x = x_1 2^{n/2} + x_2$$

$$x_1, x_2, y_1, y_2 \in \{0,1\}^{n/2}$$

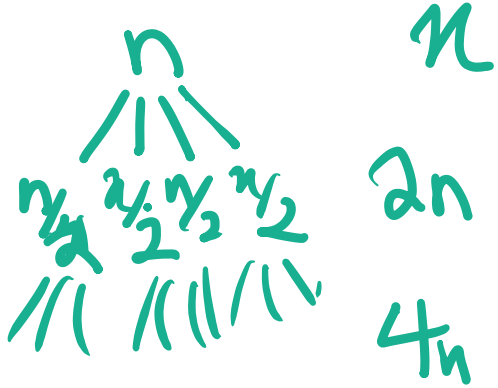
$$y = y_1 2^{n/2} + y_2$$

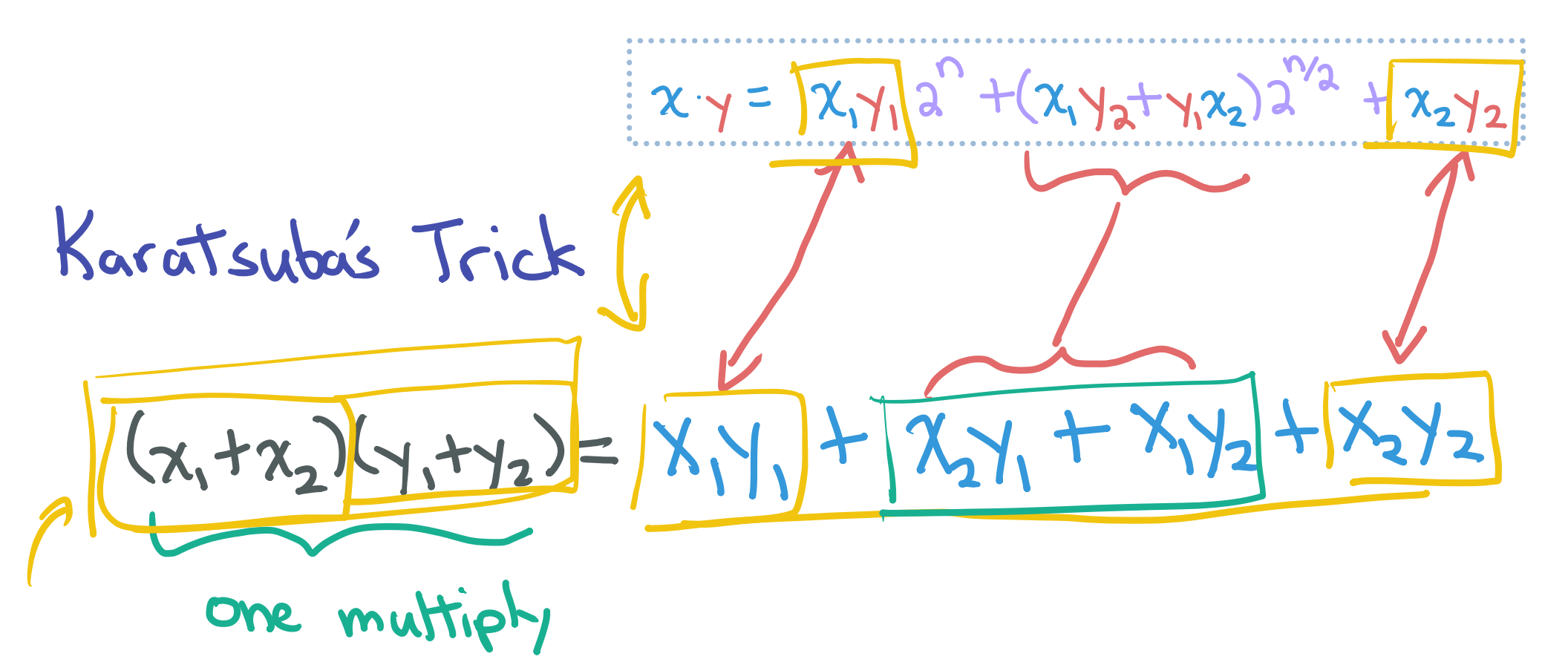
$$xy = (x_1 2^{n/2} + x_2) \cdot (y_1 2^{n/2} + y_2)$$

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

which is 4 multiplies of $\frac{n}{2}$ bits

$$T(n) = 4T(n/2) + O(n)$$





$$x_2y_1 + x_1y_2 = (x_1+x_2)(y_1+y_2) - x_1y_1 - x_2y_2$$

Steps:

1. Recursive spec
2. Recursive implementation
3. Relate recursive spec to problem
   (here its obvious)
4. Running time
5. Proof of correctness (by induction)

1. Recursive spec

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

2. Recursive implementation

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

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

1. If $n = 0$ then return 0.
2. If $n = 1$ then return $x[1] * y[1]$.

} base cases

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). // $3T((n+2)/2)$.

