Divide and Conquer 2

Outline

- Presorting
  - Counting Inversions
  - Closest Pair
- Reduce the number of subproblems
  - Multiplication of Large Integers
  - Strassen’s Matrix Multiplication
- Some examples
  - Quickhull
  - Selection Problem
  - Maximum Subarray
  - Defective Chessboard Problem

Counting Inversions: problem

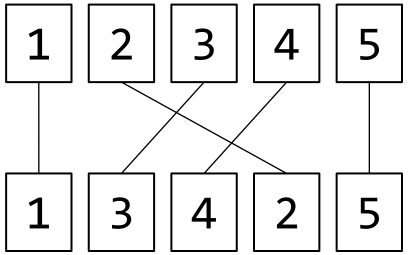
- Given an integer array nums, return the number of inversions in the array
- An inversion is a pair (nums[i], nums[j]) or (i, j) where:
  - 0 <= i < j < nums.length and
  - nums[i] > nums[j]
- Application: Similarity metric - number of inversions between two rankings, e.g., match your song preferences with others

Songs

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

<u>Inversions</u>

3-2, 4-2



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

Counting Inversions: divide and conquer

- Divide and Conquer Steps:
  - 1.**Divide** the problem into a number of smaller subproblems
  - 2.**Conquer** the subproblems by solving them recursively
  - 3.**Combine** the solutions to the subproblems to form the solution

- Consider:
  - Solution to the big problem =

    Solution to the left subproblem

    \+ Solution to the right subproblem

    \+ Solution to the part that crossing between left and right subproblems
  - If preprocessing like presorting can be helpful
  - (Usually) If the above 2 conditions hold, the merge sort approach can be used

Counting Inversions: intuition

- The pairs: left half, right half or (nums[i], nums[j]) in two halves separately
- Order helps:
  - 1,2,3,4,5: 0 inversions
  - 5,4,3,2,1: 4 + 3 + 2 + 1 = 10 inversions

left

13	8	10	6	15	18

right

12	20	9	14	17	19

$\{(13,12),(15,12),(18,12)\}$

3	?	?	?	?	?

Sort the left:

6	8	10	13	15	18

12	20	9	14	17	19

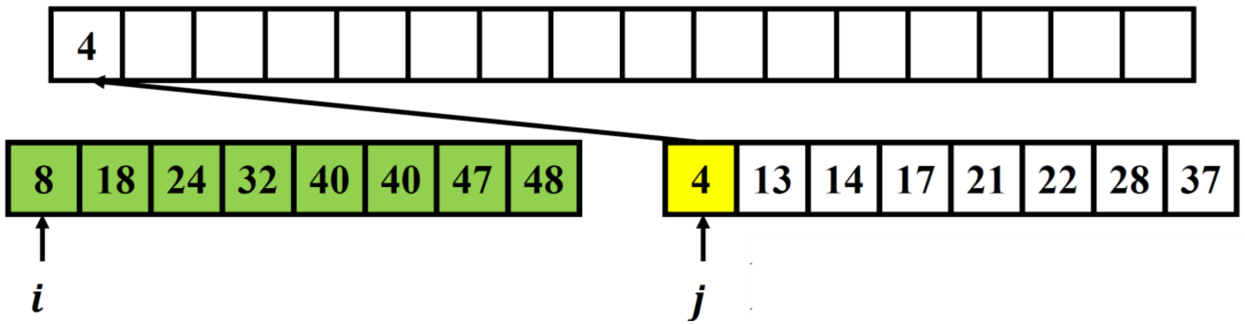
$\{(13,12),(15,12),(18,12)\}$

3	?	?	?	?	?

Sort then binary search?
O(nlogn)

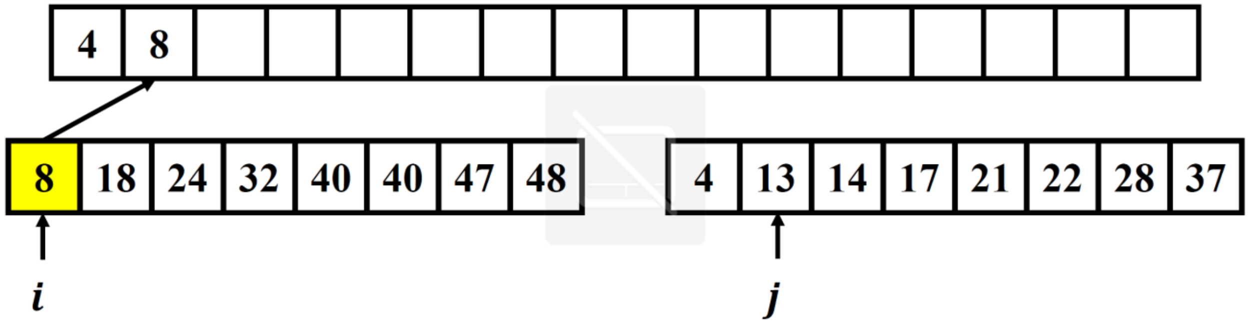
Counting Inversions: merge sort framework

- Integrate the sorting/counting into the combine step
- The left and right halves are counted before combine, thus no impact
- Use the merge sort framework



nums[j]<nums[i]

(nums[i] and right, nums[j]):
All are inversions



nums[i]<nums[j]

(nums[i], nums[j] and right):
All are not inversions

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	10	13	15	18

i=1

9	12	14	17	19	20

j=7

Inversion count:

?	?	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6											

6	8	10	13	15	18

i=1

9	12	14	17	19	20

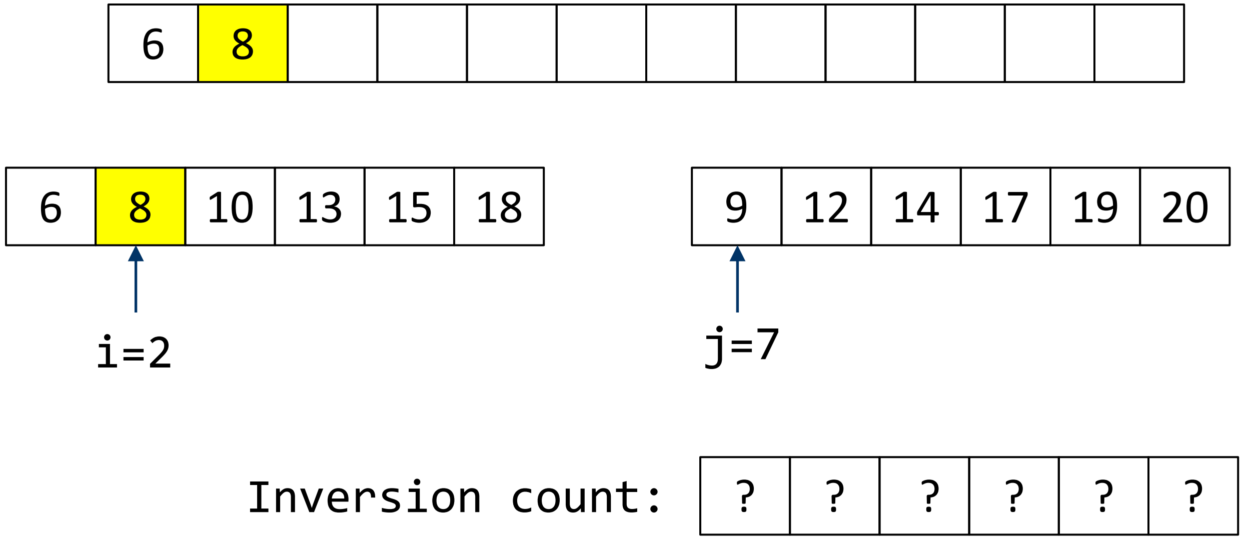
j=7

Inversion count:

?	?	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++



Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9									

6	8	10	13	15	18

i=3

9	12	14	17	19	20

j=7

Inversion count:

4	?	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10								

6	8	10	13	15	18

i=3

9	12	14	17	19	20

j=8

Inversion count:

4	?	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12							

6	8	10	13	15	18

i=4

9	12	14	17	19	20

j=8

Inversion count:

4	3	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13						

6	8	10	13	15	18

i=4

9	12	14	17	19	20

j=9

Inversion count:

4	3	?	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14					

6	8	10	13	15	18

i=5

9	12	14	17	19	20

j=9

Inversion count:

4	3	2	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15				

6	8	10	13	15	18

i=5

9	12	14	17	19	20

j=10

Inversion count:

4	3	2	?	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15	17			

6	8	10	13	15	18

i=6

9	12	14	17	19	20

j=10

Inversion count:

4	3	2	1	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15	17	18		

6	8	10	13	15	18

i=6

9	12	14	17	19	20

j=11

Inversion count:

4	3	2	1	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15	17	18	19	

6	8	10	13	15	18

9	12	14	17	19	20

j=11

Inversion count:

4	3	2	1	?	?

