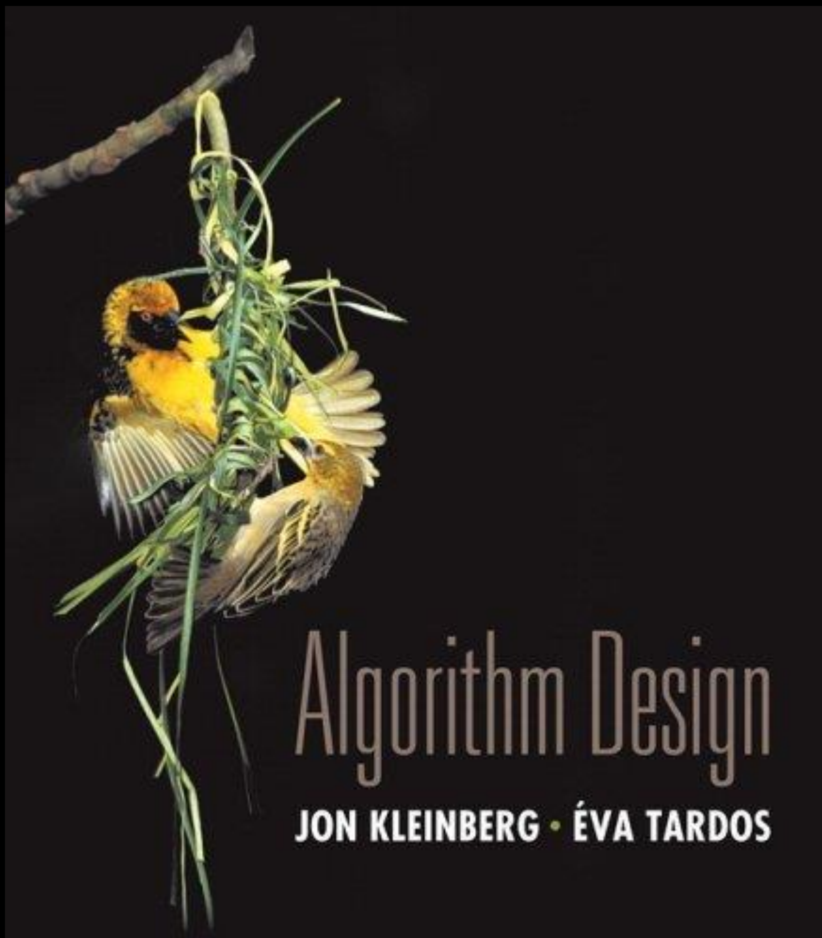


Chapter 5

Divide and Conquer

Recursion as Algorithmic Design Technique

Three important classes of algorithms

- Divide and conquer
- Back tracking

Recursion in design and analysis

- Dynamic programming

Recursion in design and proof of correctness, but time/space analysis is more "global"

Will see later

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size n into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Brute force: n^2.
- Divide-and-conquer: $n \log n$.

> Divide et impera.
> Veni, vidi, vici.
> \- *Julius Caesar*

Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Examples.

- Mergesort, quicksort, binary search
- Geometric problems: convex hull, nearest neighbors, line intersection, algorithms for planar graphs
- Algorithms for processing trees
- Many data structures (binary search trees, heaps, k-d trees,...)

Divide-and-Conquer: Analyzing Recursive Algorithms

Correctness. Almost always use strong induction

1. Prove correctness of the base cases (typically: $n \le constant$).

2. For an arbitrary n:

   - Assume algorithm performs correctly on all input sizes ($k \le n$).
   - Prove that the algorithm is correct on input size n.

Time / space analysis: Often use recurrence:

- Structure of the recurrence reflects the algorithm.

5.1 Mergesort

Sorting

Given n elements, rearrange in ascending order.

Applications.

- Sort a list of names.
- Display Google PageRank results.

Obvious application

- Find the closest pair.
- Binary search in a database.
- Find duplicates in a mailing list.

Problem is easier once sorted

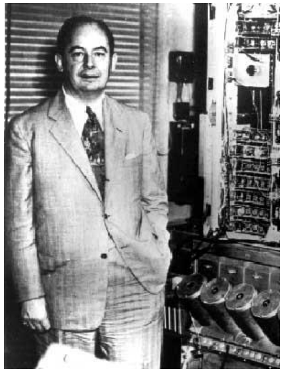
- Data compression.
- Computer graphics.
- Load balancing on a parallel computer.
- Computational biology

Non-obvious applications

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.



Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R		I	T	H	M	S	divide	O(1)
A	G	L	O	R		H	I	M	S	T	sort	2T(n/2)

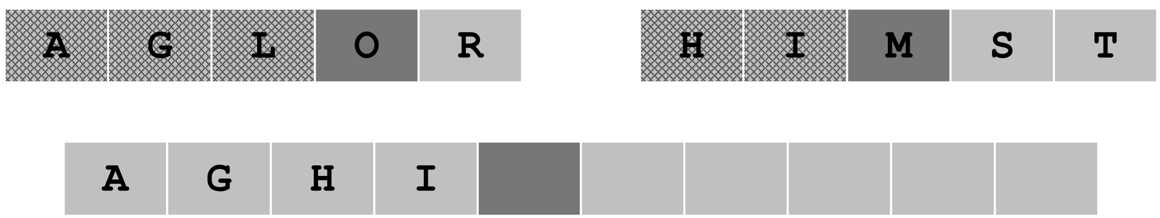
A	G	H	I	L	M	O	R	S	T	merge	O(n)

Merging

Merging. Combine two pre-sorted lists into a sorted whole.

How to merge efficiently?

- Linear number of comparisons.
- Use temporary array.



Challenge for the bored. In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage

A Useful Recurrence Relation

Def. T(n) = number of comparisons to mergesort an input of size n.

Mergesort recurrence.

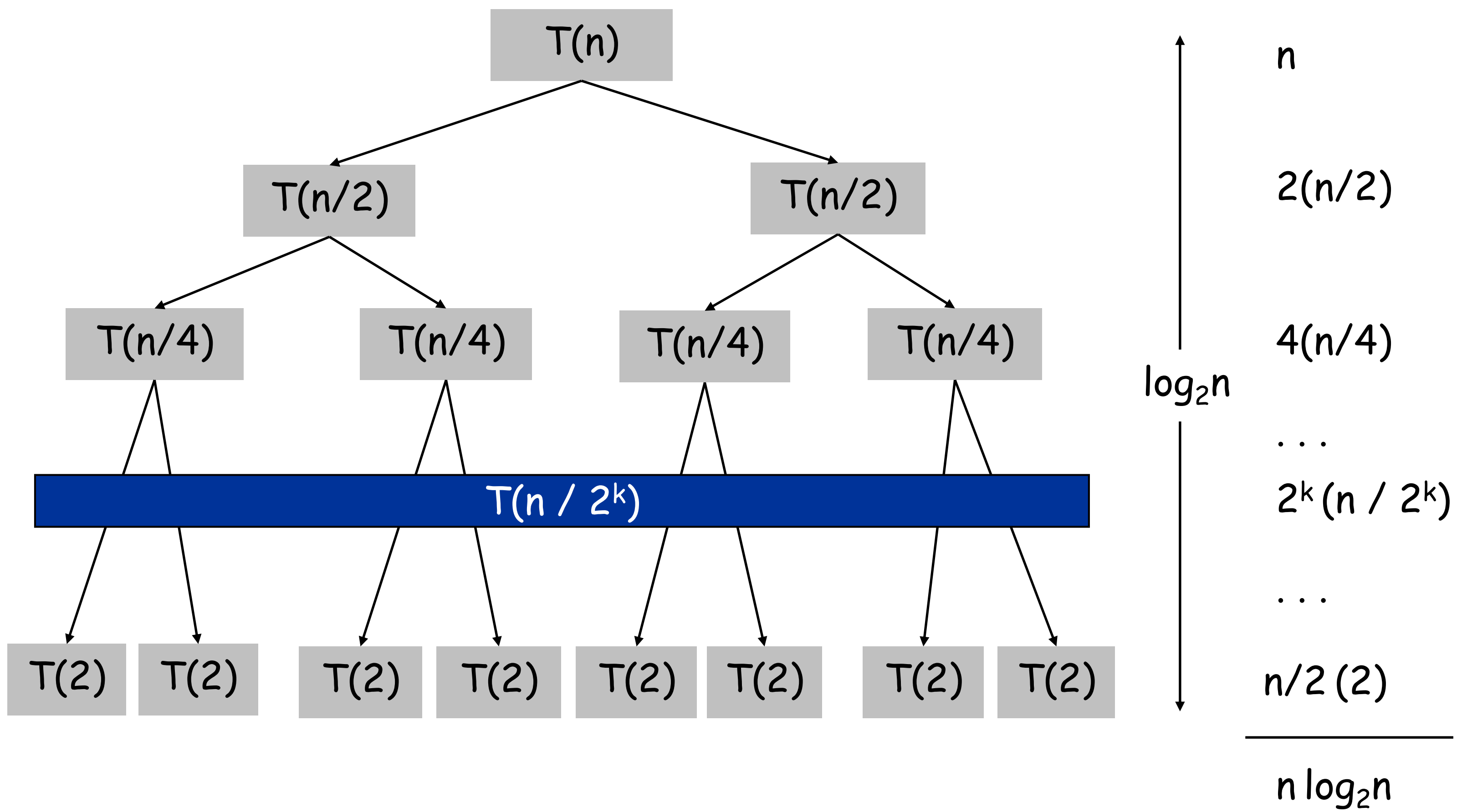
$$\mathrm{T}(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Solution. $T(n) = O(n \log_2 n)$.

Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\le$ with $=$.

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by Telescoping

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. For n > 1:

$$
\begin{aligned}
\frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\
&= \frac{T(n/2)}{n/2} + 1 \\
&= \frac{T(n/4)}{n/4} + 1 + 1 \\
&\cdots \\
&= \frac{T(n/n)}{n/n} + \underbrace{1 + \cdots + 1}_{\log_2 n} \\
&= \log_2 n
\end{aligned}
$$

Proof by Induction

Claim. If T(n) satisfies this recurrence, then $T(n) = n \log_2 n$.

assumes n is a power of 2

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

$$\begin{aligned} T(2n) &= 2T(n) + 2n \\ &= 2n \log_2 n + 2n \\ &= 2n\left(\log_2 (2n) - 1\right) + 2n \\ &= 2n \log_2 (2n) \end{aligned}$$

Proof by Master Theorem

The master theorem applies to recurrences of the form.

$$T(n) = a \cdot T(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive, that is $f(n) > 0$ for all $n > n_0$.

Compare $f(n)$ to $n^{\log_b a}$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$. (This means $f(n)$ grows polynomially slower than $n^{\log_b a}$ by an n^{ϵ} factor.) Then,

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

Proof by Master Theorem

The master theorem applies to recurrences of the form.

$$\mathrm{T}(n) = a \cdot T(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive, that is $f(n) > 0$ for all $n > n_0$.

Compare $f(n)$ to $n^{\log_b a}$:

Case 2: $f(n) = \Theta(n^{\log_b a})$. (This means $f(n)$ and $n^{\log_b a}$ grow at the same rate.) Then,

$$\mathrm{T}(n) = \Theta\left(n^{\log_b a} \log n\right)$$

Proof by Master Theorem

The master theorem applies to recurrences of the form.

$$T(n) = a \cdot T(n/b) + f(n)$$

where $a \geq 1, b > 1$ and f is asymptotically positive, that is $f(n) > 0$ for all $n > n_0$.

Compare $f(n)$ to $n^{\log_b a}$:

Case 3: $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and $f(n)$ satisfes the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$. (This means $f(n)$ grows polynomially faster than $n^{\log_b a}$ by an n^{ϵ} factor.) Then,

$$T(n) = \Theta(f(n))$$

Proof by Master Theorem

Applying Master Theorem to the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

$a = 2, b = 2, f(n) = n$.

Case 2 applies since $f(n) = \Theta(n^{\log_b a})$. Therefore,

$$T(n) = \Theta\left(n^{\log_b a} \log n\right) \Rightarrow T(n) = \Theta\left(n \log n\right)$$

Analysis of Mergesort Recurrence

Claim. If T(n) satisfies the following recurrence, then $T(n) \le n \lceil \lg n \rceil$.

$\uparrow$ $\log_2 n$

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\le n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\le \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1 \end{aligned}$$

5.3 Counting Inversions

Counting Inversions

Music site tries to match your song preferences with others.

- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.

- My rank: 1, 2, …, n.
- Your rank: $a_1, a_2, \ldots, a_n$.
- Songs i and j inverted if $i < j$, but $a_i > a_j$.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions

3-2, 4-2

Brute force: check all $\Theta(n^2)$ pairs i and j.

Applications

Applications.

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: $O(1)$.

1	5	4	8	10	2

6	9	12	11	3	7

Conquer: $2T(n / 2)$

5 blue-blue inversions

5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- Divide: separate list into two pieces.
- Conquer: recursively count inversions in each half.
- Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2

6	9	12	11	3	7

5 blue-blue inversions

8 green-green inversions

Conquer: 2T(n / 2)

9 blue-green inversions

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

Combine: count blue-green inversions

- Assume each half is sorted.
- Count inversions where a_i and a_j are in different halves.
- Merge two sorted halves into sorted whole.

to maintain sorted invariant

3	7	10	14	18	19

2	11	16	17	23	25
6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow \mathrm{T}(n) = O(n \log n)$$

Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A) ← Sort-and-Count(A)
   (rB, B) ← Sort-and-Count(B)
   (r , L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

Review Questions

Binary search

Integer exponentiation

5.4 Closest Pair of Points

Closest Pair of Points

Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.

Fundamental geometric primitive.

- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

fast closest pair inspired fast algorithms for these problems

Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

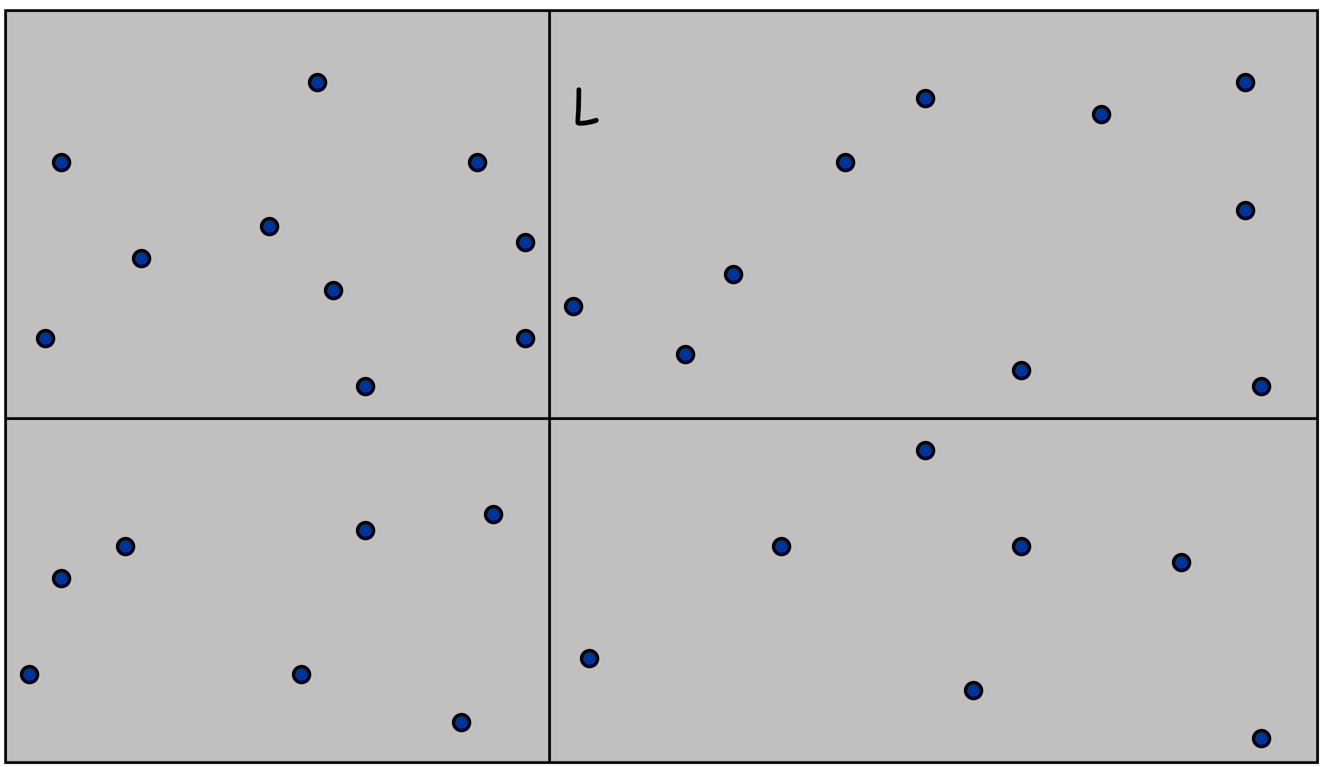
1-D version. $O(n \log n)$ easy if points are on a line.

Assumption. No two points have same x coordinate.

to make presentation cleaner

Closest Pair of Points: First Attempt

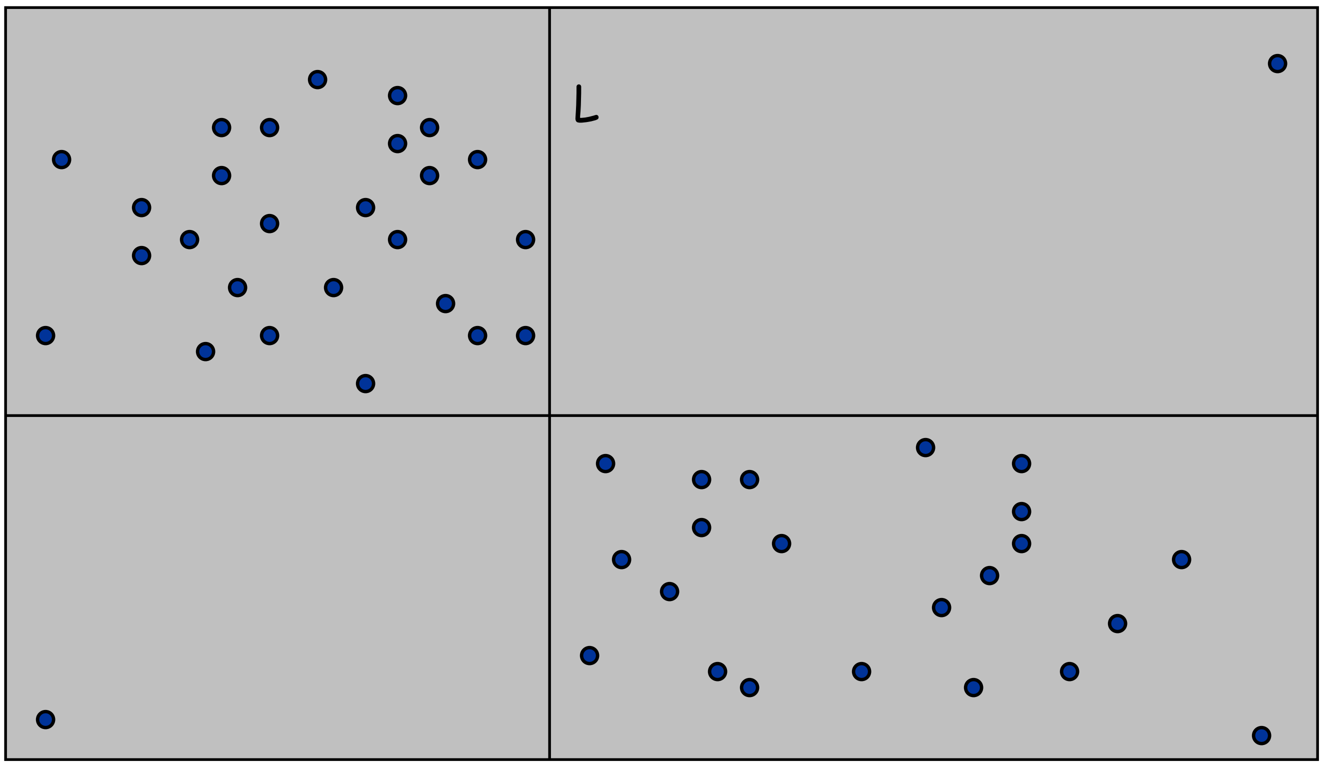
Divide. Sub-divide region into 4 quadrants.



Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

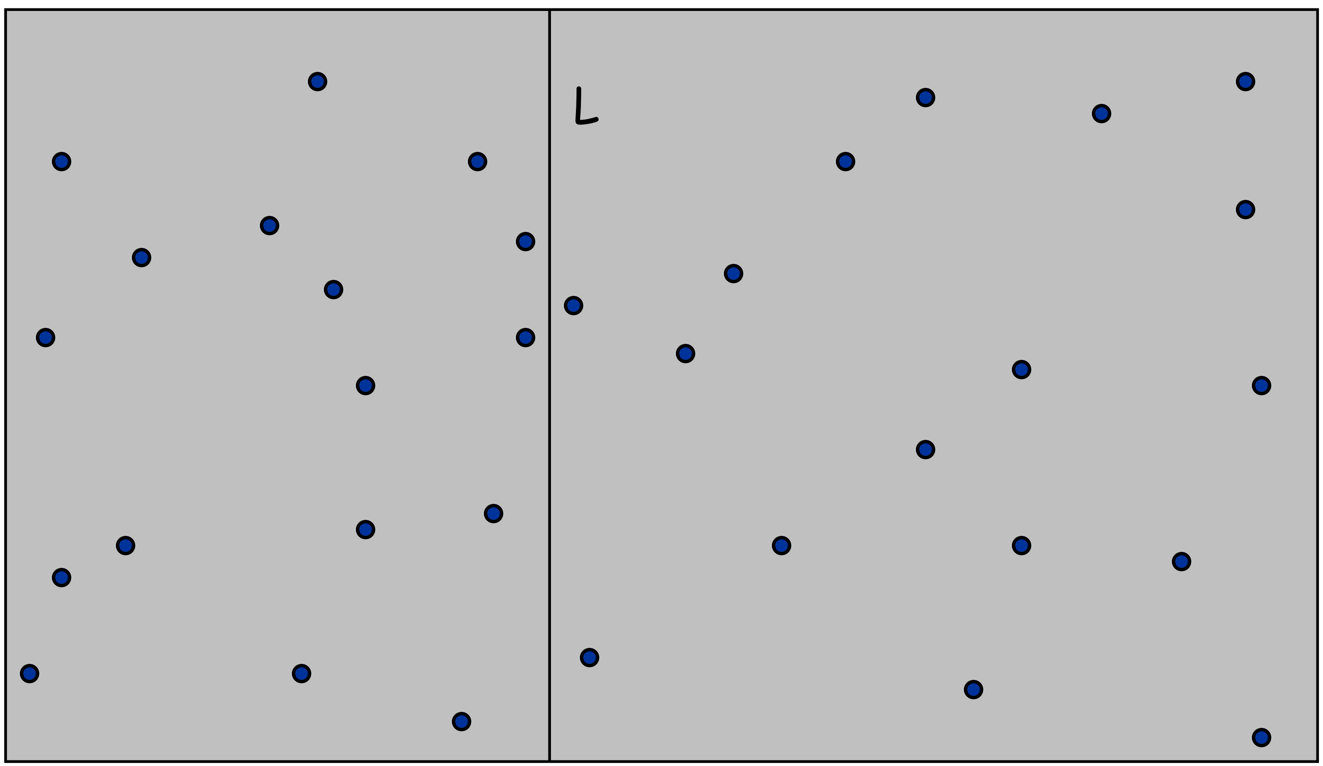
Obstacle. Impossible to ensure n/4 points in each piece.



Closest Pair of Points

Algorithm.

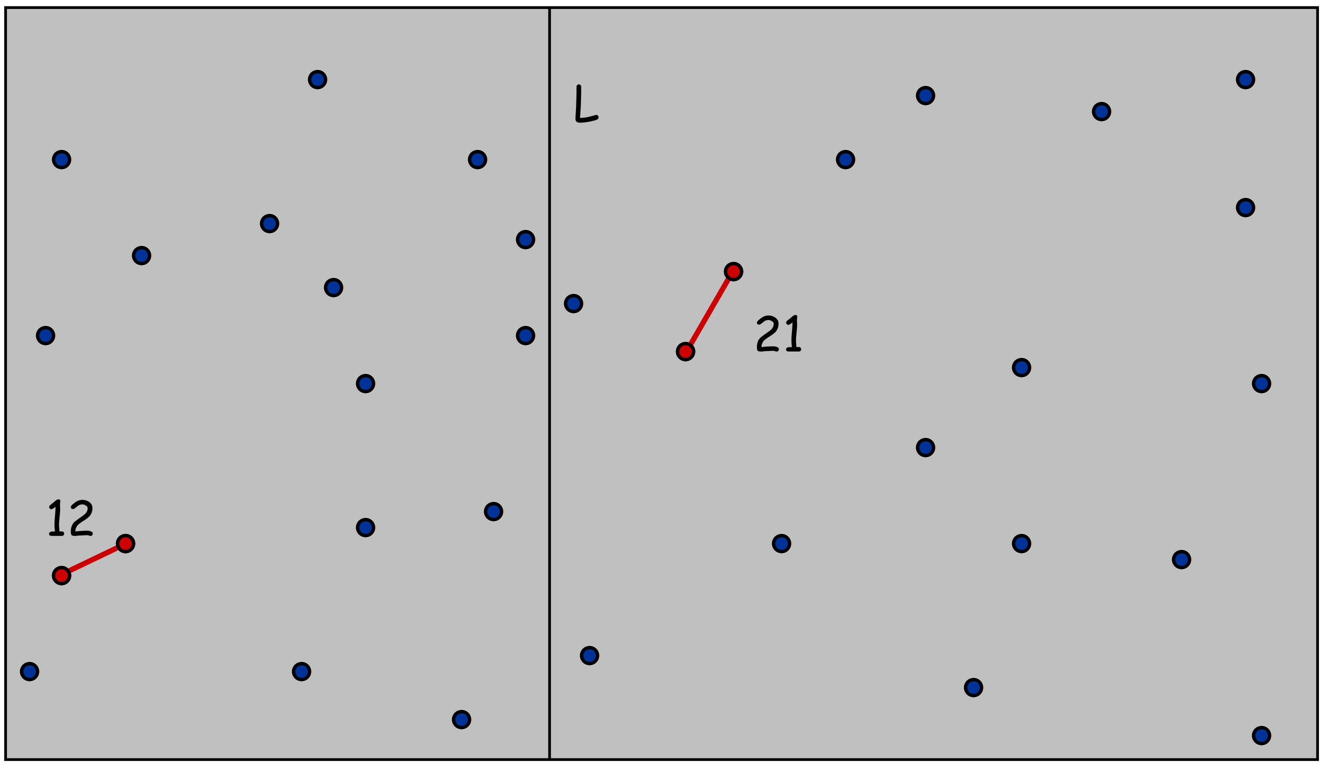
- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.



Closest Pair of Points

Algorithm.