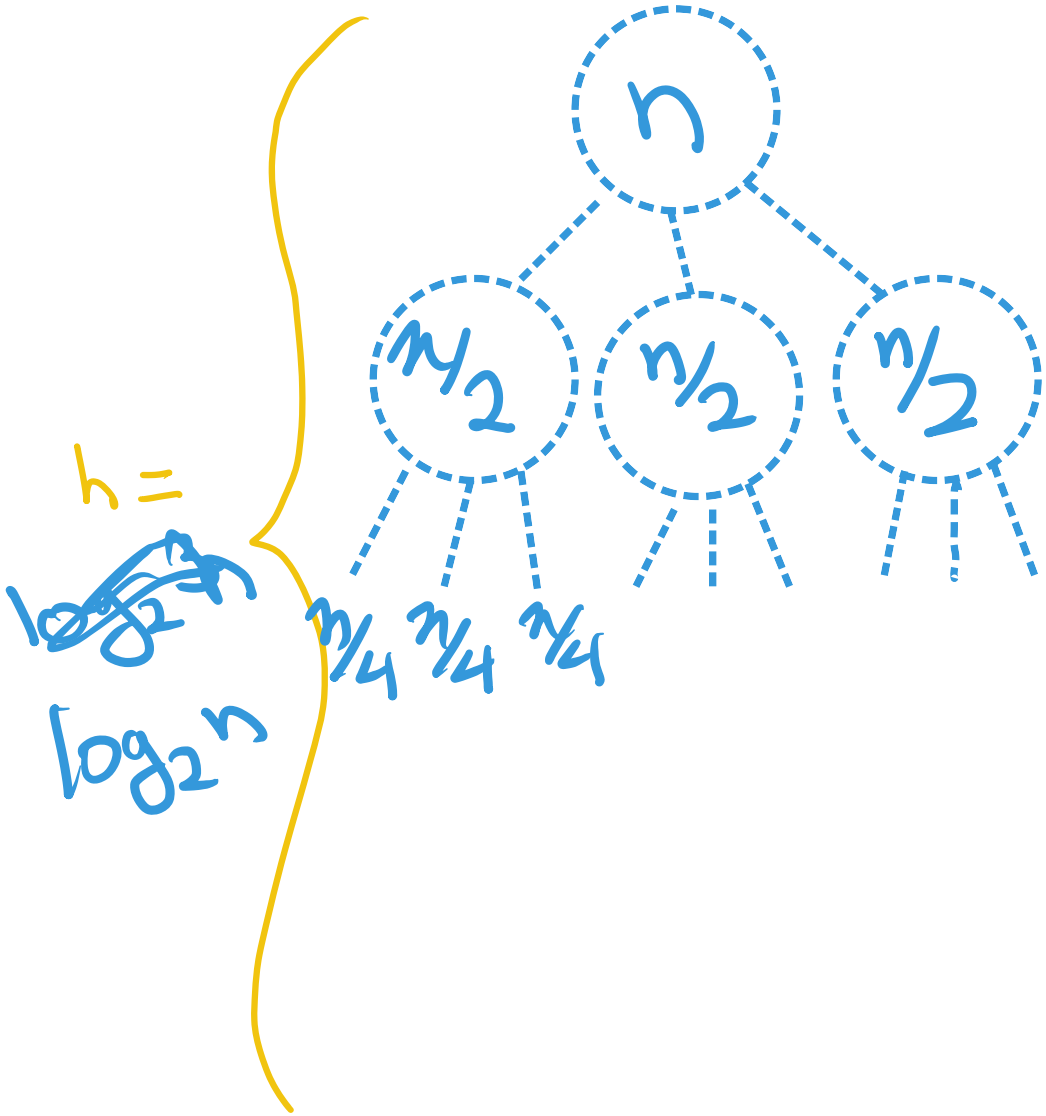
/ Note that multiplying by 2^n is just a bit-shift and takes $O(n)$ time. */*

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

4. Running time

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

$$= 3T\left(\frac{n}{2}+1\right) + O(n)$$



n

3n/2

9n/4

$\left(\frac{3}{2}\right)^i n$ ≤

`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 $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$

4. Running time

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

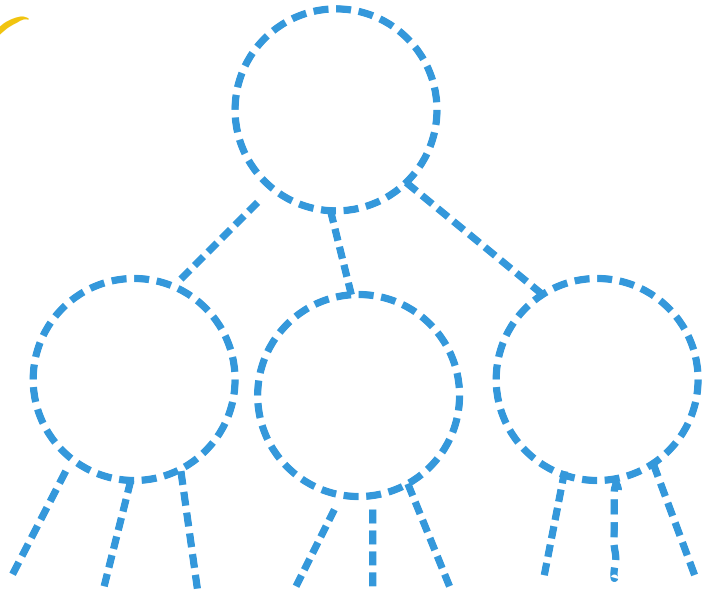
$$= 3T(\tfrac{n}{2}+1) + O(n)$$

`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 $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$



$h =$

nicer:

$$T(n) = 3T(\tfrac{n}{2}+1) + O(n)$$

$$S(n) = 3S(\tfrac{n}{2}) + O(n)$$

$S(n) = T(n+d)$ for some d TBD

$$\boxed{S(n) = T(n+\tfrac{1}{3})}$$

Running time

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

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

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

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

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

$$O\left(n^{1.585}\right)$$

$$\log_2 (3/2) = \log_2 (3) - \log_2 (2)$$

$$= \log_2 (3) - 1$$

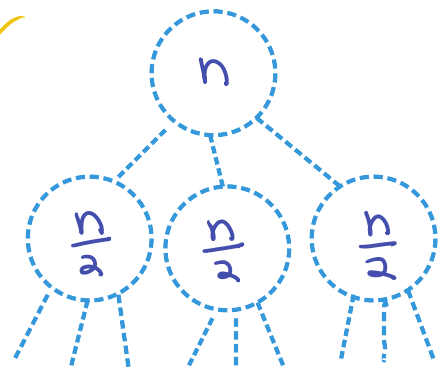
4. Running time

`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 $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$



$h = \log_2 n$

n

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

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

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

Selection

Input Numbers $A[1 \sim n]$ (not sorted)

index k

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

Easy: sort A, find kth

$\Rightarrow O(n \log n)$

Better?

