Discussion 6

CS180: Introduction to Algorithms and Complexity

Prof. Mark Burgin

TAs: Yunqi Guo guoyunqi@gmail.com
Ling Ding lingding@cs.ucla.edu

Outline

Review

- Topological order, DAG
- Greedy algorithm

Divide and Conquer

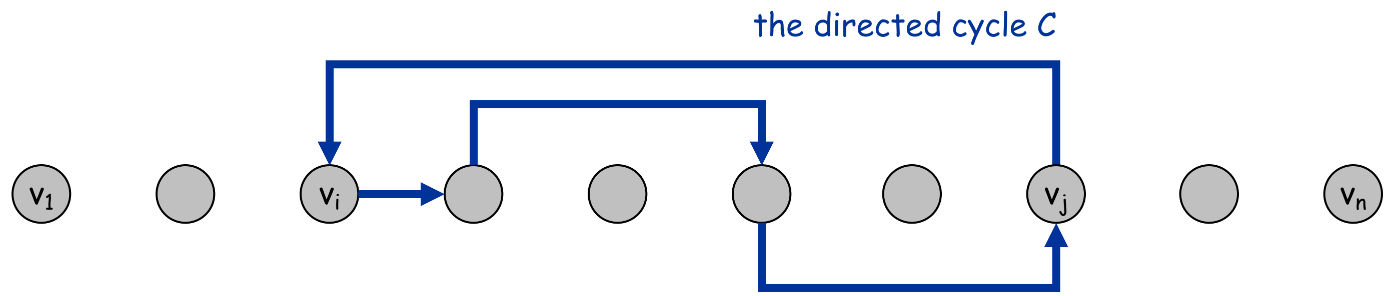
- Mergesort
- Finding the closest pair of points

Directed Acyclic Graphs

Lemma. If G has a topological order, then G is a DAG.

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, \ldots, v_n$ and that G also has a directed cycle C. Let's see what happens.
- Let v_i be the lowest-indexed node in C, and let v_j be the node just before v_i; thus (v_j, v_i) is an edge.
- By our choice of i, we have $i < j$.
- On the other hand, since (v_j, v_i) is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪



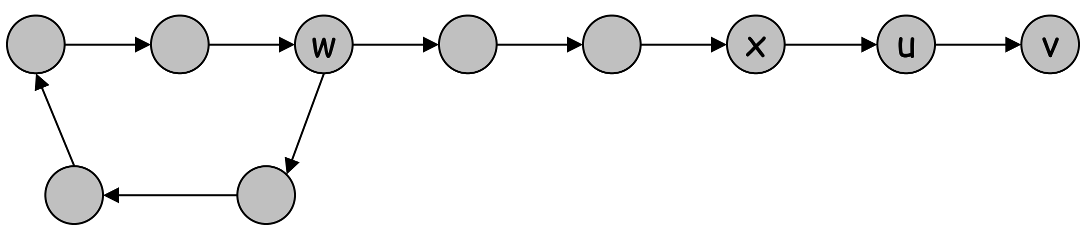
the supposed topological order: $v_1, \ldots, v_n$

Directed Acyclic Graphs

Lemma. If G is a DAG, then G has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that G is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node v, and begin following edges backward from v. Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle. ▪



Algorithmic Paradigms

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

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

5.1 Mergesort

Sorting

Sorting. Given n elements, rearrange in ascending order.

Applications.

- Sort a list of names.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

obvious applications

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

problems become easy once items are in sorted order

- Data compression.
- Computer graphics.
- Computational biology.
- Supply chain management.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

. . .

non-obvious applications

“Divide and Conquer”

Very important strategy in computer science:

- Divide problem into smaller parts
- Independently solve the parts
- Combine these solutions to get overall solution

Idea 1: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → Mergesort

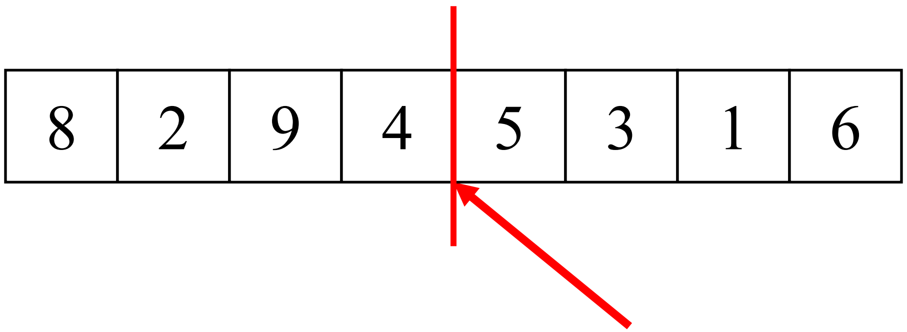
Idea 2 : Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → Quicksort