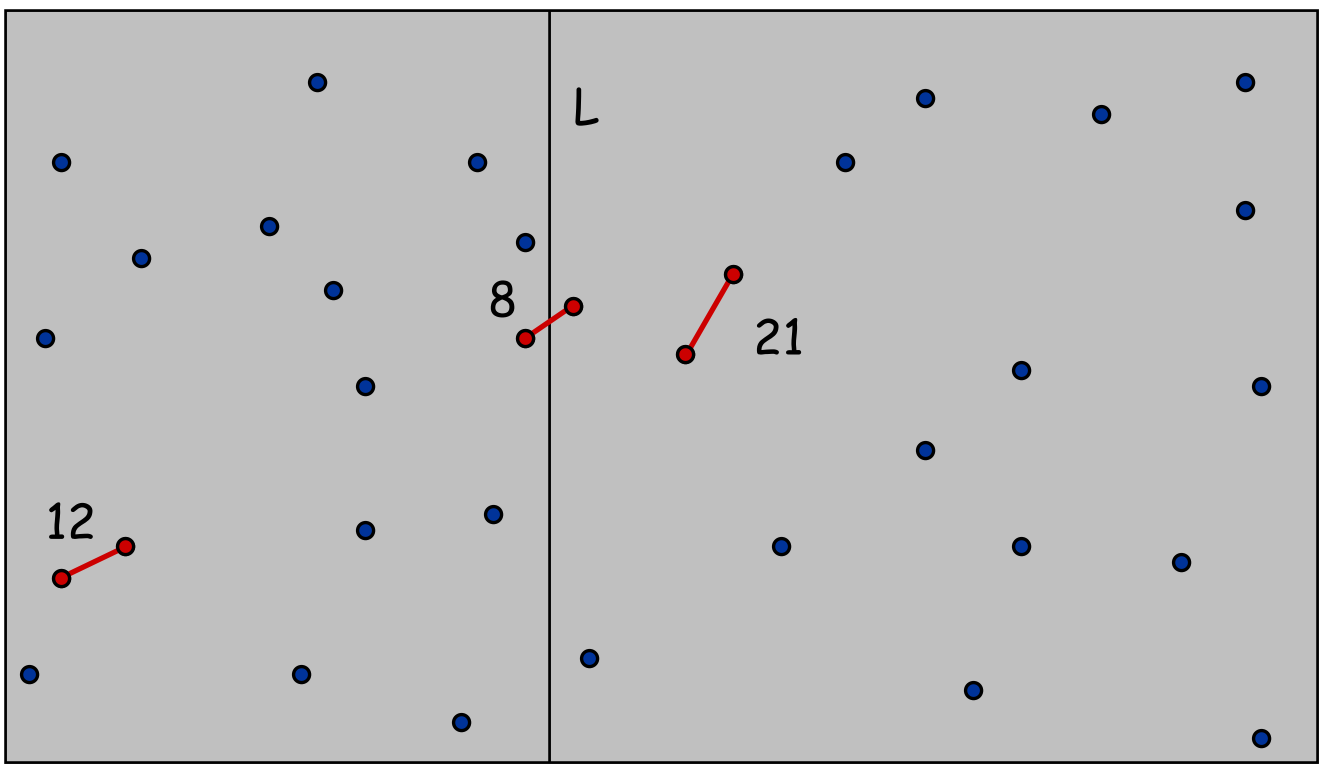
- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- Conquer: find closest pair in each side recursively.



Closest Pair of Points

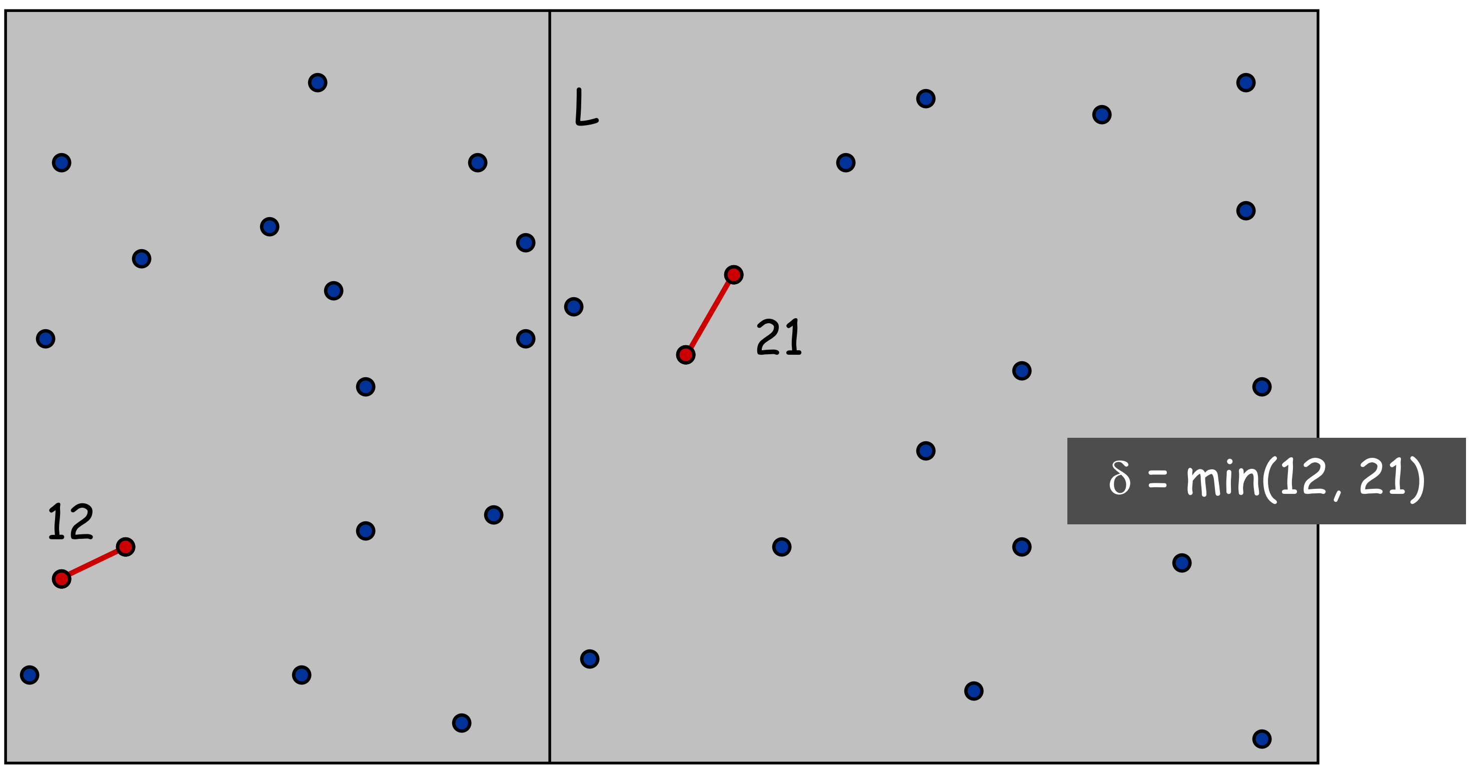
Algorithm.

- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side. ← seems like $\Theta(n^2)$
- Return best of 3 solutions.



Closest Pair of Points

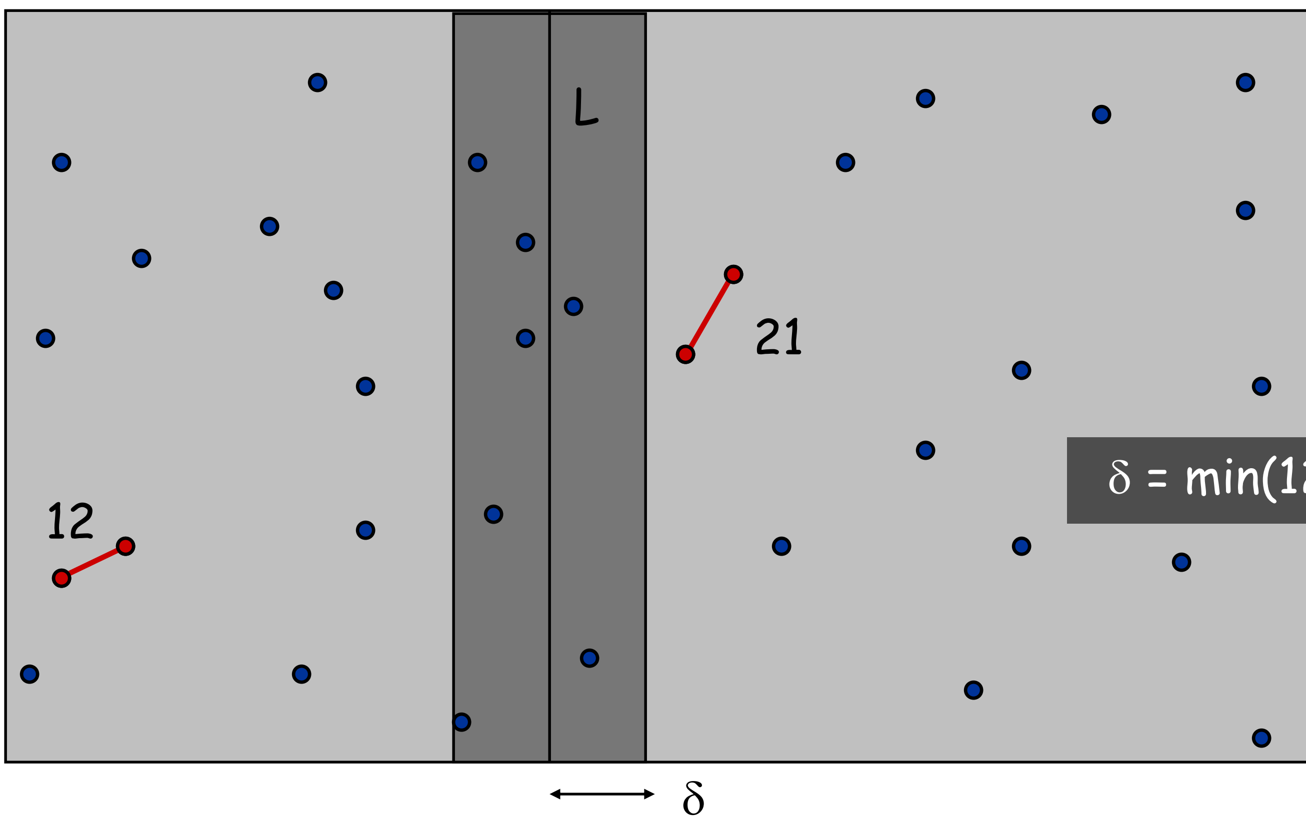
Find closest pair with one point in each side, assuming that distance < δ.



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.

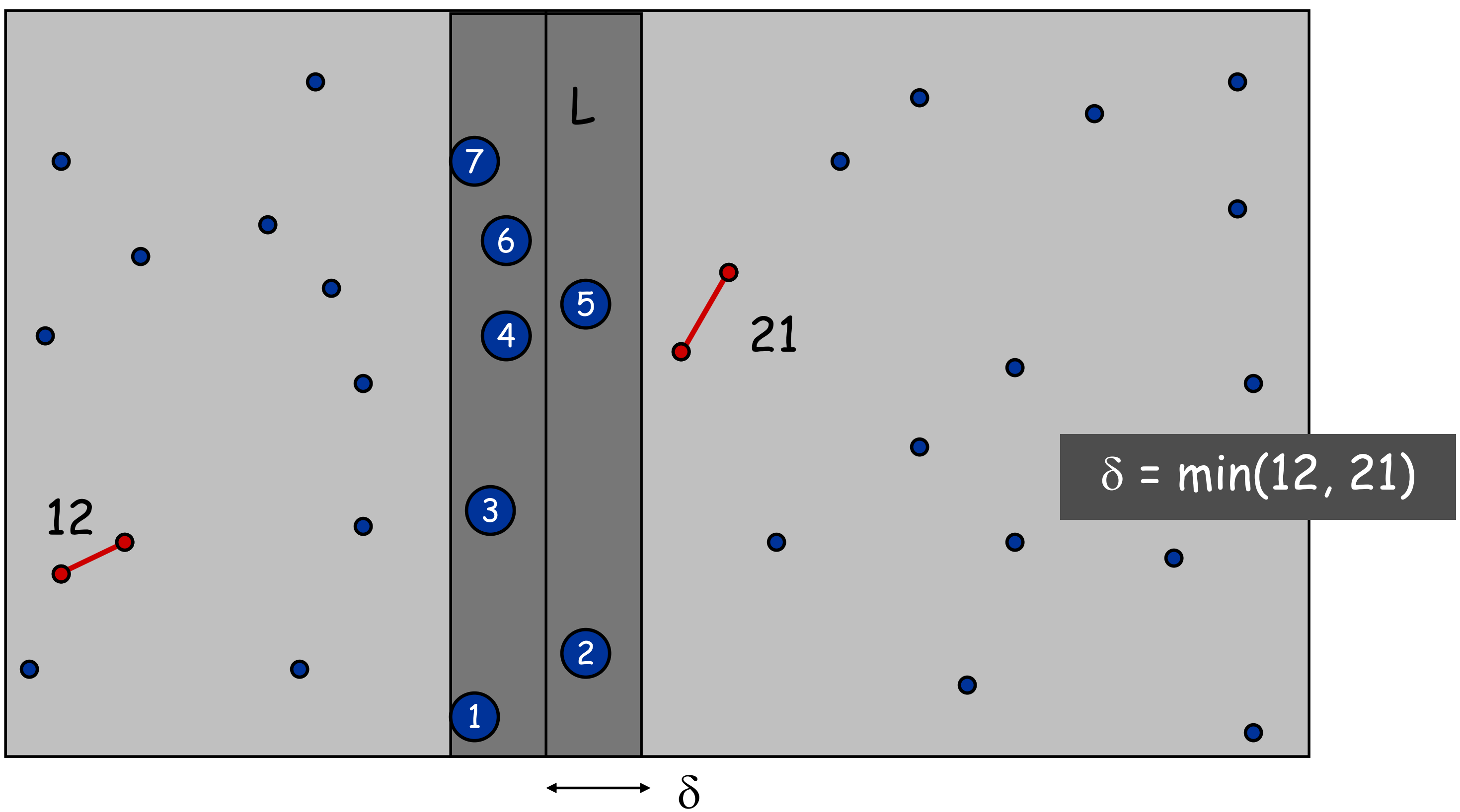


δ = min(12, 21)

Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

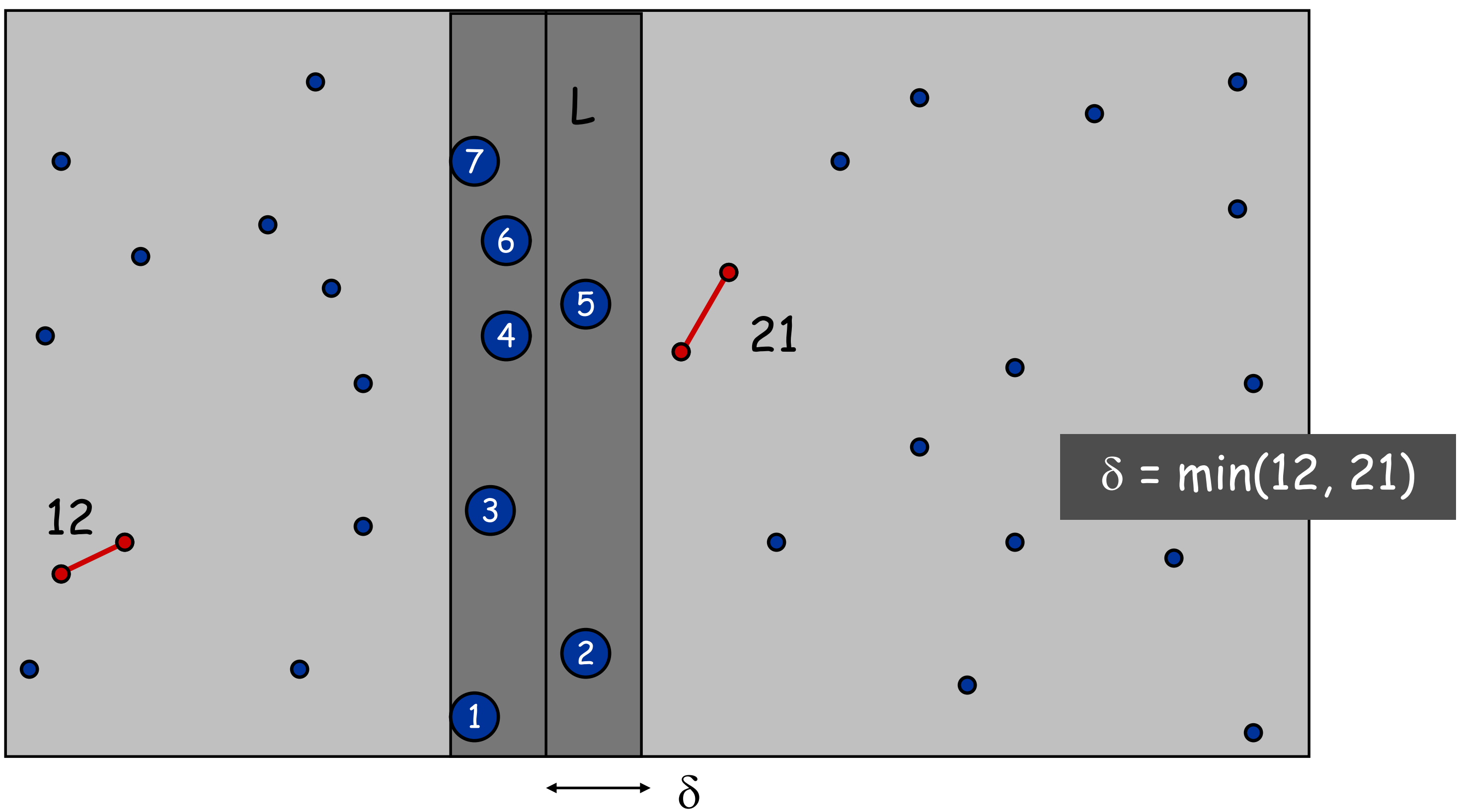
- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.



Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!



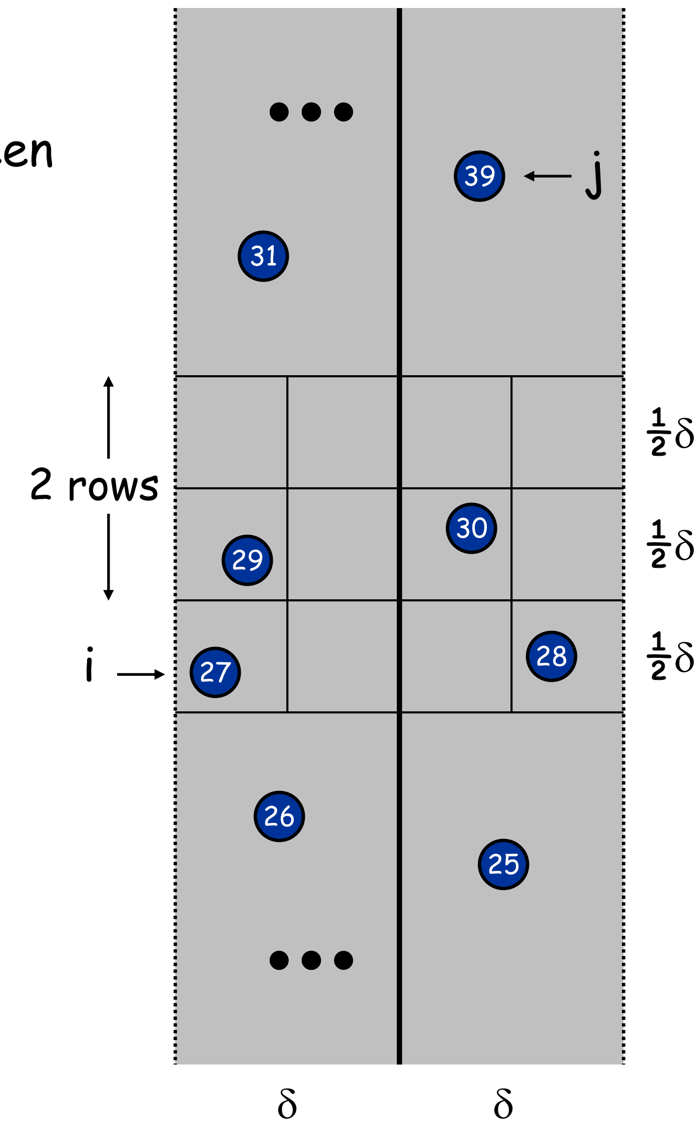
Closest Pair of Points

Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.

Pf.

- No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
- Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$. ▪



Fact. Still true if we replace 12 with 7.