I. Recursive Spec

$\texttt{select}(k, A[1..n]) =$ the kth smallest element out of $A[1..n]$.

First idea: "divide along pivot"

Thought experiment: suppose we have

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

in $O(n)$.

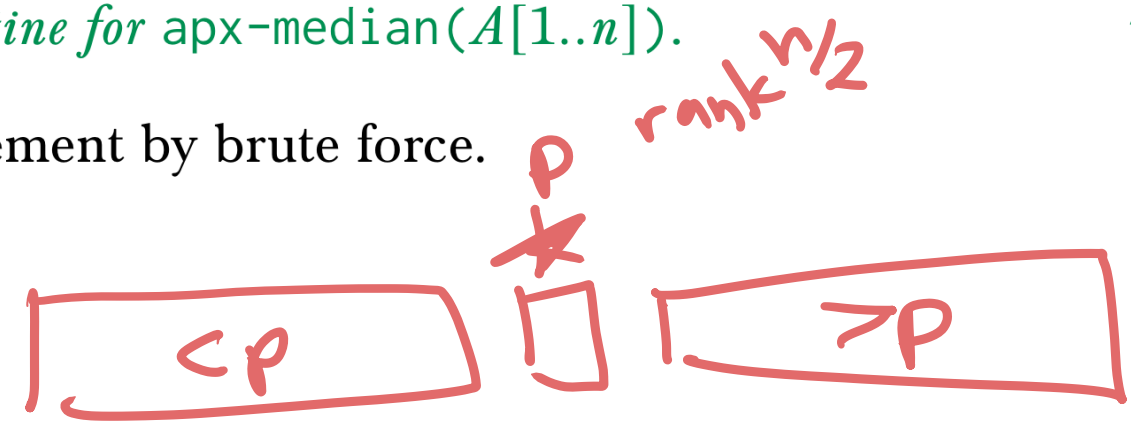
we use median as pivot.

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

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

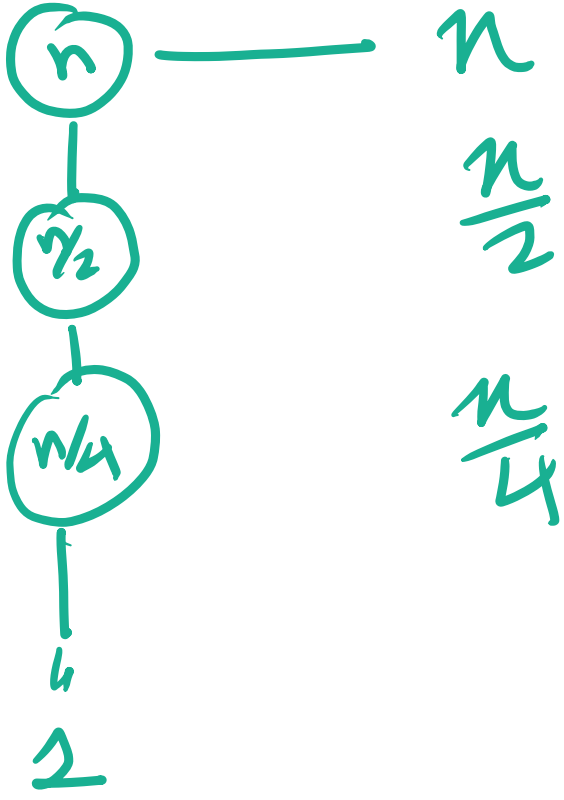
/* *We assume a linear time subroutine for* `apx-median(A[1..n])`. */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median(A)`.
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k < \lceil n/2 \rceil$
   A. Let $B[1..\lceil n/2 \rceil - 1]$ be a list/array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
   B. Return `select(k, B)`
5. If $k > \lceil n/2 \rceil$
   A. Let $B[1..n - \lceil n/2 \rceil]$ be a list/array of the $n - \lceil n/2 \rceil$ elements larger than p.
   B. Return `select(k − ⌈n/2⌉, B)`



Running time

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



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

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

/ We assume a linear time subroutine for* `apx-median`$(A[1..n])$. **/*

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A). $O(n)$
3. If $k = \lceil n/2 \rceil$ then return p. $O(1)$
4. If $k > \lceil n/2 \rceil$ $O(1)$
   A. Let $B[1..\lceil n/2 \rceil - 1]$ be a list/array of the $\lceil n/2 \rceil - 1$ elements smaller than p. $O(n)$
   B. Return `select`(k, B) $T(n/2)$
5. If $k < \lceil n/2 \rceil$
   A. Let $B[1..n - \lceil n/2 \rceil]$ be a list/array of the $n - \lceil n/2 \rceil$ elements larger than p. $O(n)$
   B. Return `select`$(k - \lceil n/2 \rceil, B)$ $T(n/2)$

`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.

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

/* *Here we implement* select($k, A[1..n]$) *assuming a linear time subroutine for* median($A[1..n]$). */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let p = apx-median(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 a list/array of the $\lceil \ell \rceil - 1$ elements smaller than p.
6. If $k > \ell$ then return select($k - \ell, B$), where $B[1..n - \ell]$ is a list/array of the $n - \ell$ elements larger than p.