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15	17	18	19	20

6	8	10	13	15	18

9	12	14	17	19	20

j=12

Inversion count:

4	3	2	1	0	0

Counting Inversions: method

- Use the merge sort framework
- Scan the sorted subarrays from left to right, nums[i] in nums[1..mid], nums[j] in nums[mid+1..n]
  - If nums[i]>nums[j], count([i..mid]), j++
  - If nums[i]<=nums[j], i++

6	8	9	10	12	13	14	15	17	18	19	20

6	8	10	13	15	18

9	12	14	17	19	20

Inversion count:

4	3	2	1	0	0

Sum: S = 4+3+2+1 = 10

Counting Inversions: Merge-and-Count

MERGE-AND-COUNT(A,B)

Maintain a Current pointer into each list, initialized to point to the front elements

Maintain a variable Count for the number of inversions, initialized to 0

While both lists are nonempty:

Let a_i and b_j be the elements pointed to by the Current pointer

Append the smaller of these two to the output list

If b_j is the smaller element **then**

Increment Count by the number of elements remaining in A

Endif

Advance the Current pointer in the list from which the smaller element was selected.

EndWhile

Once one list is empty, append the remainder of the other list to the output

Return Count and the merged list

Counting Inversions: Sort-and-Count

- Input: List L.
- Output: Number of inversions in L and L in sorted order.

SORT-AND-COUNT(L)

IF (list L has one element)

    RETURN $(0, L)$.

Divide the list into two halves A and B.

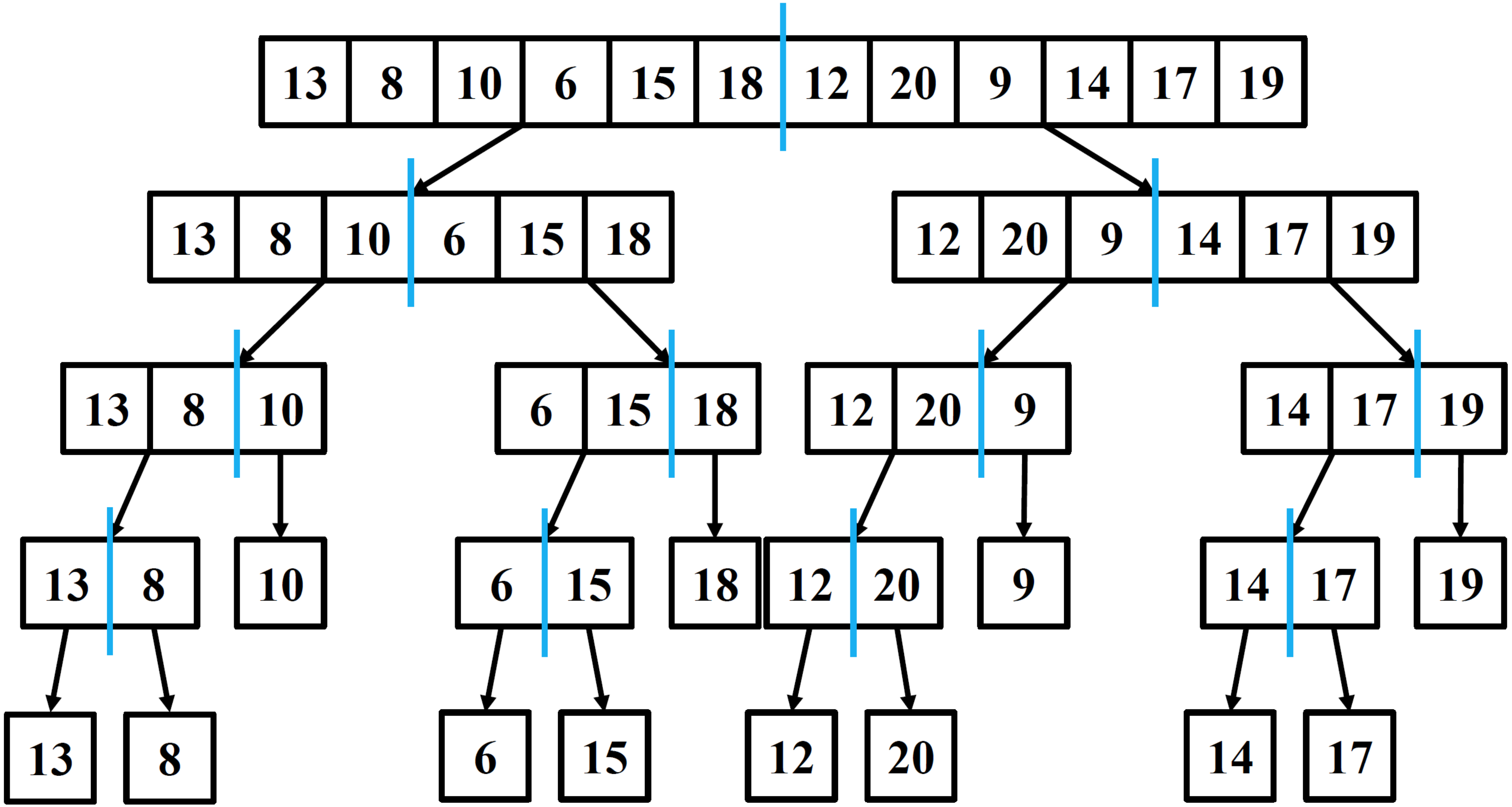
$(r_A,\ A) \leftarrow$ SORT-AND-COUNT(A). $\longleftarrow T(n / 2)$

$(r_B,\ B) \leftarrow$ SORT-AND-COUNT(B). $\longleftarrow T(n / 2)$

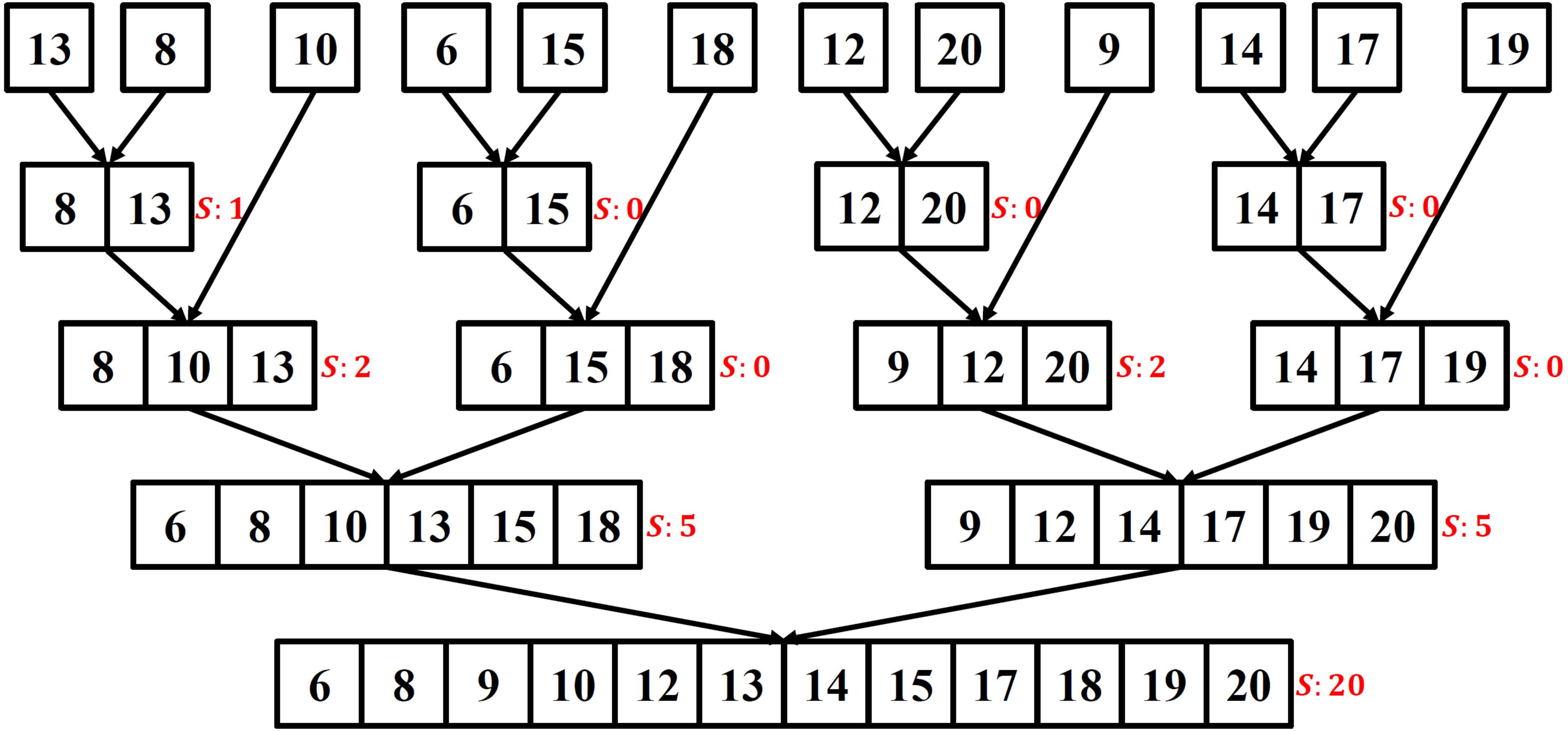
$(r_{AB},\ L) \leftarrow$ MERGE-AND-COUNT(A, B). $\longleftarrow \Theta(n)$

RETURN $(r_A + r_B + r_{AB},\ L)$.