Mergesort

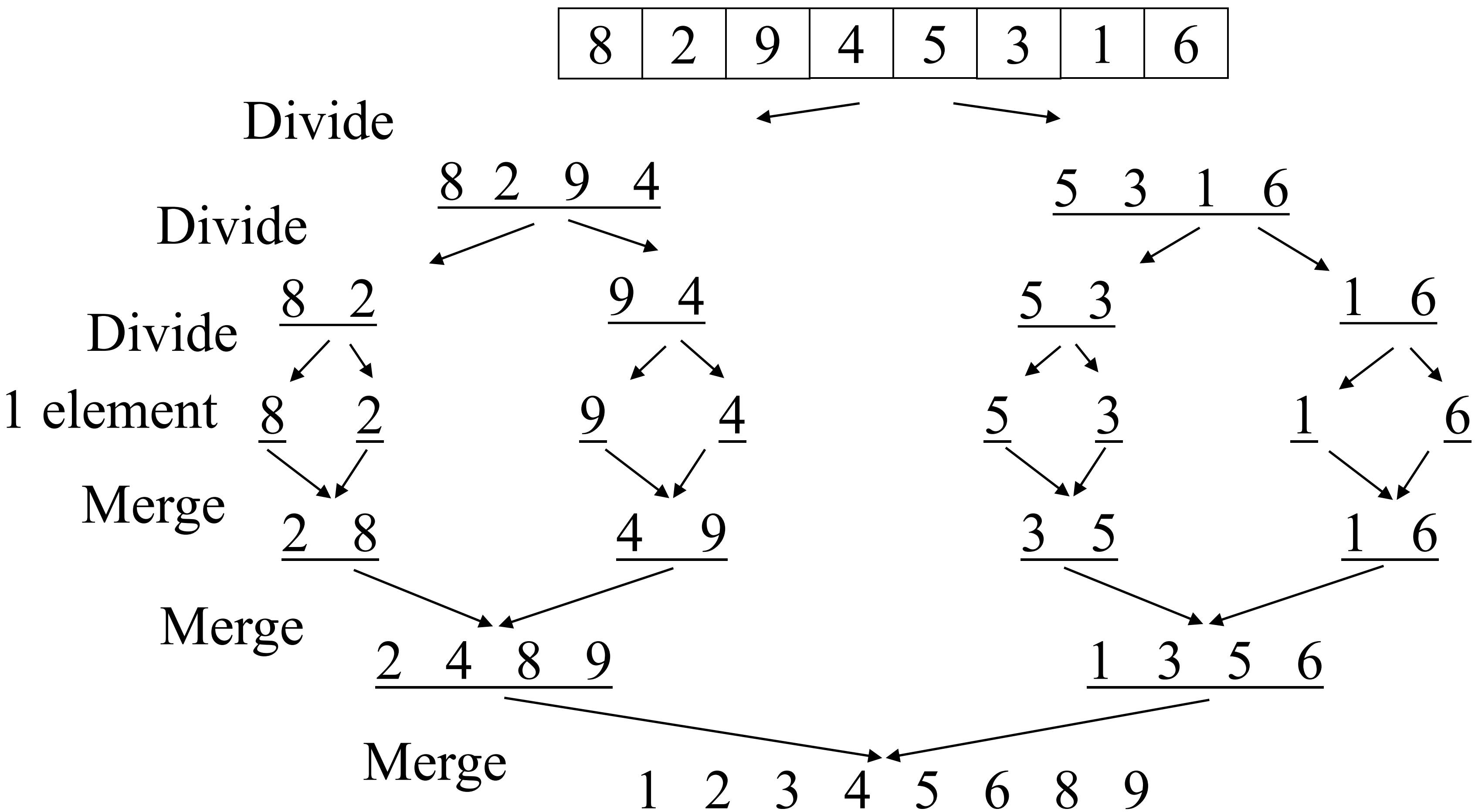
Divide it in two at the midpoint

Conquer each side in turn (by recursively sorting)