Running time

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

$T(n) = T(\frac{9n}{10}) + O(n)$

n

$\frac{9}{10} n$

$(\frac{9}{10})^2 n$

$\vdots$

$(\frac{9}{10})^i n = O(n)$

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

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

/* *Here we implement* `select(`k`,`$A[1..n]``)` *assuming a linear time subroutine for* `median(`$A[1..n]$`)`. */

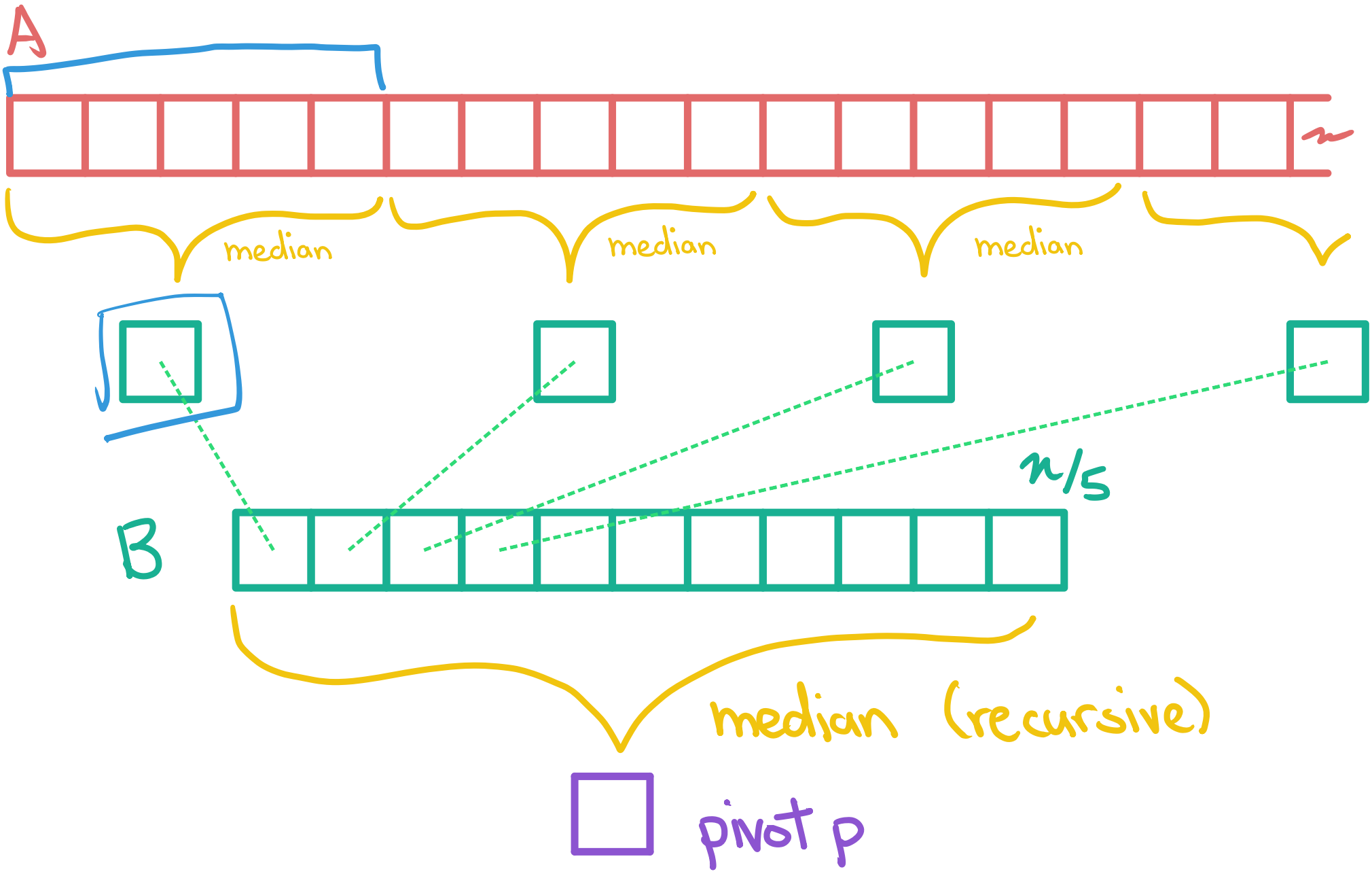
1. If $n \leq 10$ then find the kth element by brute force.
2. Let p = `apx-median(`A`)`.
3. Compare p to each element in $A[1..n]$ to identify its rank ℓ.
4. If $k = \ell$ then return p.
5. If $k < \ell$ then return `select(`k`,`B`)`, where $B[1..\ell - 1]$ is a list/array of the $\lceil \ell \rceil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