Counting Inversions: divide



Counting Inversions: combine

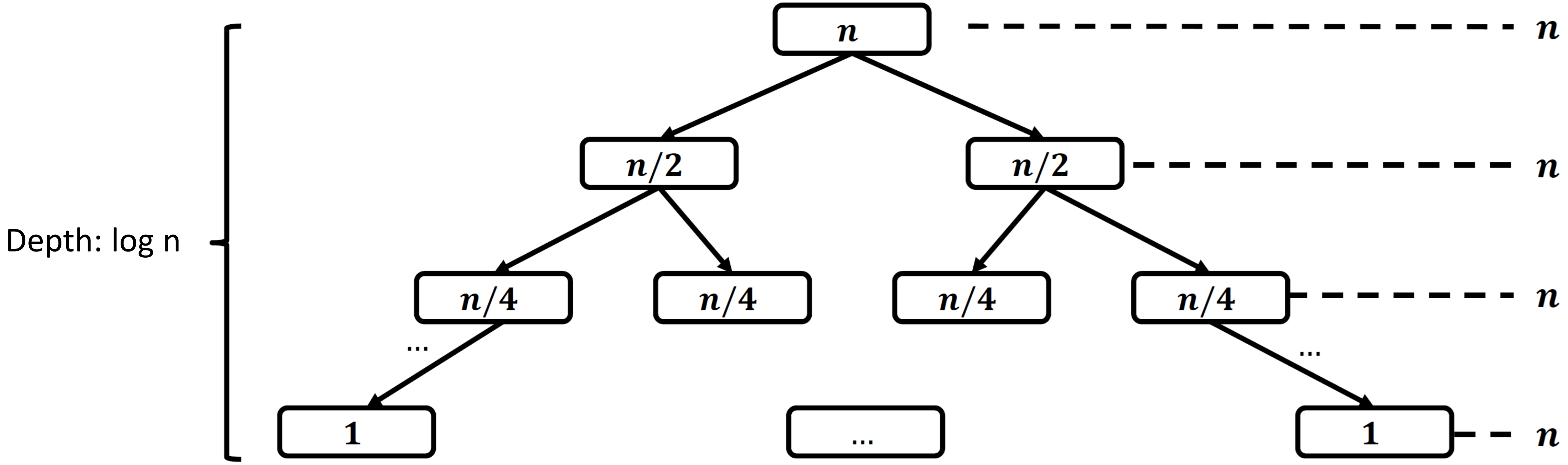


Counting Inversions: complexity

- Complexity

$$T(n) = \begin{cases} 1, & n = 1 \\ 2 \cdot T(n/2) + O(n), & n > 1 \end{cases}$$

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



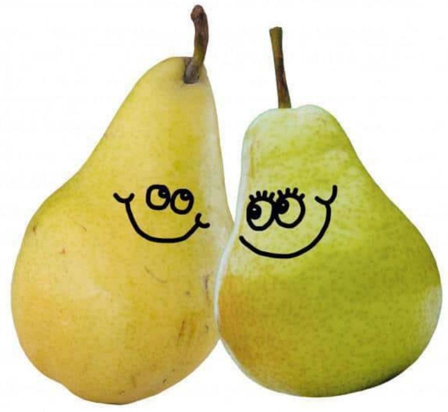
Counting Inversions: summary

- Brute force -> D&C + Sort/Search -> Marge sort framework
- Presorting often helps
- Integrating to the merge sort framework

- Exercise:
  - https://leetcode.com/problems/reverse-pairs/
  - https://leetcode.com/problems/count-of-smaller-numbers-after-self/

Closest Pair

- The nearest two points in a 2D plane.
- Remember the brute-force algorithm? How? Complexity?
- You may say not too bad, but it is frequently used -> **aggregated effect.**
  - i.e., graphics, computer vision, and molecular modeling.
- Target: an algorithm **faster than quadratic.**
- Define the problem:
  - Points $P = \{p_1, p_2, \cdots, p_n\}$ where $p_i = (x_i, y_i)$.
  - Distance $d(p_i, p_j)$ between point p_i and point p_j.
  - $\arg_{i,j} \min\{d(i,j) | 1 \leq i, j \leq n\}$.
- Assume, all x_i are different and all y_i are different.
  - Easy for analysis.
  - Do not worry, i.e., you can just add an epsilon like 10^{-12}.



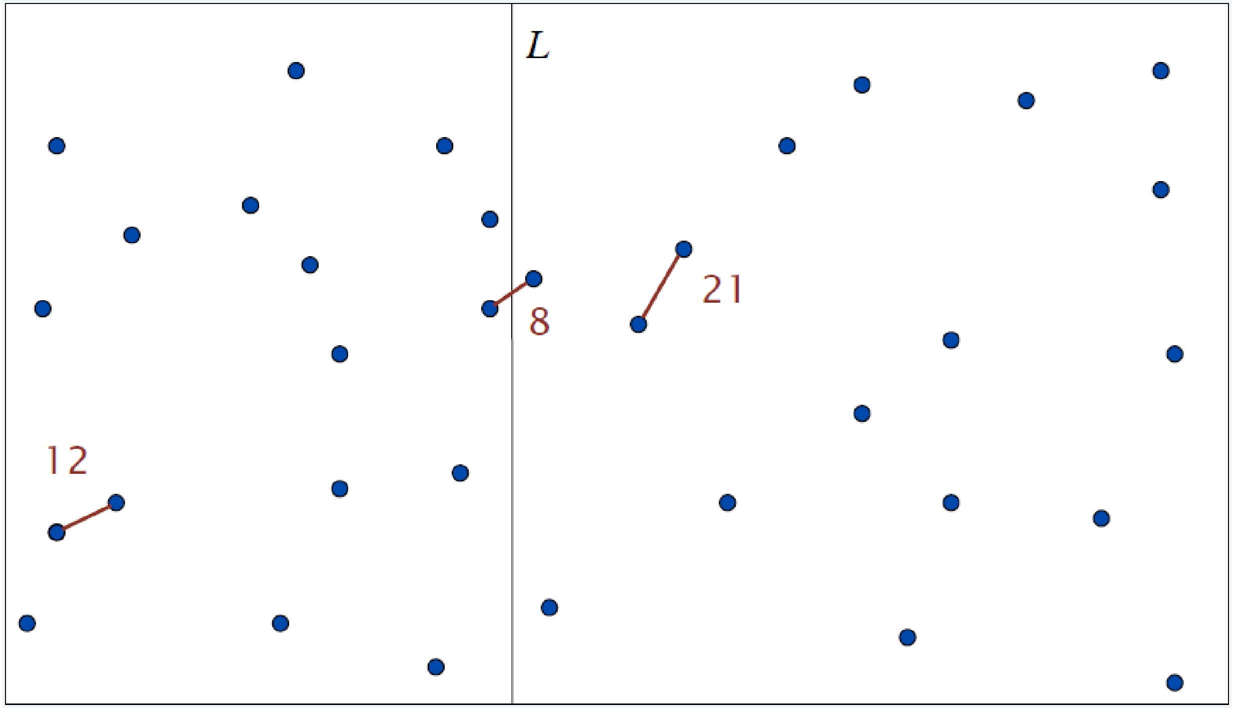
https://www.lovethesat.com/wp-content/uploads/2017/09/a-pair-of-pears.jpg

Closest Pair: solution

- Idea: divide and conquer.
- Think about the 1D case. 
- Three possible min distances after dividing the line:
  - Left min.
  - Right min.
  - Right-most point in the left part + the left-most point in the right part.
- Recursively we can find the global min in around $n \log n$ time.
- 2D would be similar