Closest Pair Algorithm

```
Closest-Pair(p₁, …, pₙ) {
   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ₁ = Closest-Pair(left half)
   δ₂ = Closest-Pair(right half)
   δ  = min(δ₁, δ₂)

   Delete all points further than δ from separation line L

   Sort remaining points by y-coordinate.

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

$O(n \log n)$

$2T(n / 2)$

$O(n)$

$O(n \log n)$

$O(n)$

Closest Pair of Points: Analysis

Running time.

$$T(n) \leq 2T\big(n/2\big) + O(n \log n) \;\Rightarrow\; T(n) = O(n \log^2 n)$$

Q. Can we achieve O(n log n)?

A. Yes. Don't sort points in strip from scratch each time.

- Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
- Sort by merging two pre-sorted lists.

$$T(n) \leq 2T\big(n/2\big) + O(n) \;\Rightarrow\; T(n) = O(n \log n)$$

5.5 Integer Multiplication

Arithmetic on Large Integers

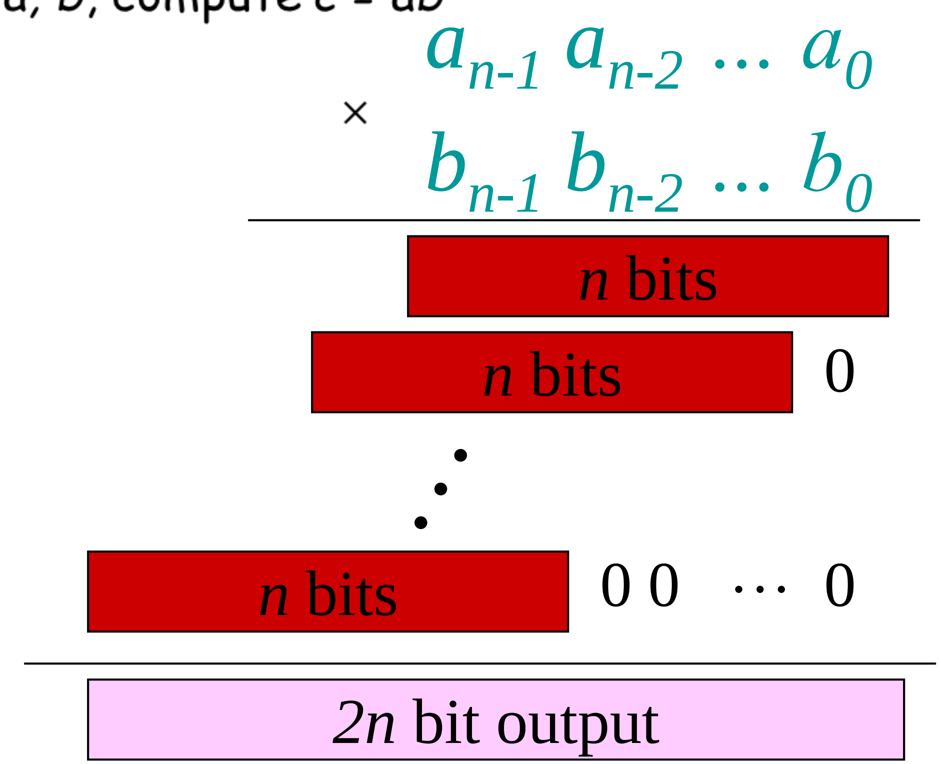
Addition: Given n-bit integers a, b (in binary), compute $c=a+b$

- $O(n)$ bit operations.

Multiplication: Given n-bit integers a, b, compute $c = ab$

Naïve (grade-school) algorithm:

- Write a,b in binary
- Compute n intermediate products
- Do n additions
- Total work: $\Theta(n^2)$



Slide by S. Raskhodnikova and A. Smith

Multiplying large integers

Divide and Conquer (warmup):

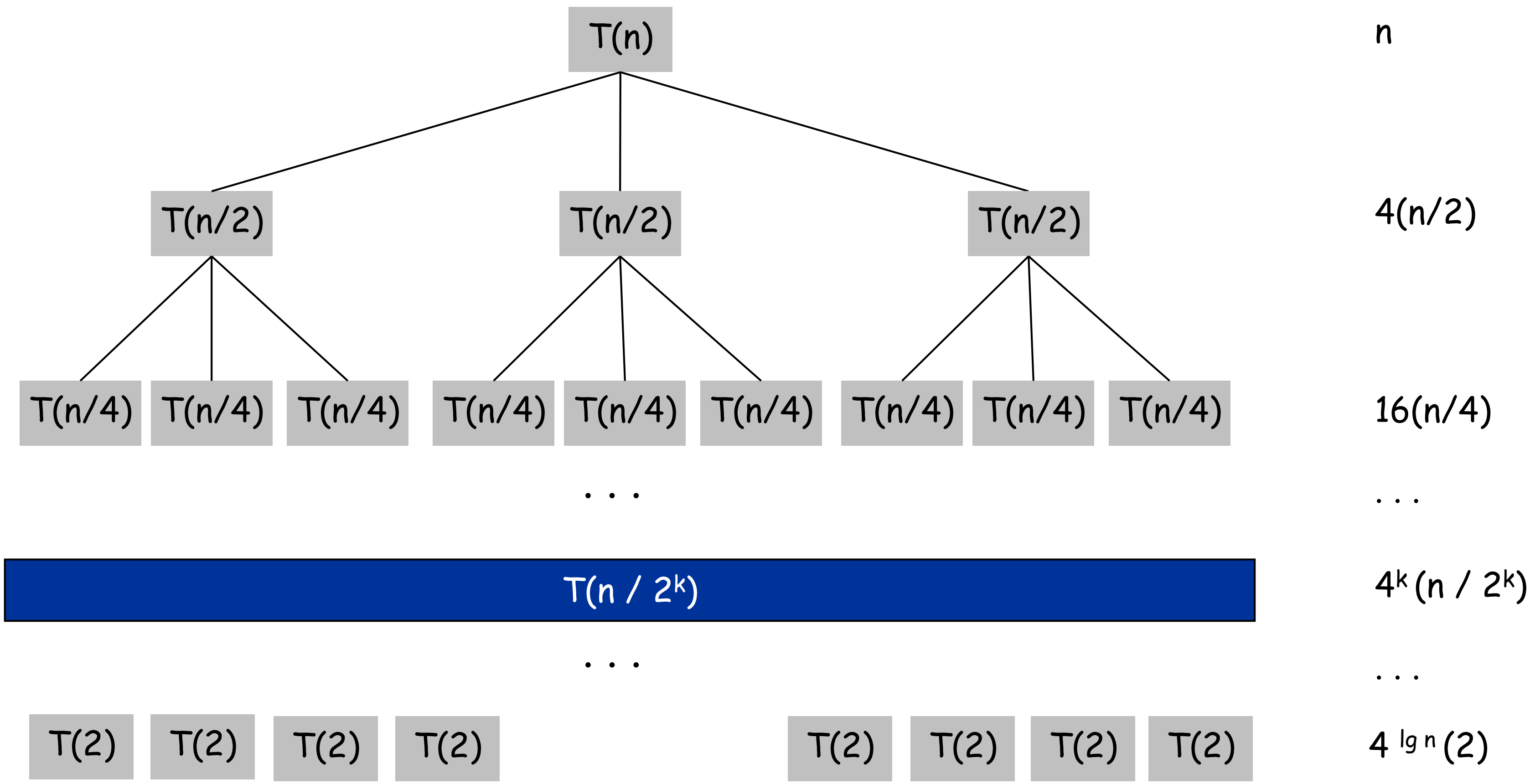
- Write $a = A_1 2^{n/2} + A_0$
  $b = B_1 2^{n/2} + B_0$
- We want $ab = A_1B_1 2^n + (A_1B_0 + B_1A_0) 2^{n/2} + A_0B_0$
- Multiply $n/2$ –bit integers recursively
- $T(n) = 4T(n/2) + \Theta(n)$
- Alas! this is still $\Theta(n^2)$

Slide by S. Raskhodnikova and A. Smith

Recursion Tree Argument

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 4T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n\left(\tfrac{4}{2}\right)^k = O(n^{\log_2 4}) = O(n^2)$$



Multiplying large integers

Divide and Conquer (Karatsuba's algorithm):

- Write $a = A_1\, 2^{n/2} + A_0$
  $b = B_1\, 2^{n/2} + B_0$

- We want $ab = A_1B_1\, 2^{n/2} + (A_1B_0 + B_1A_0)\, 2^{n/2} + A_0B_0$

Karatsuba's idea:

$$(A_0+A_1)\,(B_0 + B_1) = A_0B_0 + A_1B_1 + (A_0B_1 + B_1A_0)$$

- We can get away with 3 multiplications! (in yellow)
-

$$\boldsymbol{x} = A_1B_1 \qquad \boldsymbol{y} = A_0B_0 \qquad \boldsymbol{z} = (A_0+A_1)(B_0+B_1)$$

- Now we use

$$\begin{aligned} ab &= A_1B_1\, 2^n + (A_1B_0 + B_1A_0)\, 2^{n/2} + A_0B_0 \\ &= \boldsymbol{x}\; 2^n + (\boldsymbol{z}-\boldsymbol{x}-\boldsymbol{y})\, 2^{n/2} + \boldsymbol{y} \end{aligned}$$

Slide by S. Raskhodnikova and A. Smith

Karatsuba Multiplication

To multiply two n-digit integers:

- Add two $\frac{1}{2}$n digit integers.
- Multiply three $\frac{1}{2}$n-digit integers.
- Add, subtract, and shift $\frac{1}{2}$n-digit integers to obtain result.

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

$$\mathrm{T}(n) \le \underbrace{T\big(\lfloor n/2 \rfloor\big) + T\big(\lceil n/2 \rceil\big) + T\big(1+\lceil n/2 \rceil\big)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}$$