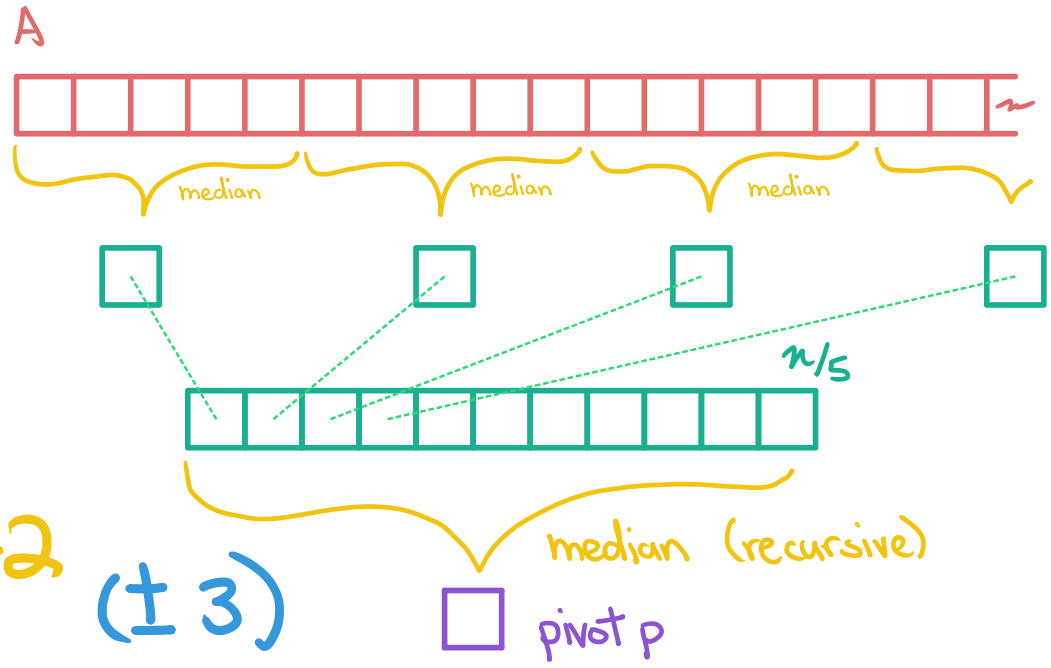
Blum, Floyd, Pratt, Rivest, Tarjan



claim

M-of-M pivot p has

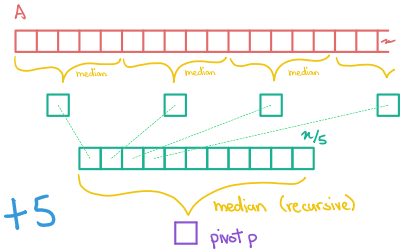
$$\frac{3n}{10} - 2 \leq \text{rank}(p) \leq \frac{7n}{10} + 2 \quad (\pm 3)$$



claim

M-of-M pivot p has

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



B sorted →

each set of 5 sorted ↓

B

n/5

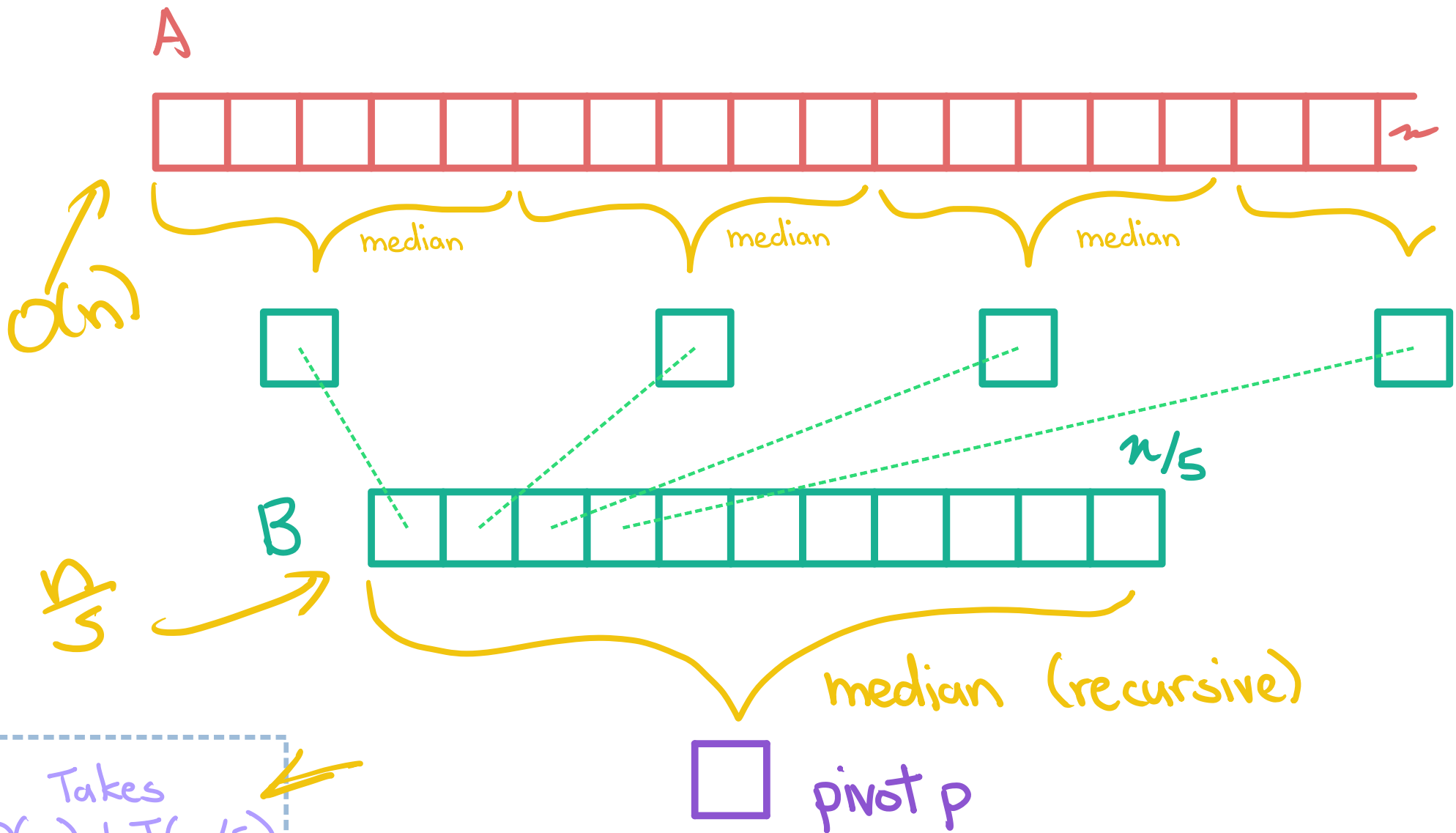
= def. smaller

= def. bigger

$\rightarrow \left(\frac{n}{5}\right) \times \left(\frac{1}{2}\right) \times 3 = \frac{3n}{10}$

"Median of Medians"