Closest Pair: divide and conquer

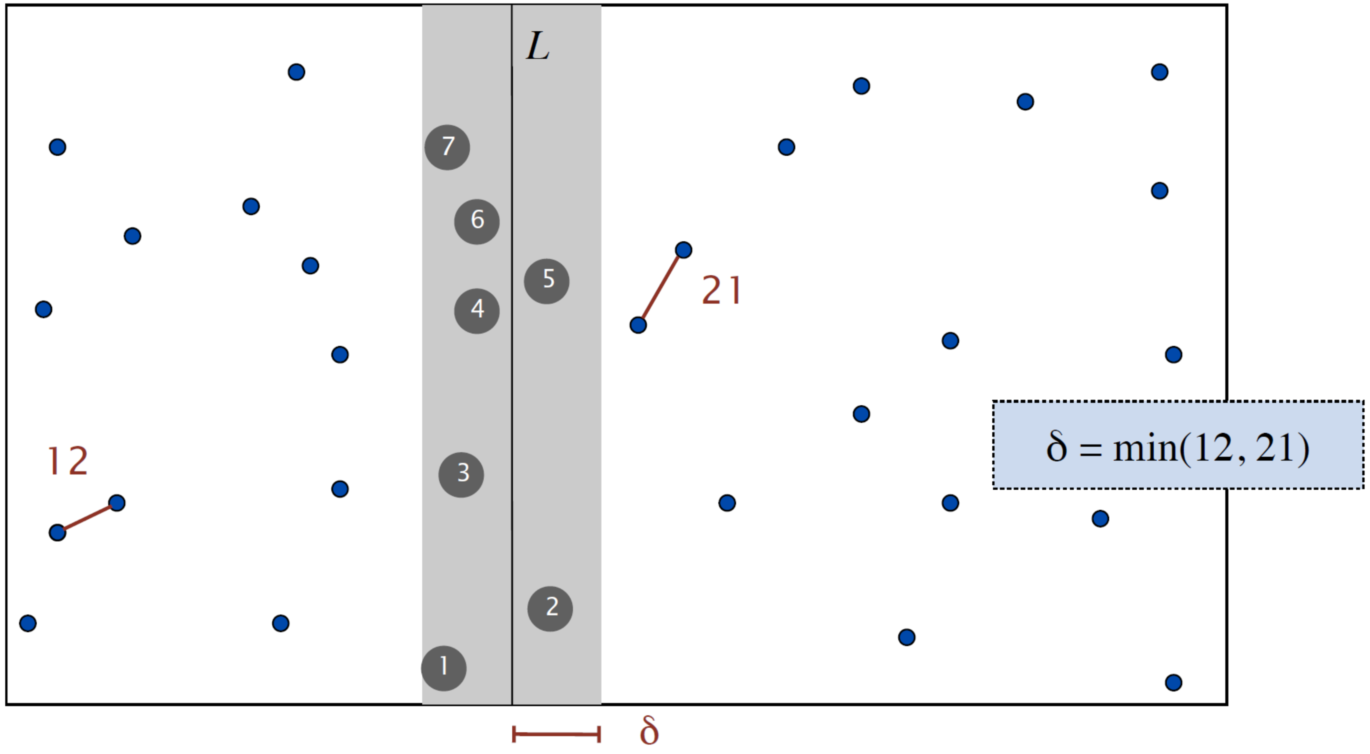
- Divide: draw vertical line L so that n / 2 points on each side, pre-sort x, P_x
- Conquer: find closest pair in each side recursively.
- Combine: find closest pair with one point in each side. <- seems like $\Theta(n^2)$
- Return best of 3 solutions.



Closest Pair: divide and conquer

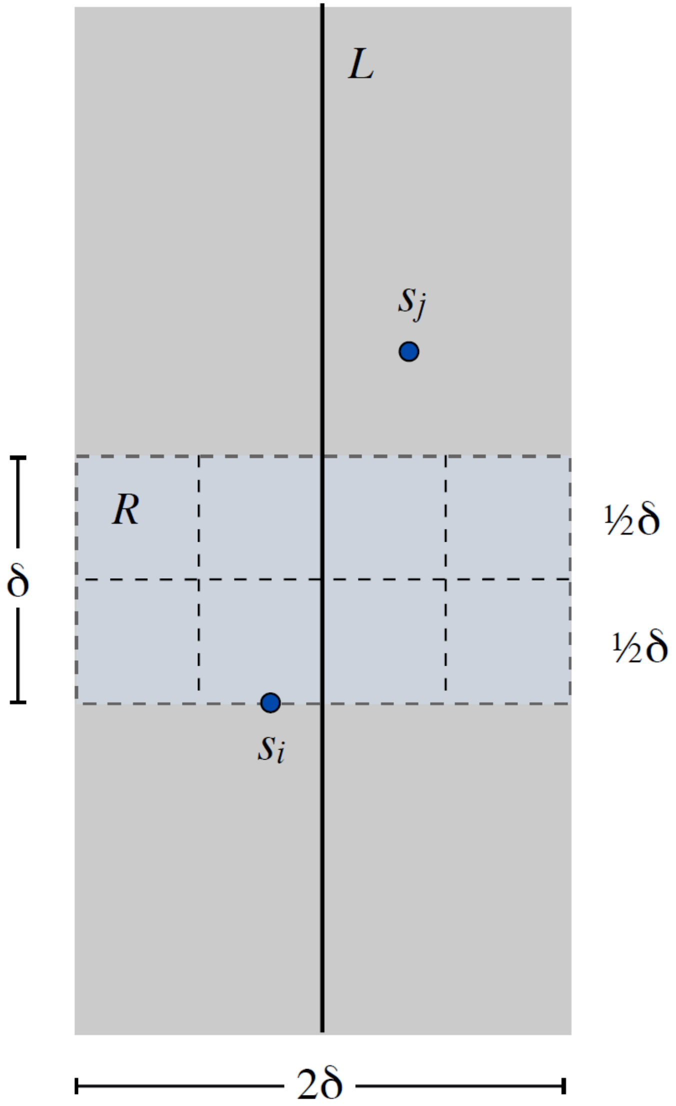
- Find closest pair with one point in each side, assuming that distance $<\delta$.
- Observation: suffices to consider only those points within δ of line L.
- Sort points in 2δ-strip by their y-coordinate. (Pre-sort y P_y or using mergsort)
- Check distances of only those points within **7** positions in sorted list!

Closest pair in left Q is (q_0^*, q_1^*): 12
Similarly, (r_0^*, r_1^*) for right R :21
$d(q_0^*, q_1^*)$ and $d(r_0^*, r_1^*)$ -> δ is the smaller one of the two: 12



Closest Pair: divide and conquer

- **Def.** Let s_i be the point in the 2 δ-strip, with the *ith* smallest y-coordinate
- **Claim.** If $|j - i| > 7$, then the distance between s_i and s_j is at least δ.
- **Proof.**
  - Consider the 2 δ-by-δ rectangle *R* in strip
  - whose min y-coordinate is y-coordinate of s_i .
  - Distance between s_i and any point s_j above *R* is $\geq \delta$.
  - Subdivide *R* into 8 squares.
  - At most 1 point per square, diameter is $\delta/\sqrt{2} < \delta$
  - At most **7 other** points can be in *R*. (constant can be improved with more refined geometric packing argument)



Closest Pair: Algorithm and Summary

SINGAPORE INSTITUTE OF TECHNOLOGY

- **Theorem 1:** The algorithm **correctly** outputs a closest pair of points in P.
- Proof: just organize the cases.
  - Leaf node case: limited points, like ≤ 3.
  - Min distance in Q.
  - Min distance in R.
  - Min distance across Q and R.
    - With the 7 boxes property.
  - All cases captured.
- **Theorem 2:** The time complexity is $\boldsymbol{O(n \log n)}$.
- Proof: just do it operation by operation.
  - Initial sorting $O(n \log n)$.
  - Each iteration $O(n)$.
  - # iter $O(\log n)$, i.e., half, half, half ....