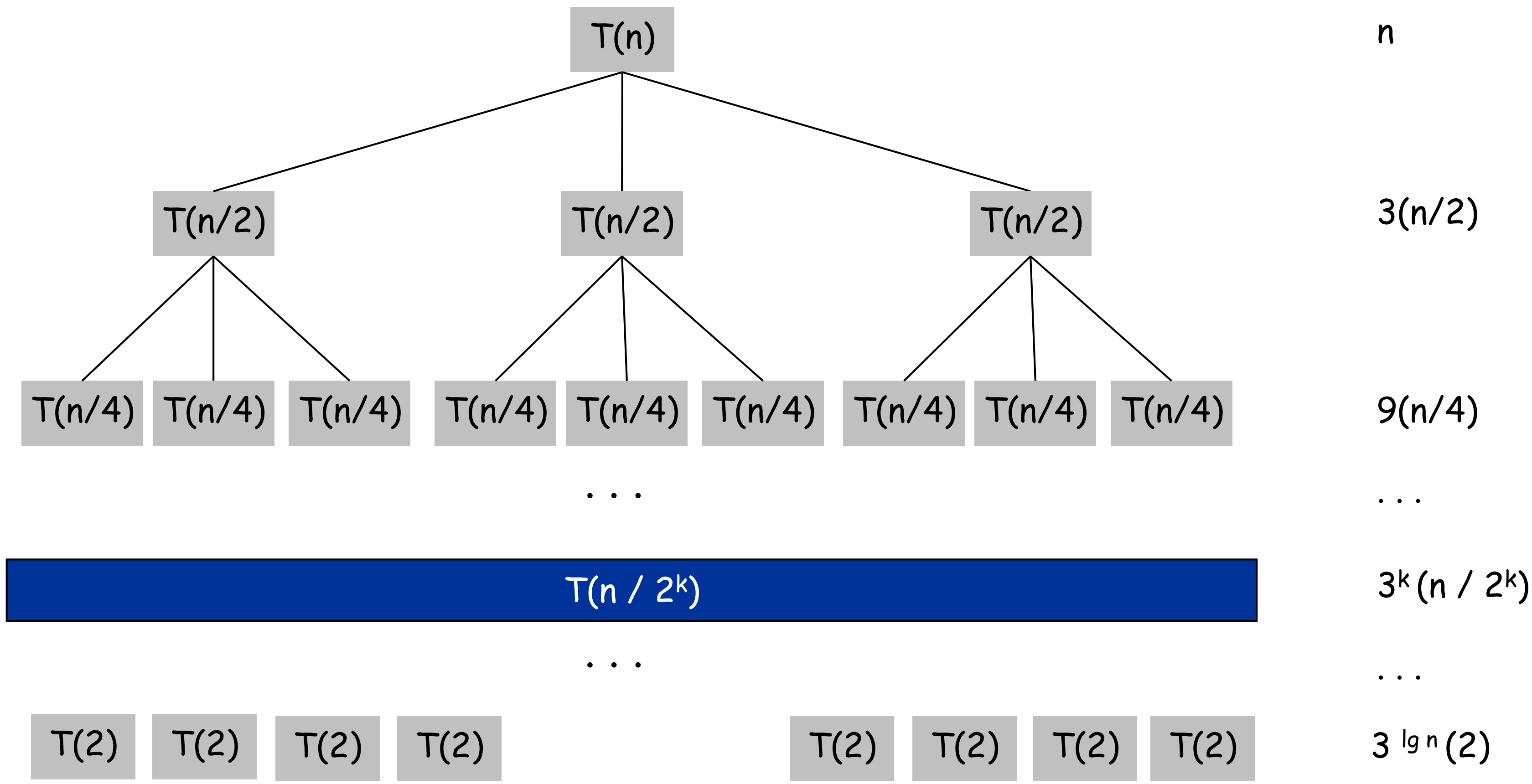
$$\Rightarrow \mathrm{T}(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Faster algorithm (FFT-based): O(n log n (log log n))

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n\left(\tfrac{3}{2}\right)^k = O(n^{\log_2 3})$$



Matrix Multiplication

Matrix Multiplication

Matrix multiplication. Given two n-by-n matrices A and B, compute $C = AB$.

$$c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

Brute force. $\Theta(n^3)$ arithmetic operations.

Fundamental question. Can we improve upon brute force?

Brute-force Matrix Multiplication

$$
\begin{array}{l}
\textbf{for } i \leftarrow 1 \textbf{ to } n \\
\quad \textbf{do for } j \leftarrow 1 \textbf{ to } n \\
\qquad\qquad \textbf{do } c_{ij} \leftarrow 0 \\
\qquad\qquad\quad \textbf{for } k \leftarrow 1 \textbf{ to } n \\
\qquad\qquad\qquad \textbf{do } c_{ij} \leftarrow c_{ij} + a_{ik} \times b_{kj}
\end{array}
$$

Running time $= \Theta(n^3)$

Divide and Conquer Algorithm

IDEA:

$n \times n$ matrix $= 2\times2$ matrix of $(n/2)\times(n/2)$ submatrices:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C \quad = \quad A \quad \times \quad B$$

$$\left.\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}\right\}$$

recursive

8 mults of $(n/2)\times(n/2)$ submatrices

4 adds of $(n/2)\times(n/2)$ submatrices

Slide by S. Raskhodnikova and A. Smith

Matrix Multiplication: Warmup

Divide-and-conquer.

- Divide: partition A and B into $\frac{1}{2}$n-by-$\frac{1}{2}$n blocks.
- Conquer: multiply 8 $\frac{1}{2}$n-by-$\frac{1}{2}$n recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\mathrm{T}(n) = \underbrace{8T\big(n/2\big)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

Matrix Multiplication: Strassen's idea

- Multiply 2×2 matrices with only 7 recursive mults.

$$
\begin{aligned}
M_1 &= A_{11}\times(B_{12}-B_{22}) \\
M_2 &= (A_{11}+A_{12})\times B_{22} \\
M_3 &= (A_{21}+A_{22})\times B_{11} \\
M_4 &= A_{22}\times(B_{21}-B_{11}) \\
M_5 &= (A_{11}+A_{22})\times(B_{11}+B_{22}) \\
M_6 &= (A_{12}-A_{22})\times(B_{21}+B_{22}) \\
M_7 &= (A_{11}-A_{21})\times(B_{11}+B_{12})
\end{aligned}
$$

$$
\begin{aligned}
C_{11} &= M_5+M_4-M_2+M_6 \\
C_{12} &= M_1+M_2 \\
C_{21} &= M_3+M_4 \\
C_{22} &= M_5+M_1-M_3-M_7
\end{aligned}
$$

7 mults, 18 adds/subs.

Slide by S. Raskhodnikova and A. Smith

Fast Matrix Multiplication

Fast matrix multiplication. (Strassen, 1969)

- Divide: partition A and B into ½n-by-½n blocks.
- Compute: 14 ½n-by-½n matrices via 10 matrix additions.
- Conquer: multiply 7 ½n-by-½n matrices recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume n is a power of 2.
- T(n) = # arithmetic operations.

$$\mathrm{T}(n) = \underbrace{7T\big(n/2\big)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow \mathrm{T}(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Practice