Blum, Floyd, Pratt, Rivest, Tarjan



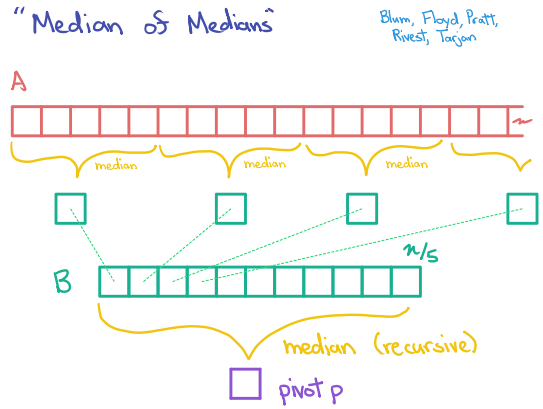
Takes $O(n) + T(n/5)$ time

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

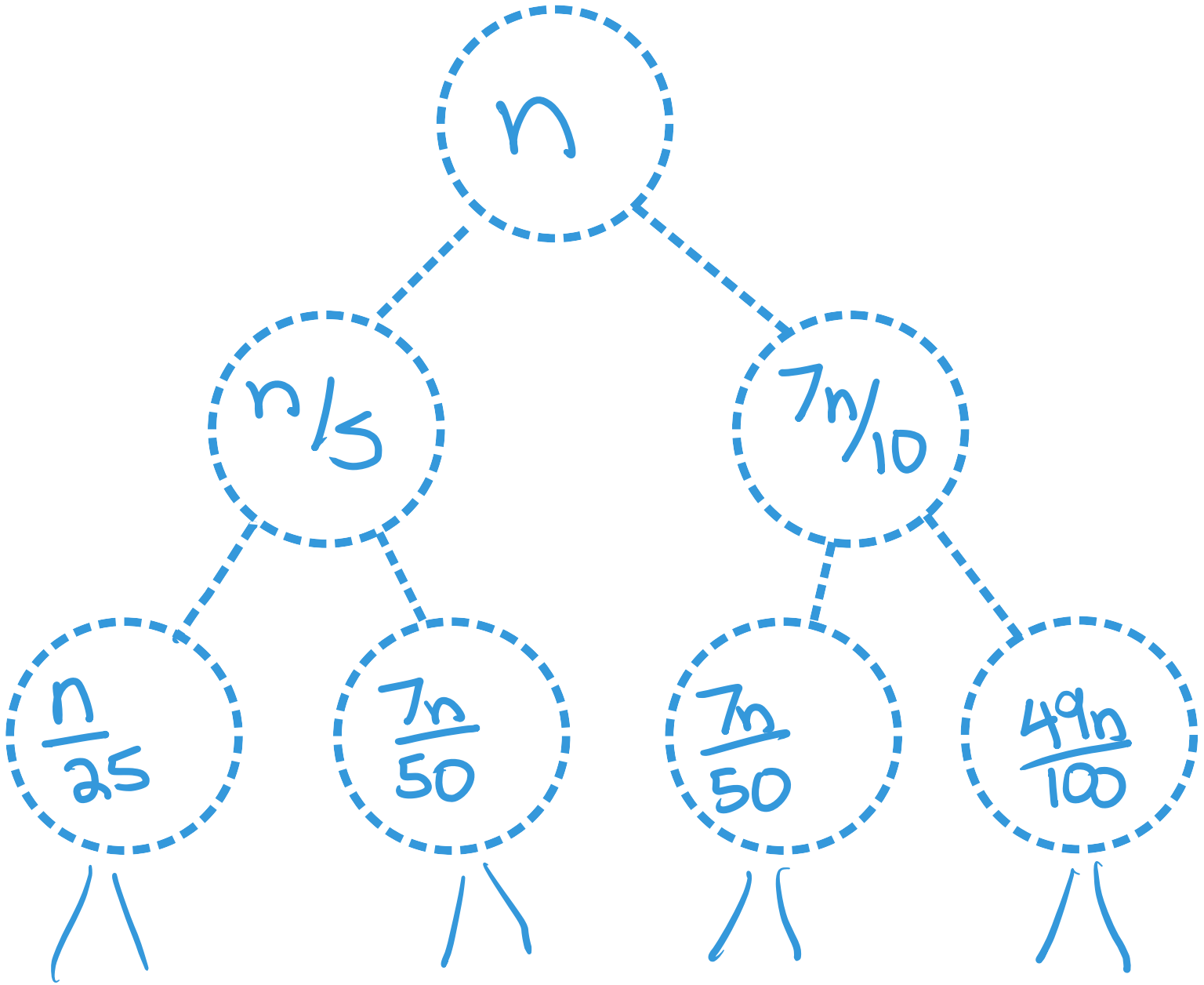
/* `select(k, A[1..n])` = *the ith smallest element in* $A[1..n]$. *We assume the elements in* A *are distinct for simplicity. (Otherwise, break ties consistently.)* */

/* *Here we implement* `select(k, A[1..n])` *assuming a linear time subroutine for* `median(A[1..n])`. */

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 $k > \ell$
   A. Let $B[1..-1]$ be a list/array of the $\lceil \ell \rceil - 1$ elements smaller than p.
   B. Return `select(k, B)`
6. If $k < \lceil n/2 \rceil$
   A. Let $B[1..n - \lceil n/2 \rceil]$ be a list/array of the $n - \lceil n/2 \rceil$ elements larger than p.
   B. Return `select(k - ⌈n/2⌉, B)`



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



$O(n)$

$\frac{9n}{10}$

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

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

$$\sum_{i=0}^{\infty} \left(\frac{9}{10}\right)^i = O(1)$$

$$O(n)$$

Chapter 11

Faster algorithms by divide and conquer

The lectures before this generally emphasized the difference between polynomial and exponential time algorithms. This theme is strongly expressed in our many discussions on recursion with caching (i.e., dynamic programming), which at its core, was based on identifying induction hypotheses that break down problems into polynomially many subproblems.