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{if } n > 1 \end{cases}$$

```
Closest-Pair(P)
  Construct Px and Py  (O(n log n) time)   <- Can only sort x
  (p0*, p1*) = Closest-Pair-Rec(Px,Py)

Closest-Pair-Rec(Px, Py)
  If |P| ≤ 3 then
    find closest pair by measuring all pairwise distances
  Endif   <- Sort the 3 by y here (if not pre-sorted by y)

  Construct Qx, Qy, Rx, Ry (O(n) time)
  (q0*,q1*) = Closest-Pair-Rec(Qx, Qy)
  (r0*,r1*) = Closest-Pair-Rec(Rx, Ry)

  δ = min(d(q0*,q1*), d(r0*,r1*))   <- Merge Q and R by y (if not pre-sorted by y)
  x* = maximum x-coordinate of a point in set Q
  L = {(x,y) : x = x*}
  S = points in P within distance δ of L.

  Construct Sy (O(n) time)
  For each point s ∈ Sy, compute distance from s
     to each of next 7 points in Sy
     Let s, s' be pair achieving minimum of these distances
     (O(n) time)

  If d(s,s') < δ then
     Return (s,s')
  Else if d(q0*,q1*) < d(r0*,r1*) then
     Return (q0*,q1*)
```

Algorithm Design, Jon Kleinberg and Eva Tardos (ISBN: 9780321372918)

Divide and Conquer: another improvement

- Sorting can be helpful, e.g., Counting Inversions, Closest Pair
- Examine the divide and conquer time complexity function:

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

- a: number of subproblems, n/b: size of subproblems, $f(n)$: divide and combine work
- If a is relatively big, b is small, $f(n)$ is small, the solution is:

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

- Idea: reduce a to reduce $T(n)$

- By combining the results of other subproblems (e.g., using addition or subtraction), the results of certain subproblems can be derived

Integer multiplication

SINGAPORE INSTITUTE OF TECHNOLOGY

- Multiplication. Given two n-bit integers a and b, compute a × b.
- Grade-school algorithm (long multiplication), $\Theta(n^2)$ bit operations.

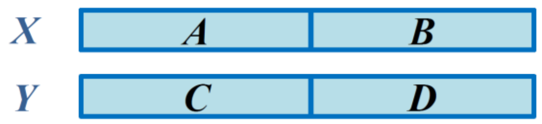
long multiplication

								1	1	0	1	0	1	0	1
							×	0	1	1	1	1	1	0	1
								1	1	0	1	0	1	0	1
							0	0	0	0	0	0	0	0	
						1	1	0	1	0	1	0	1		
					1	1	0	1	0	1	0	1			
				1	1	0	1	0	1	0	1				
			1	1	0	1	0	1	0	1					
		1	1	0	1	0	1	0	1						
	0	0	0	0	0	0	0	0							
0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1

Simple D&C:
X: A(1101) B (0101)
Y: C(0111) D (1101)

$$X = A2^{n/2} + B, \quad Y = C2^{n/2} + D.$$

$$XY = \underline{AC}\, 2^n + (\underline{AD} + \underline{BC})\, 2^{n/2} + \underline{BD}$$



$$T(n) = 4T(n/2) + O(n) \rightarrow T(n) = O(n^2)$$

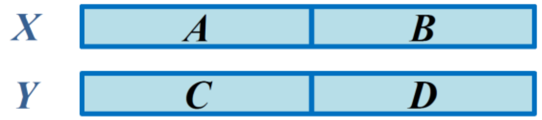
Integer multiplication

SINGAPORE INSTITUTE OF TECHNOLOGY

- Karatsuba trick, reduce the number of subproblems using Arithmetic computation

$$X = A2^{n/2} + B, \quad Y = C2^{n/2} + D.$$

$$XY = \underline{AC}\, 2^n + (\underline{AD} + \underline{BC})\, 2^{n/2} + \underline{BD}$$



$$AD + BC = \underline{(A - B)(D - C)} + \underline{AC} + \underline{BD}$$

- Complexity: $T(n) = 3T(n/2) + cn,$

  $T(1) = 1$

- Solution: $T(n) = O(n^{\log_2^3}) = O(n^{1.59})$

year	algorithm	bit operations
12xx	grade school	$O(n^2)$
1962	Karatsuba–Ofman	$O(n^{1.585})$
1963	Toom-3, Toom-4	$O(n^{1.465})$, $O(n^{1.404})$
1966	Toom–Cook	$O(n^{1+\varepsilon})$
1971	Schönhage–Strassen	$O(n \log n \cdot \log \log n)$
2007	Fürer	$n \log n\, 2^{O(\log^* n)}$
2019	Harvey–van der Hoeven	$O(n \log n)$
	???	$O(n)$

number of bit operations to multiply two n-bit integers

Matrix multiplication

- Matrix multiplication: Given two n-by-n matrices A and B, compute C = AB
- Grade-school. $\Theta(n^3)$ arithmetic operations: $\Theta(n^2)$ dot products with $\Theta(n)$ arithmetic operations each
- Simple D&C:

$$\begin{pmatrix} \boxed{A_{11} \quad A_{12}} \\ A_{21} \quad A_{22} \end{pmatrix} \begin{pmatrix} \boxed{\begin{matrix} B_{11} \\ B_{21} \end{matrix}} & \begin{matrix} B_{12} \\ B_{22} \end{matrix} \end{pmatrix} = \begin{pmatrix} \boxed{C_{11}} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

, where

$$C_{11} = \underline{A_{11}B_{11}} + \underline{A_{12}B_{21}} \qquad C_{12} = \underline{A_{11}B_{12}} + \underline{A_{12}B_{22}}$$
$$C_{21} = \underline{A_{21}B_{11}} + \underline{A_{22}B_{21}} \qquad C_{22} = \underline{A_{21}B_{12}} + \underline{A_{22}B_{22}}$$

- $T(n) = 8\,T(n/2) + cn^2$, $\quad T(1) = 1$
- Solution: $T(n) = O(n^3)$

Strassen Matrix multiplication

- Define 7 subproblems: $M_1, M_2, M_3 \ldots M_7$:

$$\begin{pmatrix} \boxed{A_{11} \quad A_{12}} \\ A_{21} \quad A_{22} \end{pmatrix} \begin{pmatrix} \boxed{\begin{matrix} B_{11} \\ B_{21} \end{matrix}} & \begin{matrix} B_{12} \\ B_{22} \end{matrix} \end{pmatrix} = \begin{pmatrix} \boxed{C_{11}} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

$$C_{11} = \underline{A_{11}B_{11}} + \underline{A_{12}B_{21}} \quad C_{12} = \underline{A_{11}B_{12}} + \underline{A_{12}B_{22}}$$

$$C_{21} = \underline{A_{21}B_{11}} + \underline{A_{22}B_{21}} \quad C_{22} = \underline{A_{21}B_{12}} + \underline{A_{22}B_{22}}$$

$$M_1 = A_{11}(B_{12} - B_{22})$$
$$M_2 = (A_{11} + A_{12})B_{22}$$
$$M_3 = (A_{21} + A_{22})B_{11}$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

$$C_{11} = M_5 + M_4 - M_2 + M_6$$
$$C_{12} = M_1 + M_2$$
$$C_{21} = M_3 + M_4$$
$$C_{22} = M_5 + M_1 - M_3 - M_7$$

$T(n) = 7\,T(n/2) + 18(n/2)^2, \quad T(1) = 1$

Solution: $T(n) = O(n^{\log 7}) = O(n^{2.8075})$

Coppersmith–Winograd: $O(n^{2.376})$

[Williams, Xu, Xu, and Zhou]: $O(n^{2.371552})$

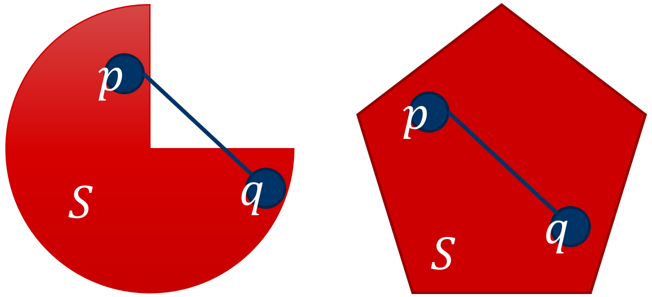
Another improvement: Summary

- By combining the results of other subproblems (e.g., using addition or subtraction), the results of certain subproblems can be derived

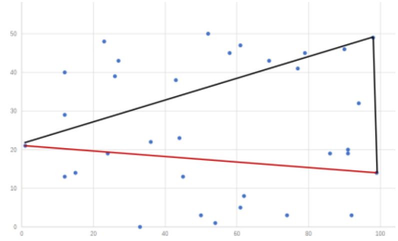
- Exercise:
  - https://leetcode.com/problems/multiply-strings/

Convex Hull

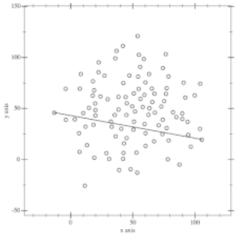
- Definition: A set S is **convex** if for any $p, q \in S$, line segment $\overline{pq} \in S$.
- Remember our brute-force algorithm?
  - **Enumerate all** $O(n^2)$ **lines** (fixed by a pair of pts).
  - For each line, **all the points** are in the **same side**.
    - Verification cost $O(n)$ for n points.
  - Overall: $O(n^2 * n) = O(n^3)$.