Implementation issues.

- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception: "Strassen is only a theoretical curiosity."

- Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when $n \sim 2{,}500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax=b$, determinant, eigenvalues, and other matrix ops.

Fast Matrix Multiplication in Theory

Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?

A. Yes! [Strassen, 1969]

$$\Theta(n^{\log_2 7}) = O(n^{2.81})$$

Q. Multiply two 2-by-2 matrices with only 6 scalar multiplications?

A. Impossible. [Hopcroft and Kerr, 1971]

$$\Theta(n^{\log_2 6}) = O(n^{2.59})$$

Q. Two 3-by-3 matrices with only 21 scalar multiplications?

A. Also impossible.

$$\Theta(n^{\log_3 21}) = O(n^{2.77})$$

Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?

A. Yes! [Pan, 1980]

$$\Theta(n^{\log_{70} 143640}) = O(n^{2.80})$$

Decimal wars.

- December, 1979: $O(n^{2.521813})$.
- January, 1980: $O(n^{2.521801})$.

Fast Matrix Multiplication in Theory

Best known. $O(n^{2.3728...})$ [Williams, 2014.]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Selection in Linear Time

Order statistics

Select the ith smallest of n elements (the element with ***rank*** i).

- $i = 1$: ***minimum***;
- $i = n$: ***maximum***;
- $i = \lfloor (n+1)/2 \rfloor$ or $\lceil (n+1)/2 \rceil$: ***median***.

Naive algorithm: Sort and index ith element.

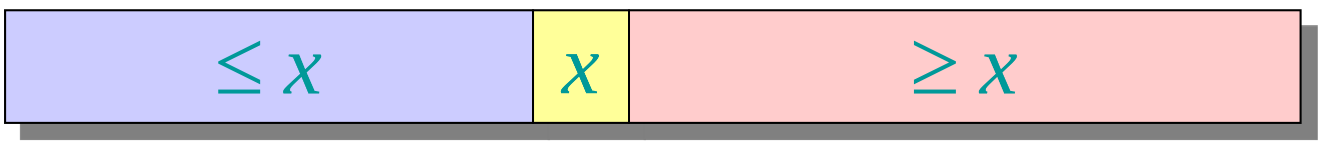
Worst-case running time $= \Theta(n \lg n) + \Theta(1)$
$= \Theta(n \lg n)$,

using merge sort or heapsort (*not* quicksort).

Divide and conquer

Order Statistics in an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



2. ***Conquer:*** Recurse on one subarray.
3. ***Combine:*** Trivial.

Key: *Linear-time partitioning subroutine.*

Partitioning subroutine

PARTITION(A, p, q) ⊳ $A[p\,.\,.\,q]$
$x \leftarrow A[p]$ ⊳ pivot $= A[p]$
$i \leftarrow p$
for $j \leftarrow p + 1$ **to** q
do if $A[j] \leq x$
then $i \leftarrow i + 1$
exchange $A[i] \leftrightarrow A[j]$
exchange $A[p] \leftrightarrow A[i]$
return i

Running time $= O(n)$ for n elements.

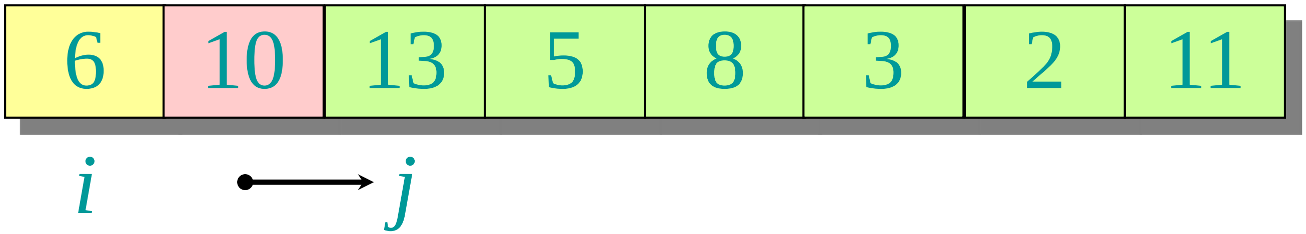
Invariant:

x	$\leq x$	$\geq x$	?
p	i	j	q

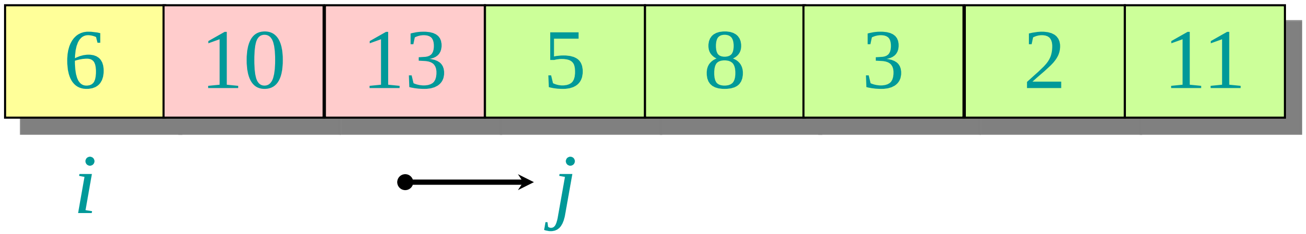
Example of partitioning

6	10	13	5	8	3	2	11
i	*j*						

Example of partitioning



Example of partitioning



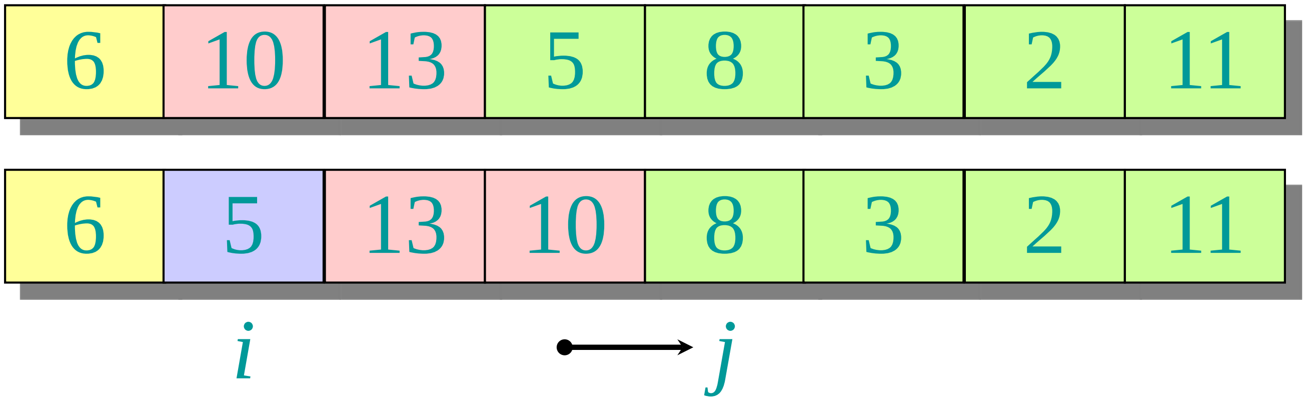
Example of partitioning

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

i j

Example of partitioning



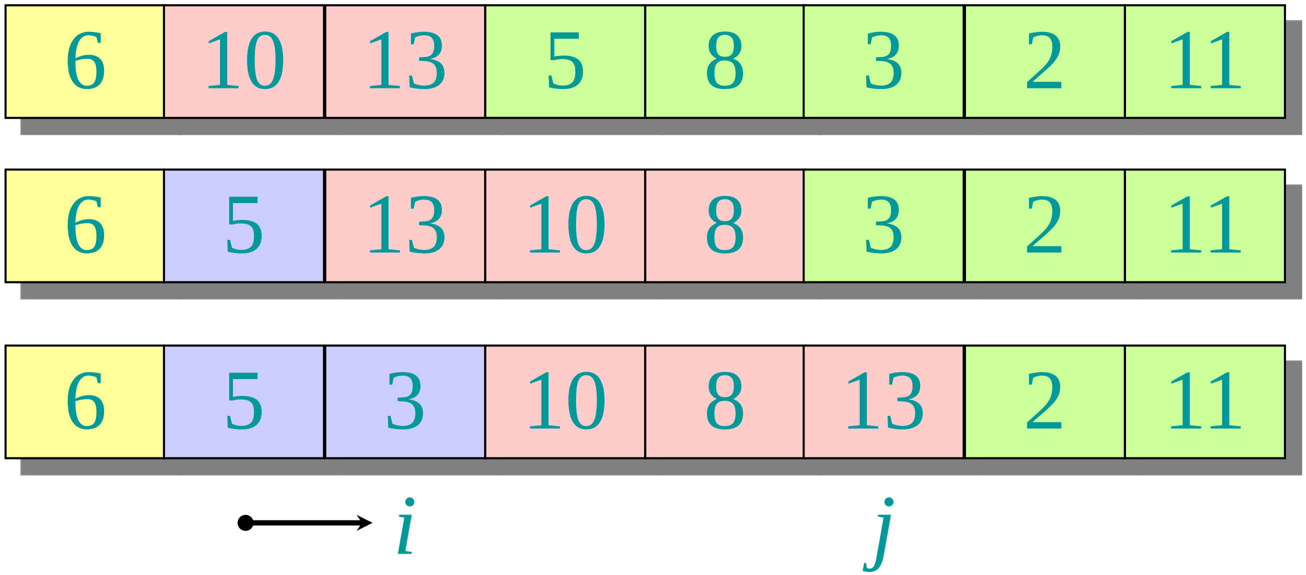
Example of partitioning

6	10	13	5	8	3	2	11

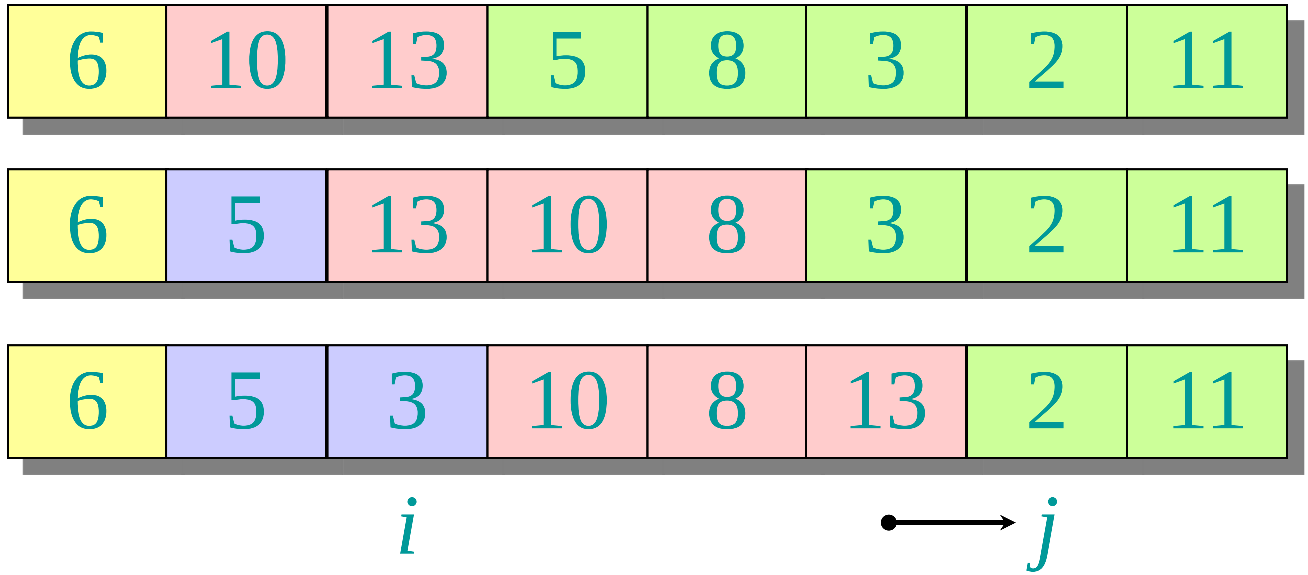
6	5	13	10	8	3	2	11

i j

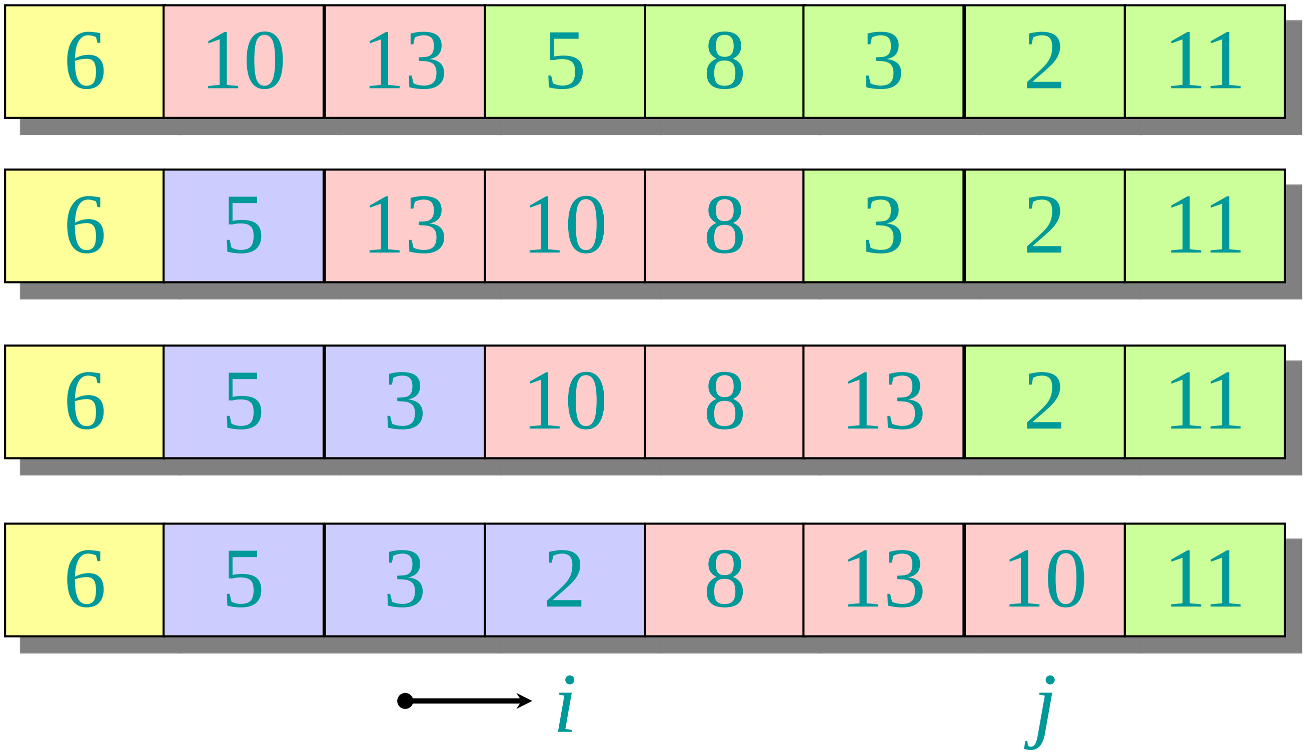
Example of partitioning



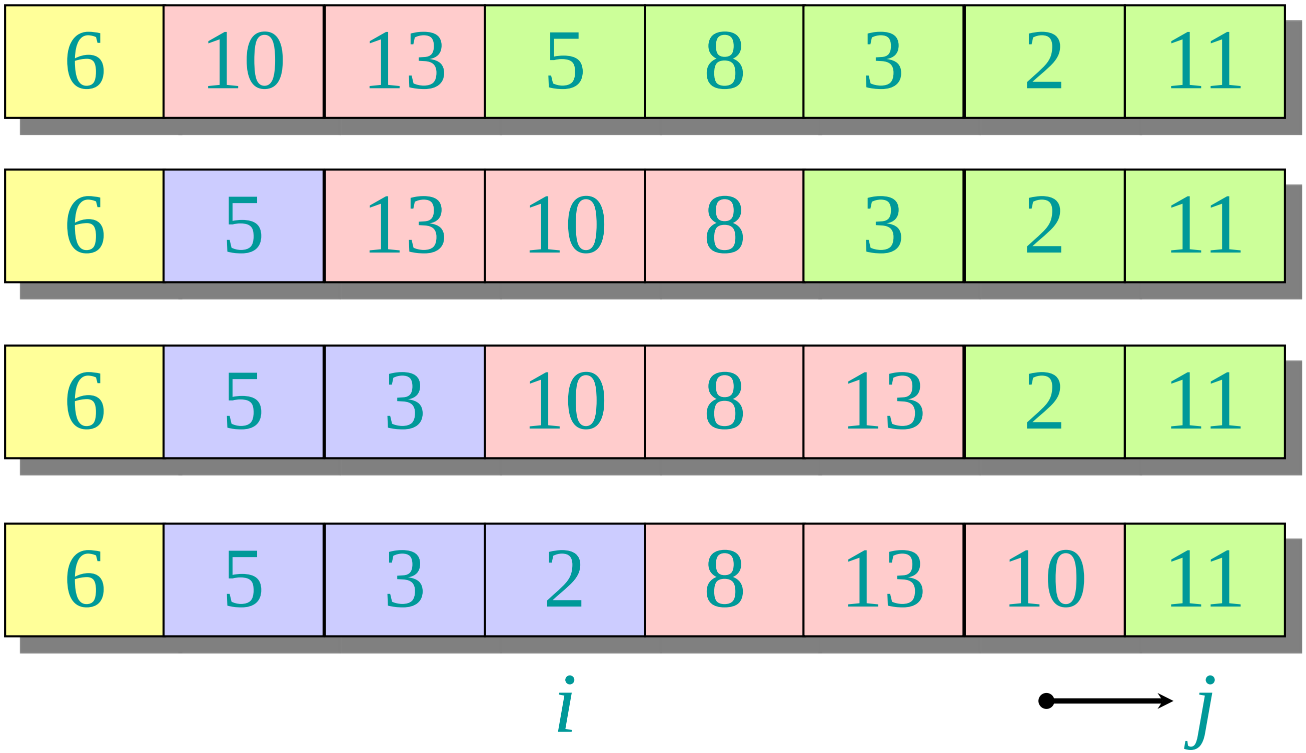
Example of partitioning



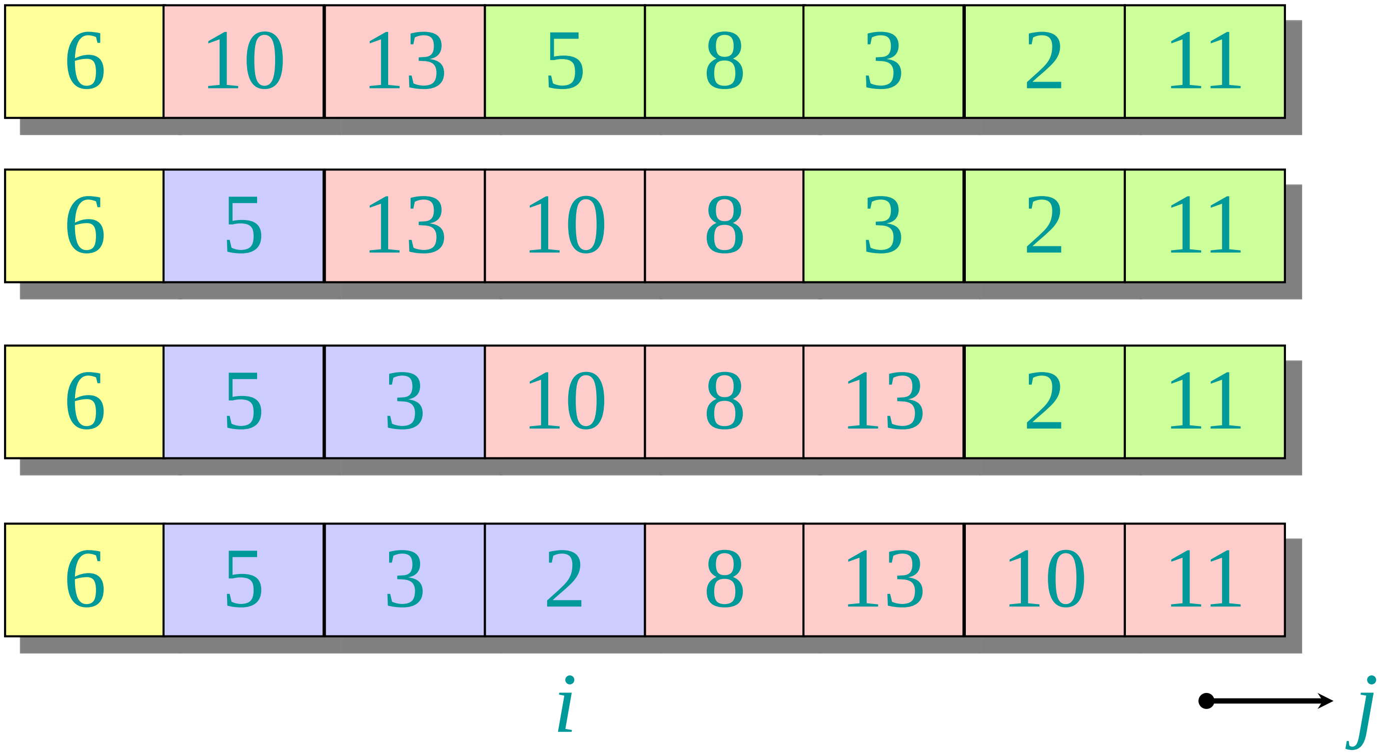
Example of partitioning