This chapter is also about recursive algorithms that rely on induction for correctness. However, for the problems discussed here, it is already easy to see that they have 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. In particular, they are not simulating brute-force algorithms as was often the case in dynamic programming.

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 will generally matching the running theme of "identifying subproblems". The second "conquer" component, so to speak, will highlight an additional angle of "combining subproblem solutions".

We recommend all three of [DPV08; Eri19; KT06] for more on this topic (including several more applications; notably, fast Fourier transforms).

11.1 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 a $O(n^2)$-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

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

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 $x_1y_2 + x_2y_1$. Observe that

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2.$$

The equation above rewrites two products as three, which seems like a step backwards. *Except two of them, x_1y_1 and x_2y_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 $x_1y_2 + x_2y_1$.[1]

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).

Now we implement `multiply`$(x[1..n], y[1..n])$ exactly as described.

`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 $2^n * Z_1 + 2^{n/2}(Z_3 - Z_1 - Z_2) + Z_2$

11.1.1 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

[1] 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.

to multiply integers with at most $n/2+2$ bits, plus $O(n)$ additional work. This gives the recurrence

$$T(n) \leq 3T(n/2+2) + O(n) \tag{11.1}$$

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

$$S(n) \leq 3S(n/2) + O(n) \tag{11.2}$$

$S(n)$ 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.

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 of the tree, h, is bounded above by $h \leq \log_2(n) + O(1)$.

We have

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

(a) is because the sum of coefficients is a geometrically increasing sequence (cf. appendix C.1.3).

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 (11.2). 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

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

Meanwhile substituting $S(n) = T(n+\alpha)$ into our target recurrence (11.2) 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 (11.1) 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, $S(n) = O\left(n^{\log_2(3)}\right)$. To transfer this to $T(n)$, we have

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

as desired.[2]

11.2 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 very fast, recursive algorithm for selection. To this end, it is necessary to define the intent of our recursive function, which also gives our induction hypothesis for correctness. We can do this even without having developed any algorithm details. We define, for an array $A[1..n]$ of comparable elements and an integer $i \in \{1, \ldots, n\}$:

$$\texttt{select}(k, A[1..n]) = \text{the } k\text{th smallest element out of } A[1..n].$$

We now focus on implementing `select(`i`)` exactly as specified. As mentioned above, we will be more strategic in the implementation than we typically were with dynamic programming. There, in dynamic programming, the emphasis was on being exhaustive for the sake of correctness. Here, the emphasis is on aggressively cutting down the search space – very much like binary search.

[2] 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".

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 make this explicitly clear, we define

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

and assume we can use `median`$(A[1..n])$ as a $O(n)$-time subroutine. `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. 11.1 for pseudocode of this algorithm.

The more interesting part of `select`$(k, A[1..n])$ is in understanding the running time. 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 would 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 (cf. appendix C.1.3), 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.

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

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

/* *We assume a linear time subroutine for* `apx-median`$(A[1..n])$. */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `median`(A).
3. If $k = \lceil n/2 \rceil$ then return p.
4. If $k > \lceil n/2 \rceil$
   A. Let $B[1..\lceil n/2 \rceil - 1]$ be a list/array of the $\lceil n/2 \rceil - 1$ elements smaller than p.
   B. Return `select`(k, B)
5. If $k < \lceil n/2 \rceil$
   A. Let $B[1..n - \lceil n/2 \rceil]$ be a list/array of the $n - \lceil n/2 \rceil$ elements larger than p.
   B. Return `select`$(k - \lceil n/2 \rceil, B)$

Figure 11.1: A (conceptual) divide-and-conquer algorithm for `select`$(k, A[1..n])$ using `median`$(A[1..n])$ to generate a pivot.

Selection via approximate medians. Of course, we do not actually 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. 11.2.

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

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

/* *Here we implement* `select`$(k, A[1..n])$ *assuming a linear time subroutine for* `median`$(A[1..n])$. */

1. If $n \leq 10$ then find the kth element by brute force.
2. Let $p =$ `apx-median`(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 a list/array of the $\lceil \ell \rceil - 1$ elements smaller than p.
6. If $k > \ell$ then return `select`$(k - \ell, B)$, where $B[1..n - \ell]$ is a list/array of the $n - \ell$ elements larger than p.

Figure 11.2: 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.

We 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 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 a constant 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.

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-*

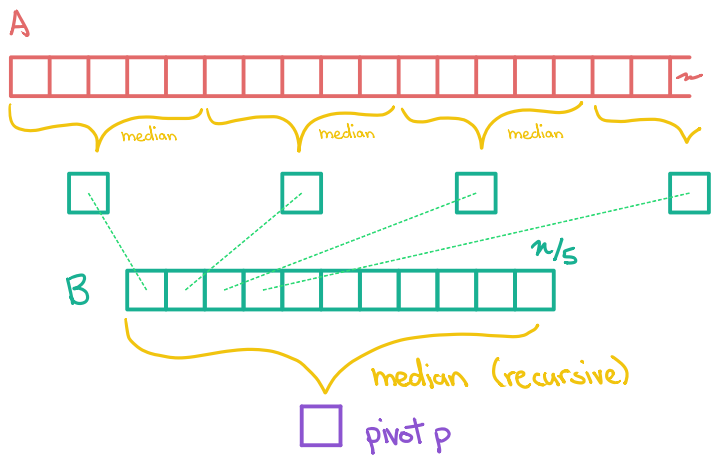


Figure 11.3: Sketch of the median-of-medians algorithm

of-medians, invented by Blum, Floyd, Pratt, Rivest, and Tarjan [BFP+72].[3]

At a high level, [BFP+72] 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

$$T(n) = T(7n/10) + T(n/5) + O(n). \tag{11.3}$$

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

Let us now explain how [BFP+72] selects the approximate median. It takes A and partitions it into subintervals of 5 elements each:

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

(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 (11.3) above. Pseudocode is given in fig. 11.4. Figure 11.6 on page 165 implements `select`$(k, A[1..n])$ with `median-of-medians`$(A[1..n])$.