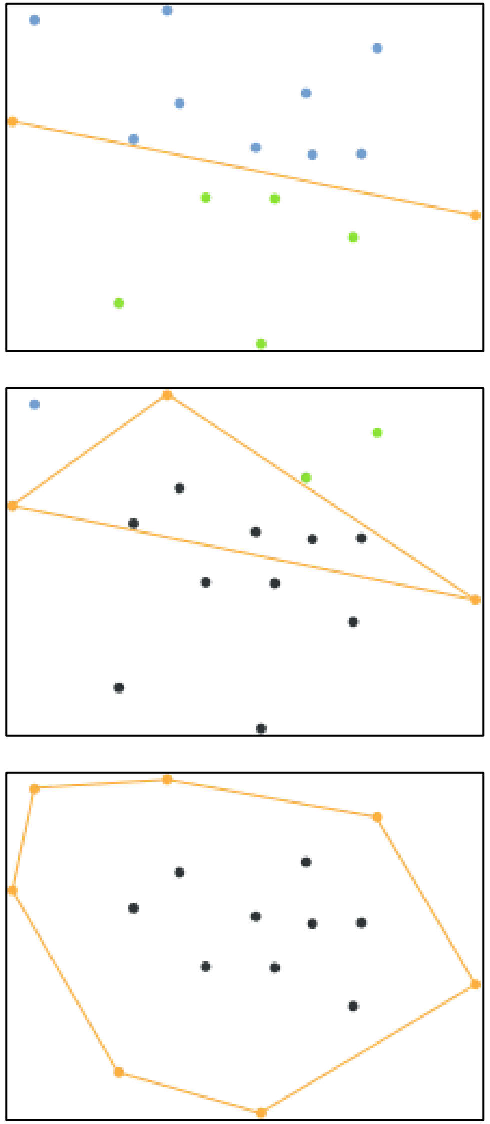
- Target: something faster with lower complexity like $O(n^2)$.
- **QuickHull** algorithm: similar to quick sort.
- Observation 1: given any direction in a 2D plane, the **farthest point** towards the direction must be **part of the convex hull**.
- Observation 2: given any convex shape, the points inside are not part of the convex hull. (Think about a triangle.)



QuickHull Idea



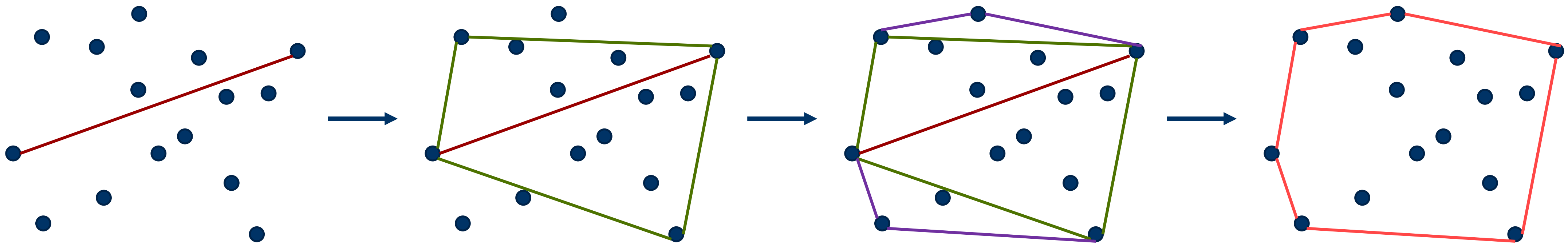
- Find the points with minimum and maximum x coordinates, as these will always be part of the convex hull. If many points with the same minimum/maximum x exist, use the ones with the minimum/maximum y, respectively.
- Use the line formed by the two points to divide the set into two subsets of points, which will be processed recursively. We next describe how to determine the part of the hull above the line; the part of the hull below the line can be determined similarly.
- Determine the point above the line with the maximum distance from the line. This point forms a triangle with the two points on the line.
- The points lying inside of that triangle cannot be part of the convex hull and can therefore be ignored in the next steps.
- Recursively repeat the previous two steps on the two lines formed by the two new sides of the triangle.
  - **Subproblem** 1: **left** side of the triangular.
  - **Subproblem** 2: **right** side of the triangular.
- Continue until no more points are left, the recursion has come to an end and the points selected constitute the convex hull.



QuickHull Algorithm

- QuickHull(S){
  - $a \leftarrow$ the left-most point.
  - $b \leftarrow$ the right-most point.
  - $S_L \leftarrow$ all the points above $\overline{ab}$.
  - $S_R \leftarrow$ all the points below $\overline{ab}$.
  - $H_L \leftarrow$ QuickHalfHull($S_L, \overline{ab}$).
  - $H_R \leftarrow$ QuickHalfHull($S_R, \overline{ab}$).
  - Return: $H_L \cup H_R \cup \{a, b\}$ convex hull.

- QuickHalfHull($S, \overline{ab}$){
  - Assert $S \neq \Phi$.
  - $c \leftarrow$ the farthest point above $\overline{ab}$.
  - $S_L \leftarrow$ all points in the left of $\overline{ac}$.
  - $S_R \leftarrow$ all points in the right of $\overline{bc}$.
  - $H_L \leftarrow$ QuickHalfHull($S_L, \overline{ac}$).
  - $H_R \leftarrow$ QuickHalfHull($S_R, \overline{bc}$).
  - Return: $H_L \cup H_R \cup \{c\}$.



Complexity

- What is the complexity of QuickHull?
  - Hint: QuickSort. Any similarity?
- **Worst-case** complexity: $\boldsymbol{O(n^2)}$.
  - Extreme case: all the points on a **circle**.
  - Every iteration, **no pt inside** the **triangular**.
  - All the points on the convex hull.
  - Each call finds one pt, or the farthest point.
  - $O(n)$ iter, each using $O(n)$ to find the farthest point.
- **In practice**: $\boldsymbol{n \log n}$.
  - Think about our proof for **QuickSort**.
  - Assume say 50% of iter we can divide quite evenly.
  - Define “quite evenly” as 25%-75% of the remaining pts.
  - Will get some log function w/ a constant base like 4/3.
  - Do not care about the constants, so overall $O(n \log n)$ with $O(\log n)$ iter.
- Practice: try to find a convex hull of some objs, i.e., mask or cat.

https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/; https://www.geekgirlauthority.com/

QuickHull: summary

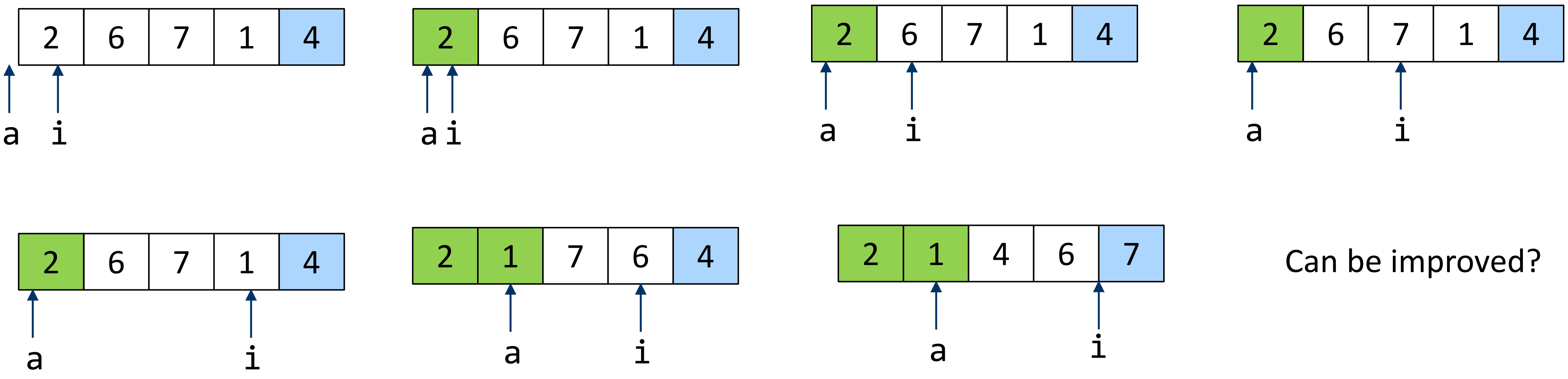
- Similar to quicksort
- Graham’s/Andrew's Convex Hull Algorithm can achieve a worst-case running time *O(nlogn)*
- Gift Wrapping Algorithm: O(nh), h is the number of points on the convex hull
- Many Graphics/Computational Geometry problems can be solved using divide and conquer, e.g., FFT. The complexity to solve some of those problems in 2D

problem	brute	clever
closest pair	$O(n^2)$	$O(n \log n)$
farthest pair	$O(n^2)$	$O(n \log n)$
convex hull	$O(n^2)$	$O(n \log n)$
Delaunay/Voronoi	$O(n^4)$	$O(n \log n)$
Euclidean MST	$O(n^2)$	$O(n \log n)$

- Exercise:
  - https://leetcode.com/problems/erect-the-fence/

More on partition

- Lomuto partition, similar to the Hoare partition introduced before
- Choose a pivot (first, last, random,…) p, separate to 2: <=p and >p
- Trace and obtain the right index (**a**) of the region <=p, swap A[**a+1**]and p
- Process: compare $A[i]$ with pivot p (4, last in the example)
  - if $A[i]$<= p, swap $A[i]$ and A[**a+1**], **a++**, i++
  - if $A[i]$>p, i++



Can be improved?