Merge two halves together

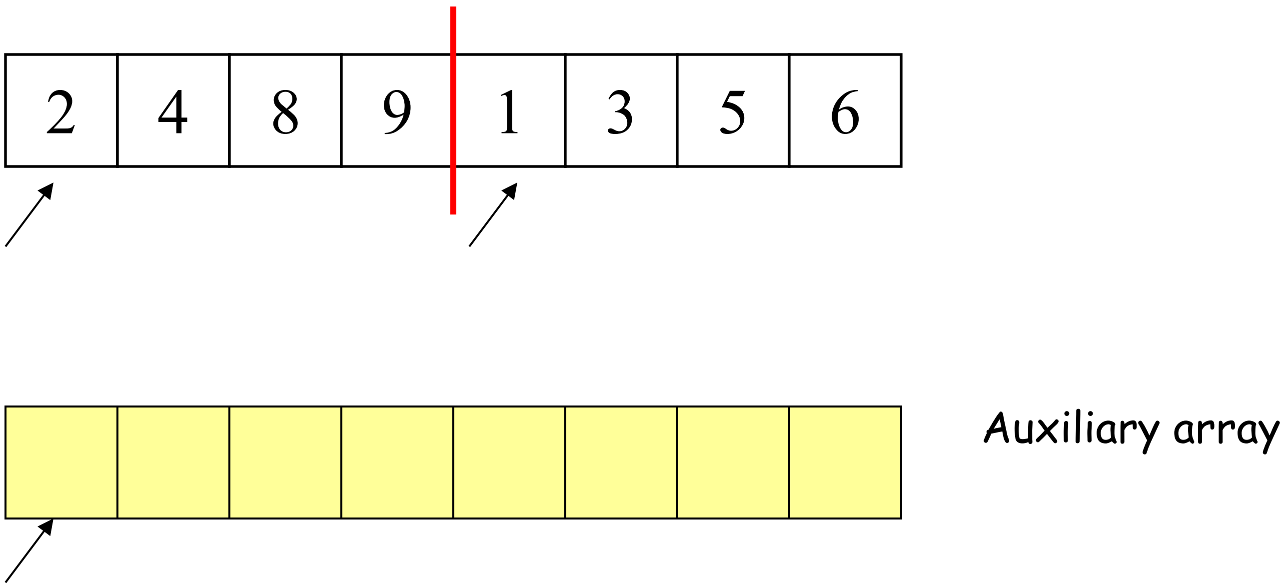


Mergesort Example



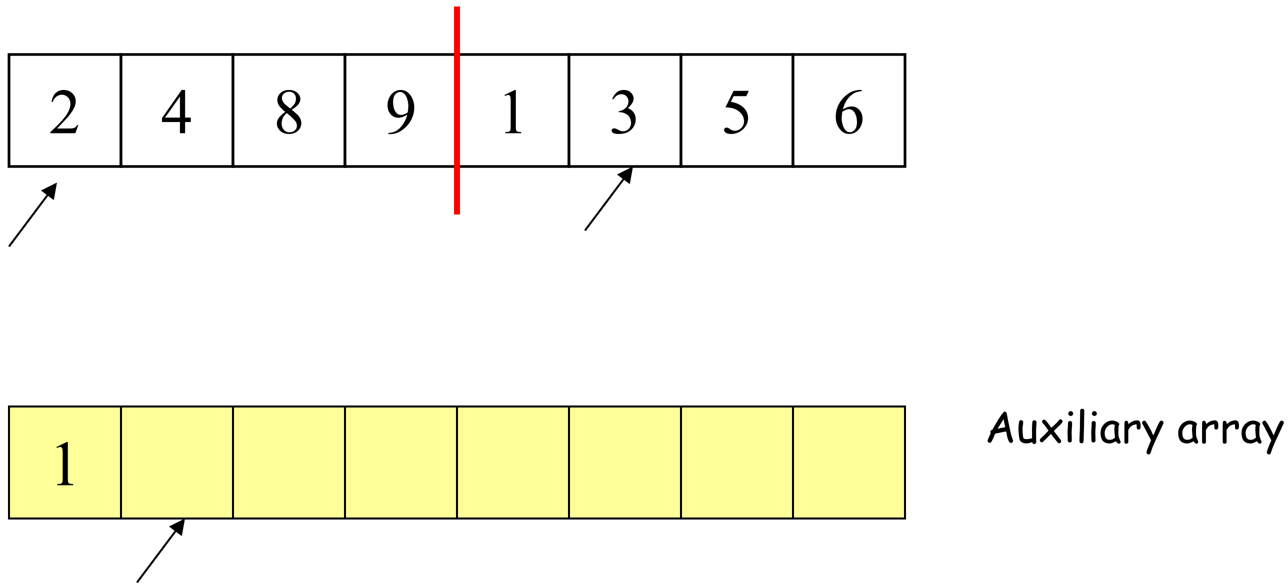
Auxiliary Array

The merging requires an auxiliary array.