Example of partitioning



Example of partitioning

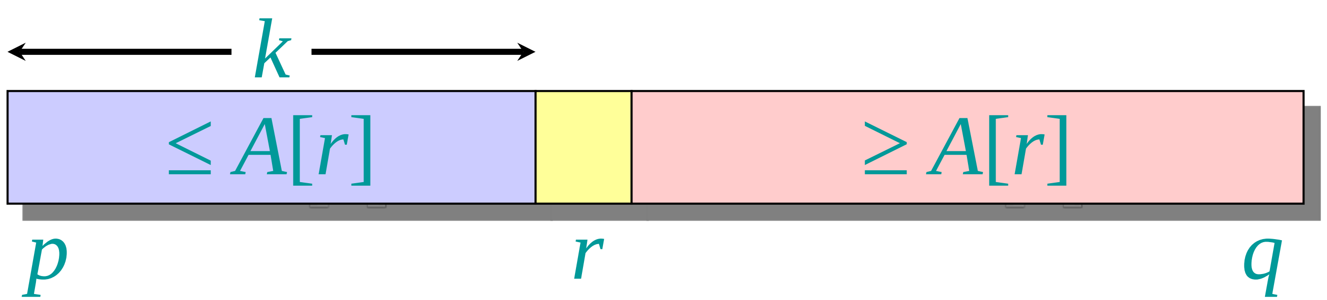


Example of partitioning

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			i				

Divide-and-conquer algorithm

SELECT(A, p, q, i) ⊳ ith smallest of $A[p\,..\,q]$
  if $p = q$ **then return** $A[p]$
  $r \leftarrow$ pivot ⊳ **Later: how to choose the pivot**
  $k \leftarrow r - p + 1$ ⊳ $k = \text{rank}(A[r])$
  if $i = k$ **then return** $A[r]$
  if $i < k$
    then return SELECT$(A, p, r - 1, i)$
    else return SELECT$(A, r + 1, q, i - k)$



Example

Select the $i = 7$th smallest:

6	10	13	5	8	3	2	11	$i = 7$

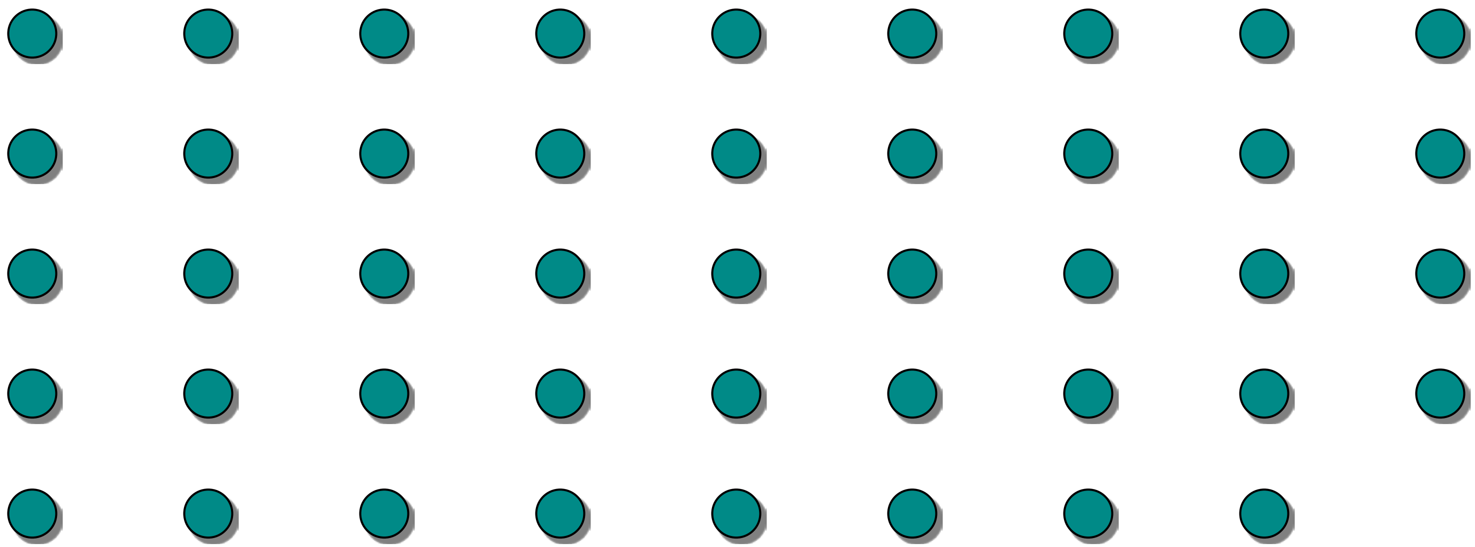
pivot

Partition:

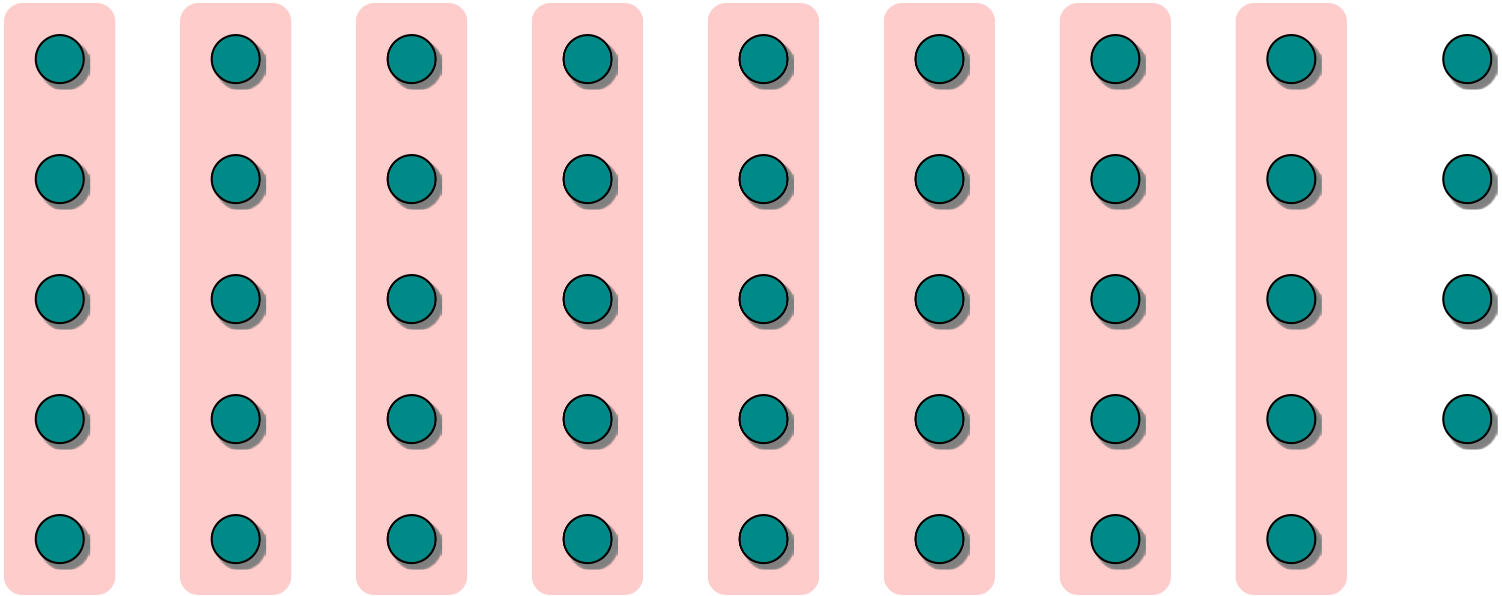
2	5	3	6	8	13	10	11	$k = 4$

Select the $7 - 4 = 3$rd smallest recursively.

Choosing the pivot

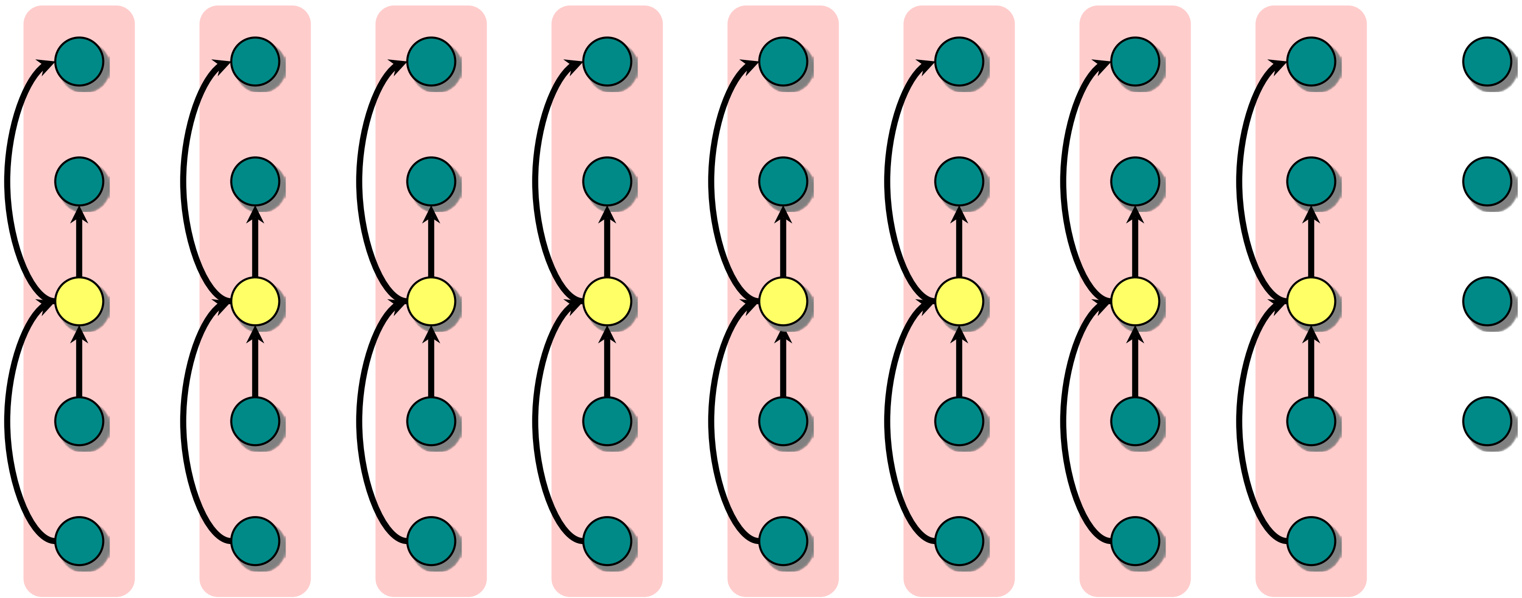


Choosing the pivot

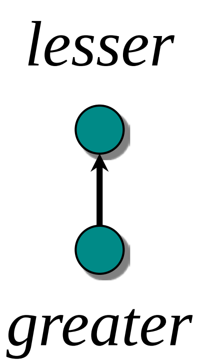


1. Divide the n elements into groups of 5.

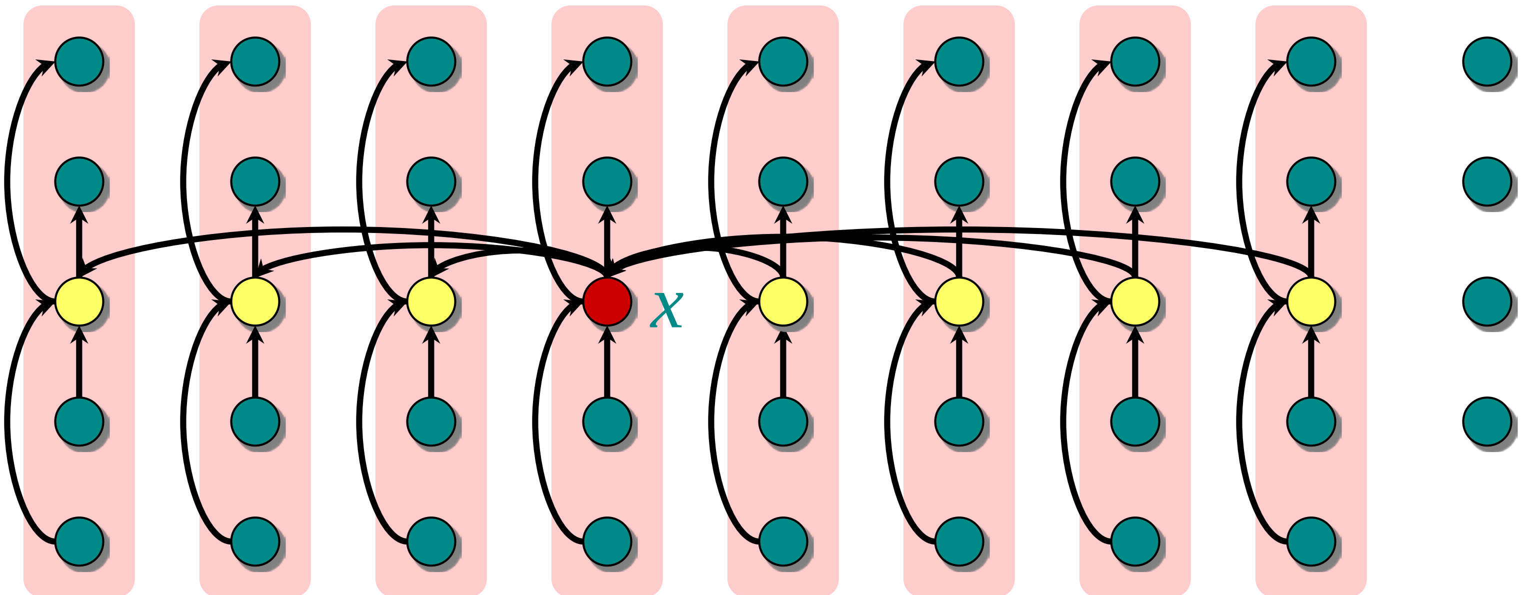
Choosing the pivot



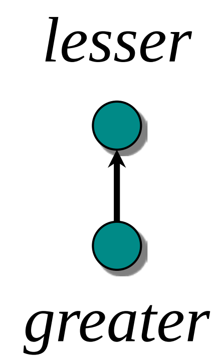
1. Divide the n elements into groups of 5. Find the median of each 5-element group.



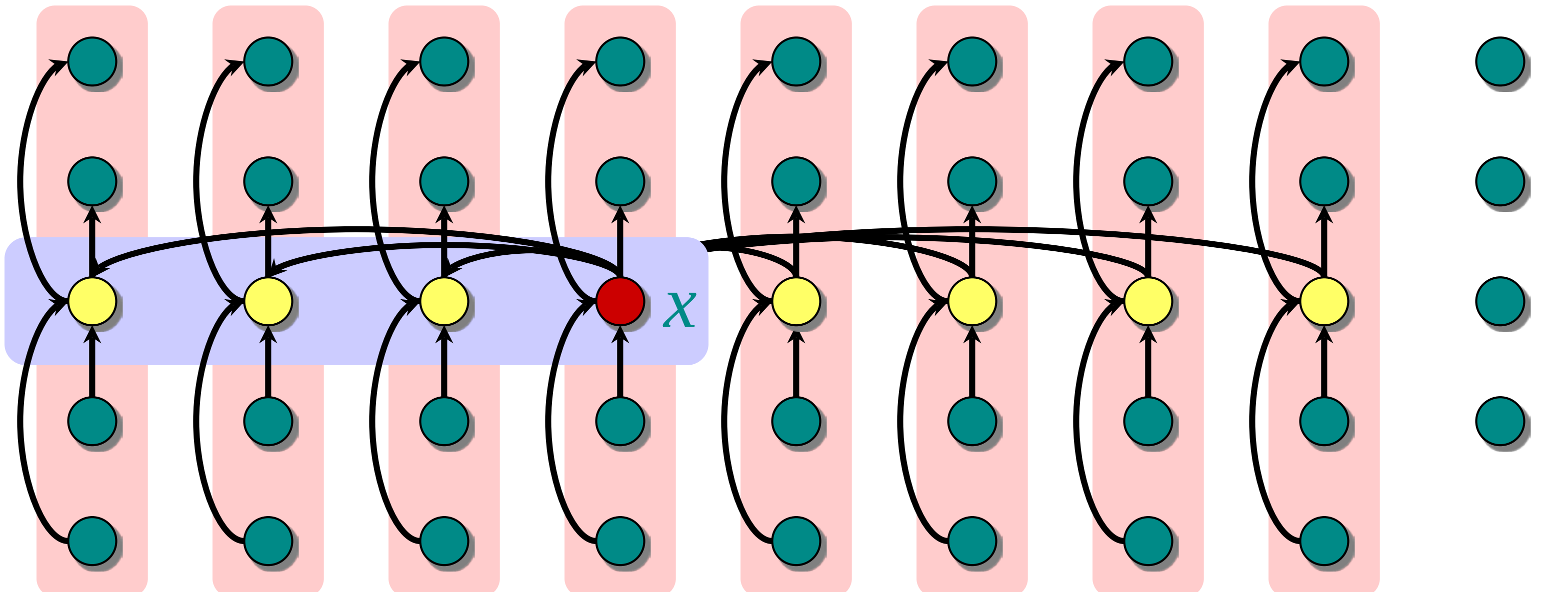
Choosing the pivot



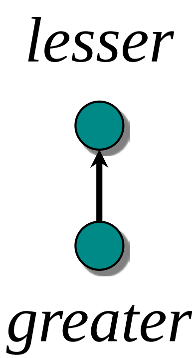
1. Divide the n elements into groups of 5. Find the median of each 5-element group.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.



Analysis

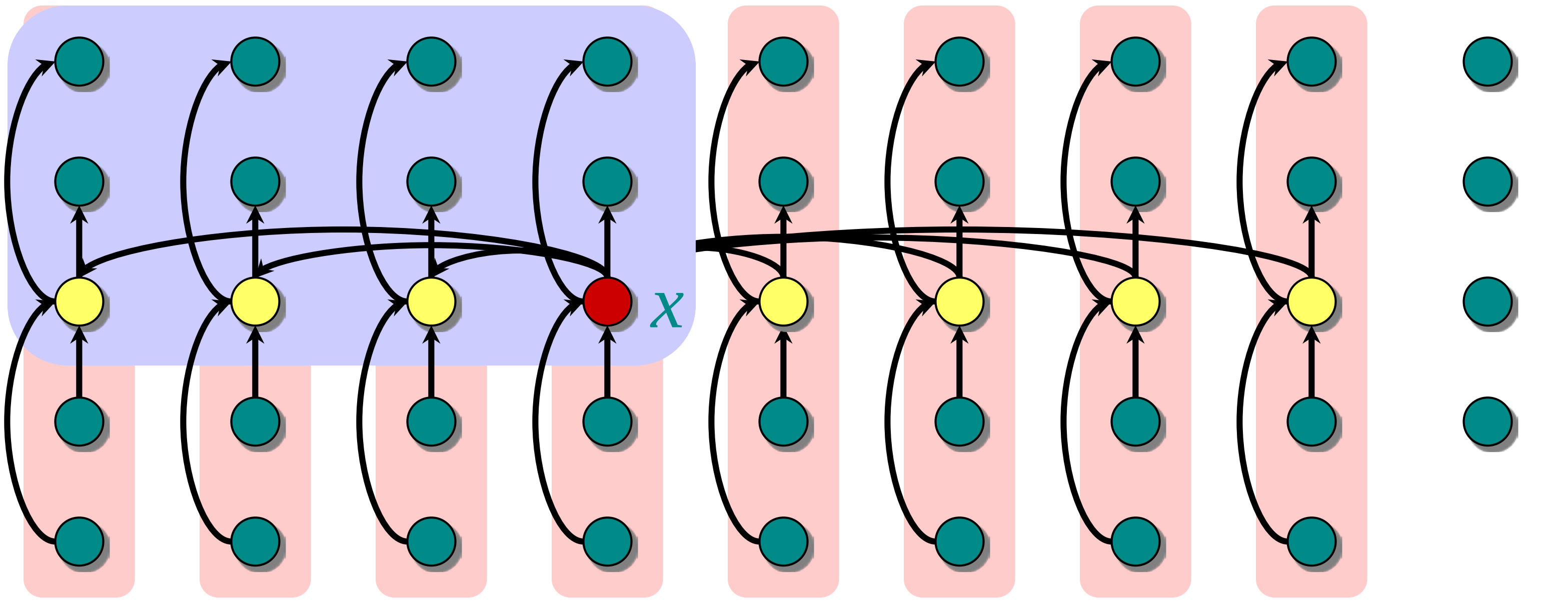


At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.



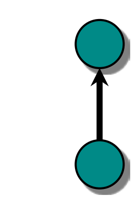
Analysis

(Assume all elements are distinct.)



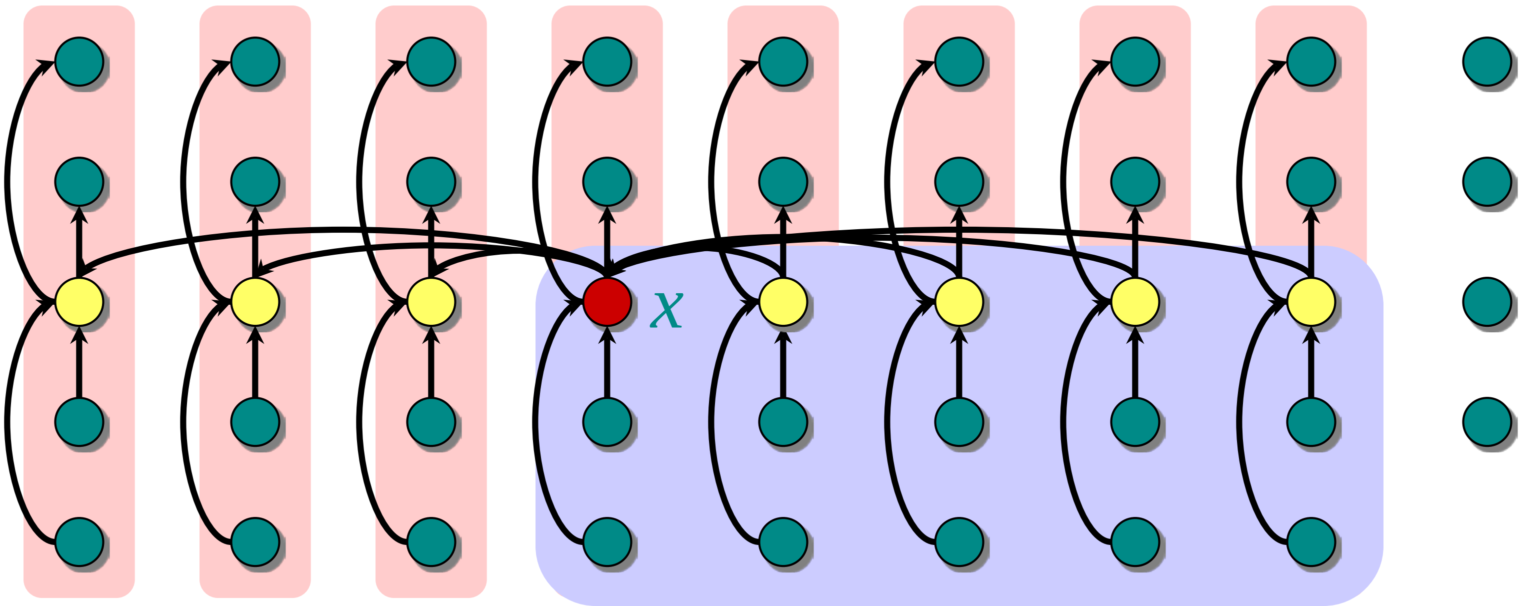
At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$.



Analysis

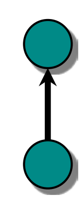
(Assume all elements are distinct.)



At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$.

lesser



greater

Developing the recurrence

$T(n)$ — SELECT(i, n)

$\Theta(n)$ — 1. Divide the n elements into groups of 5. Find the median of each 5-element group.

$T(n/5)$ — 2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

$\Theta(n)$ — 3. Partition around the pivot x. Let $k = \text{rank}(x)$.

$T(7n/10)$ — 4. **if** $i = k$ **then return** x
elseif $i < k$
then recursively SELECT the ith smallest element in the lower part
else recursively SELECT the $(i–k)$th smallest element in the upper part

Solving the recurrence

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

$T(n) \geq cn$

Recursion Tree: $T(n) \leq cn\left(1 + \frac{9}{10} + \left(\frac{9}{10}\right)^2 + \dots\right)$

$$= cn\frac{1}{1-\frac{9}{10}} = O(n)$$

T(n)=$\Theta(n)$

Conclusion

- In practice, this algorithm runs slowly, because the constant in front of n is large.
- There is a randomized algorithm that runs in expected linear time.
- The randomized algorithm is far more practical.

Exercise: *Why not divide into groups of* 3*?*