More on partition

- In the partitioning step of conventional quicksort(l,r), only one element ≤ p is positioned at a time, even if multiple elements have the same value as p
- For the case of elements < p, == p, and > p, how about handling all elements == p together?

- Dutch National Flag problem or 3-way partitioning
  - Trace and obtain the left and right indices **a, b** of the region ==p
  - (recur in the <p and >p region for sorting)
    - quicksort(l,a-1); quicksort(b+1,r);
- Process: while(i<=**b**)
  - compare $A[i]$ with pivot p
  - if $A[i]$<p, swap $A[i]$ and A[**a**], **a**++, i++
  - if $A[i]$>p, swap $A[i]$ and A[**b**], **b**--, i no change
  - else $A[i]$==p,i++

Selection Problem

- Selection: Given n elements from a totally ordered universe, find kth smallest
  - Minimum: k = 1; maximum: k = n
  - Median: k = floor((n + 1) / 2)
  - O(n) compares for min or max
  - O(n log n) compares by sorting
  - O(n log k) compares with a binary heap (max heap with k smallest)
  - Many applications, e.g., “top k”…

- Selection is easier than sorting, can we do it with O(n) compares?

Selection Problem

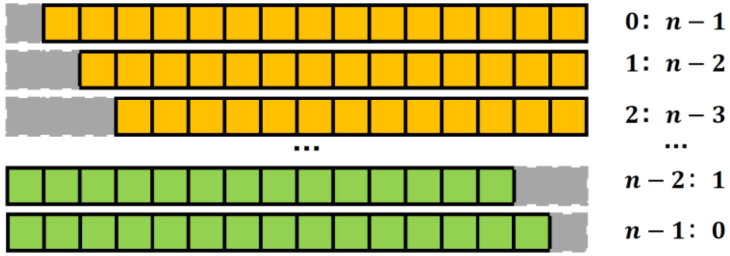
- Pick a random pivot element p in A
- Hoare/Lomuto/3-way partition the array into
  - Hoare/Lomuto: L (<=p, $|L|$=a) and R (>p)
  - 3-way: L (<p, $|L|$=a-1), M(==p, ($|L|+|M|$)=b), and R (>p)
- Recur in **one** subarray – the one containing the **k** th smallest element
- QUICK-SELECT(A, k)

  Pick pivot p A uniformly at random
  (L, R) = PARTITION-2-WAY(A, p). // $\Theta(n)$
  IF ($k <|L|$) RETURN QUICK-SELECT(L, k). // $T(i)$
  ELSE IF ($k > |L|$) RETURN QUICK-SELECT(R, $k - |L|$) //in R $T(n - i - 1)$
  ELSE IF ($k = |L|$) RETURN p.
  or
  (L, M, R) = PARTITION-3-WAY(A, p). // $\Theta(n)$
  IF ($k \leq |L|$) RETURN QUICK-SELECT(L, k). // $T(i)$
  ELSE IF ($k > |L| + |M|$) RETURN QUICK-SELECT(R, $k - |L| - |M|$) // $T(n - i - 1)$
  ELSE RETURN p. // IF (k = | L |+1), p is in M, so the |L|+1th
- It is also like binary search by computing p and finding p = = k in one subarray

Complexity

SINGAPORE INSTITUTE OF TECHNOLOGY

- Worse case: $\boldsymbol{O(n^2)}$, Best case: $\boldsymbol{O(n)}$,
- Randomized, so focus on the expected number of comparisons $\boldsymbol{O(n)}$ in expectation



$$T(n) \le \begin{cases} T(n-1) + O(n) \\ T(n-2) + O(n) \\ T(n-3) + O(n) \\ \quad\cdots \\ T(n-1) + O(n) \end{cases}$$

- Can assume we always recur of larger of two subarrays since T(n) is monotone non-decreasing
- Each has 1/n chance to occur, and occur twice

$$E[T(n)] \le E\left[\frac{2}{n} \cdot \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} \left(T(i) + O(n)\right)\right]$$

$$\le \frac{2}{n} \cdot \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} E[T(i)] + \frac{2}{n} \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} O(n)$$

$$\le \frac{2}{n} \cdot \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} E[T(i)] + O(n)$$

Assume
$\forall i < n,\ E[T(i)] \le c \cdot i$

$$E[T(n)] \le O(n) + \frac{2}{n} \cdot \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} E[T(i)]$$

$$\le O(n) + \frac{2}{n} \cdot \sum_{i=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} c \cdot i$$

$$\le O(n) + \frac{2}{n} \cdot c \cdot \frac{3}{8} n^2$$

$$= c \cdot n - \left(\frac{1}{4} c \cdot n - O(n)\right)$$

$$\le c \cdot n$$

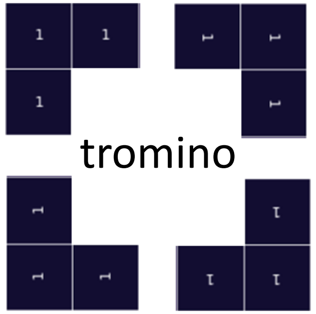
Choose $\boldsymbol{c}$ s.t., asymptoticly 1/4cn is greater than O(n)

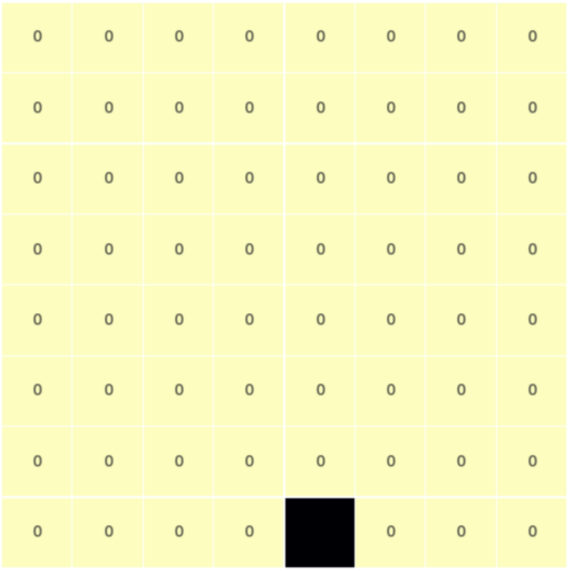
Selection: summary

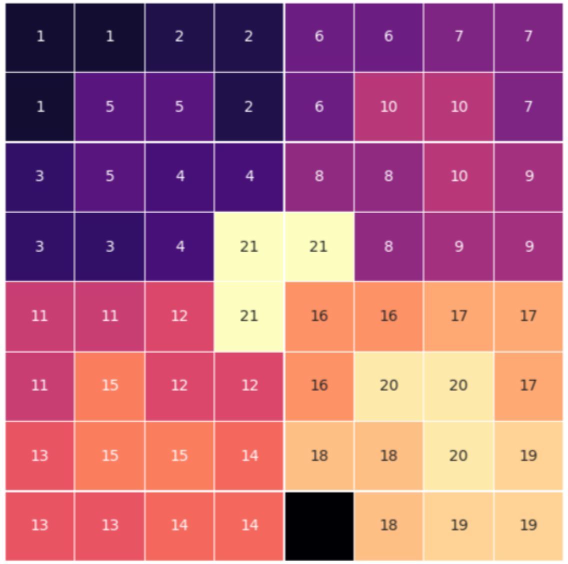
- Similar to quicksort
- Dutch National Flag Partition: < p, == p, and > p
- Median-of-medians (BFPRT) can achieve a worst-case running time O(n)
- Exercise:
  - https://leetcode.com/problems/kth-largest-element-in-an-array/
  - https://leetcode.com/problems/sort-an-array/ you may want to practice quicksort

Defective Chessboard problem

- Input: A n by n square board, with one of the 1 by 1 square missing, where $n = 2^k$ for some $k \geq 1$.
- Output: A tiling of the board using a tromino, a three square tile obtained by deleting the upper right 1 by 1 corner from a 2 by 2 square.
- You are allowed to rotate the tromino, for tiling the board.
- Subproblems needs to have the format, how to divide?