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

[3] 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 $m = \lceil n/5 \rceil$.

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

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

Figure 11.4: The median-of-medians subroutine for approximating the median.

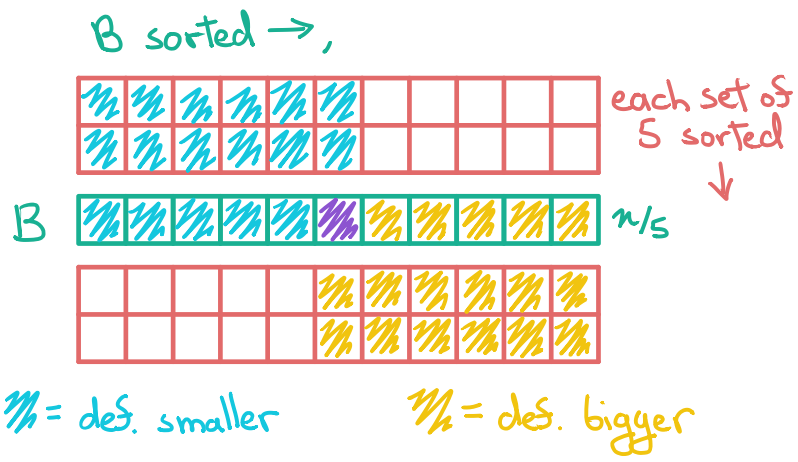


Figure 11.5: Illustration of the proof of lemma 11.1

Lemma 11.1. `median-of-medians(`$A[1..n]$`)` *returns an element p with rank ℓ in the range*

$$3n/10 \leq \ell \leq 7n/10 + 5.$$

Proof. Consider the array B 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 elements. Moreover, each of these elements is $\geq$ at least 2 additional elements in array, because it was the median of its own set of 5 elements. Thus $\ell \geq 3\lceil n/10 \rceil \geq 3n/10$.

Similarly one can show that p is less than or equal to at least $3n/10 - 5$ elements - the extra additional factor comes from the fact that the last interval may have less than 5 elements if n is not divisible by 5. ■

Above we asked the reader to try to analyze the recurrence

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

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

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

/* *Here we implement* `select`$(k, A[1..n])$ *assuming a linear time subroutine for* `median`$(A[1..n])$. */

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 $k > \ell$
   A. Let $B[1..-1]$ be a list/array of the $\lceil \ell \rceil - 1$ elements smaller than p.
   B. Return `select`(k, B)
6. If $k < \lceil n/2 \rceil$
   A. Let $B[1..n - \lceil n/2 \rceil]$ be a list/array of the $n - \lceil n/2 \rceil$ elements larger than p.
   B. Return `select`$(k - \lceil n/2 \rceil, B)$

Figure 11.6: The linear time divide-and-conquer algorithm for `select`$(k, A[1..n])$ by [BFP+72] using the median-of-medians as a pivot.

themselves. We now sketch the argument for the sake of completeness. A 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 11.2. *[BFP+72] Given an array of n numbers $A[1..n]$ and an index $k \in [n]$, the k largest 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 heads cutting 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.

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

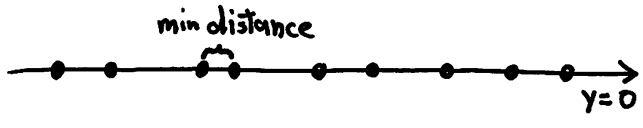
We consider the problem of computing the minimum pairwise distance among a set of points in the plane. The input consists of n points P in $\mathbb{R}^2$. The goal is to find the minimum Euclidean distance between any two points $p, q \in P$. Here we recall that for two points $p = (p_1, p_2)$ and $q = (q_1, q_2)$, 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.[4] Of course there is an $O(n^2)$-time algorithm where we simply calculate the distance. 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.



[4] 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 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 z. (We can compute z in $O(n)$ time via `select` from section 11.2). Let P_1 be the subset of points with x-coordinate $< z$ and P_2 be the subset of points with x-coordinate $\geq z$. 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 points *within* the P_1 and P_2; it remains to check the distance *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 $x = z$. So instead we do the following. We gather up all the points in P whose x-coordinates are within δ of z. Call this set P_3. Now consider the following geometric facts. First, P_3 lies in a narrow, vertical band of width 2δ. Second, 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. The latter generally forces P_3 to be spread apart. Combine the two facts leads to the following observation (proven formally in lemma 11.3): for any point $p \in P_3$, there are at most 10 other points with y-coordinate withing 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 points of one another in the sorted list.

Pseudocode of the divide-and-conquer algorithm described above is given in fig. 11.7. 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. 11.7.

`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 $x = z$ */*

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 $\delta = \min\{\delta_1, \delta\}$. *// $2T(n/2)$*

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

6. Let $P_3 = \{(x,y) \in P : |x - z| < \delta\}$ 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 11.7: Computing the minimum distance in $\mathbb{R}^2$ by divide-and-conquer along the horizontal axis.

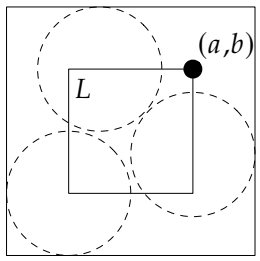
Lemma 11.3. *Let $b \in \mathbb{R}$. There are at most 10 points in P with coordinate in the rectangle $[a - \delta, a + \delta] \times [b, b + \delta]$,*

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

We split the rectangle into two squares

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

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 $[a - 1.5\delta, a + .5\delta] \times [b - .5\delta, b + 1.5\delta]$ 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. 11.7 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 11.3, 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

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