Defective Chessboard problem

Input Parameters: n, a power of 2 (the board size); the location L of the missing square

tile(n,L) {

    if (n == 2) {

    // base case the board is a right tromino T

    tile with T

    return

    }

    divide the board into four $n/2 \times n/2$ subboards

    place one tromino in the center (no.21) as in Figure

    // each of the 1×1 squares in this tromino

    // is considered as missing

    let m_1, m_2, m_3, m_4 be the locations of the missing squares

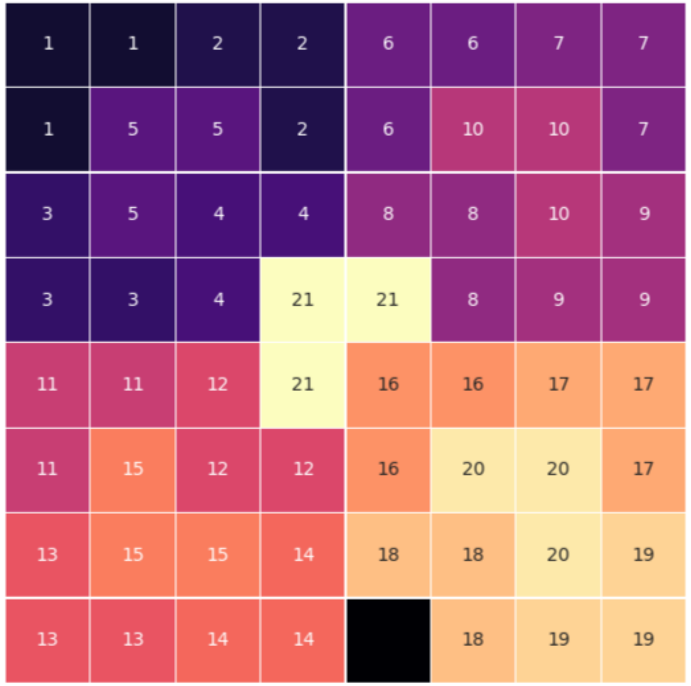
    tile($n/2$, m_1)

    tile($n/2$, m_2)

    tile($n/2$, m_3)

    tile($n/2$, m_4)

}

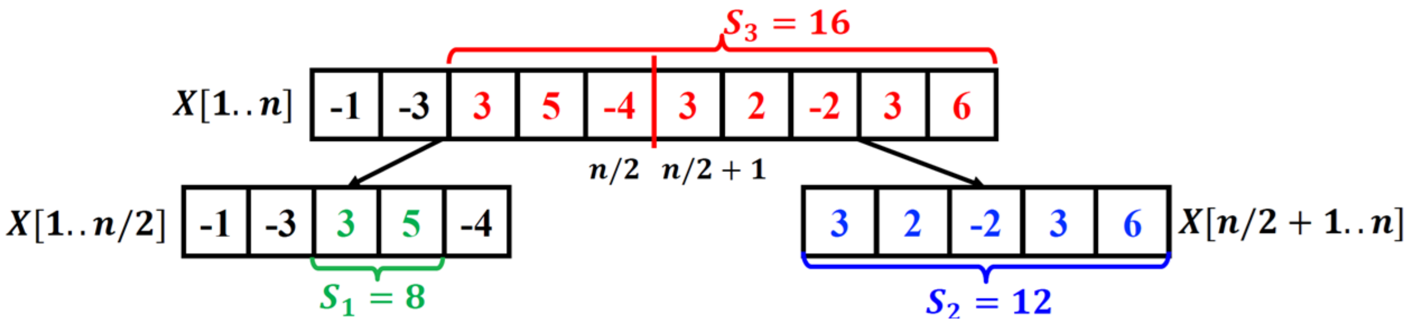


Complexity: T(n) = 4T(n/2) + c. Gives, T(n) = $\Theta(n^2)$

Or, there are $(n^2 - 1)/3$ tiles to be placed, and placing each tile takes O(1) time

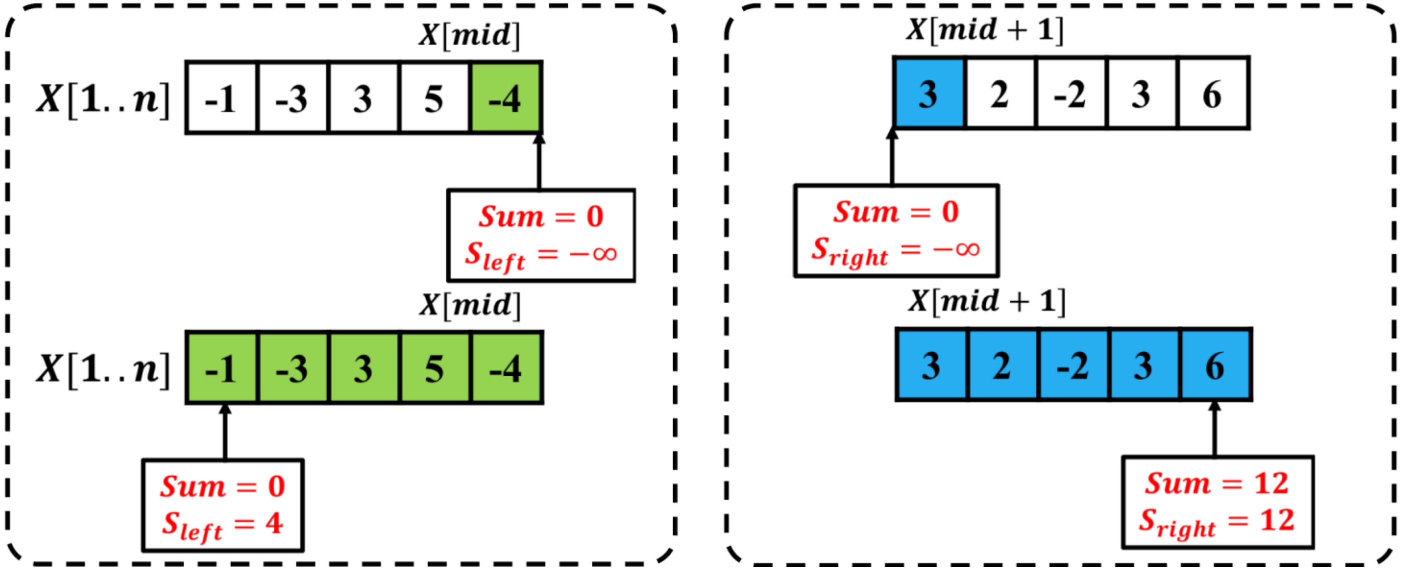
Maximum Subarray

- Given an integer array nums, find the subarray with the largest sum, and return its sum
- X = [-2,1,-3,4,-1,2,1,-5,4], subarray [4,-1,2,1] has the largest sum 6



- Divide $X[1..n]$ into left: $X[1..n/2]$ and right: $X[n/2+1..n]$
- Recursively solve the subproblems
  - $S1$: $X[1..n/2]$'s Maximum Subarray
  - $S2$: $X[n/2+1..n]$'s Maximum Subarray
- Solve the $S3$: crossing left and right
- The largest sum is: $Smax=max(S1,S2,S3)$

Maximum Subarray



Iterate to left   O(n)   Iterate to right

$S_{left} \leftarrow -\infty$
$Sum \leftarrow 0$
for $l \leftarrow mid$ *downto* low **do**
  $Sum \leftarrow Sum + X[l]$
  $S_{left} \leftarrow \max\{S_{left}, Sum\}$
end
$S_{right} \leftarrow -\infty$
$Sum \leftarrow 0$
for $r \leftarrow mid + 1$ *to* $high$ **do**
  $Sum \leftarrow Sum + X[r]$
  $S_{right} \leftarrow \max\{S_{right}, Sum\}$
end
$S_3 \leftarrow S_{left} + S_{right}$
return S_3

The overall complexity of Maximum Subarray using divide and conquer is O(nlogn) by solving T(n) = 2T(n/2) + O(n)

Can we solve it in O(n)? Yes, using Dynamic Programming in the next class.

Exercise: https://leetcode.com/problems/maximum-subarray/

Summary

- To improve divide and conquer
  - Presorting
  - Reduce the number of subproblems
- Solution to the big problem =

    Solution to the left subproblem

    + Solution to the right subproblem

    + Solution to the part that crossing between left and right subproblems
- Make use of existing framework, e.g., mergesort, quicksort
- How to divide?