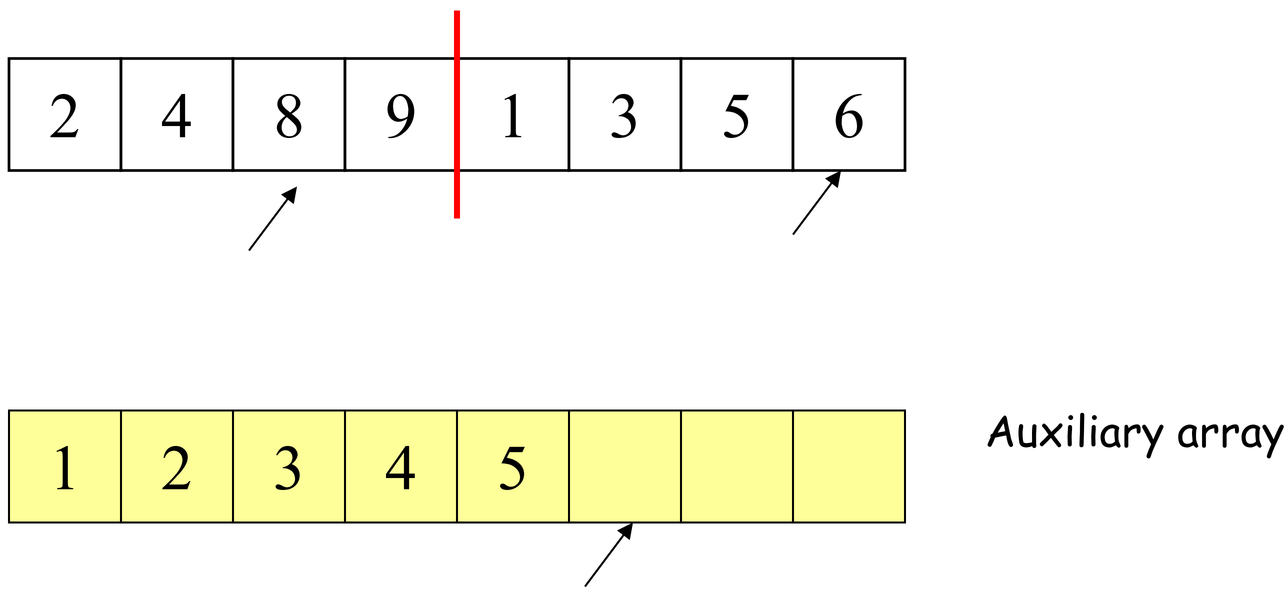
Auxiliary Array

The merging requires an auxiliary array.

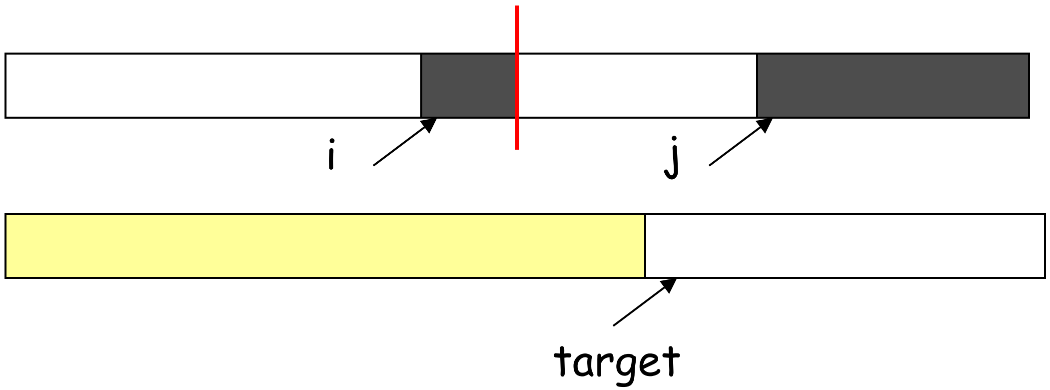


Auxiliary Array

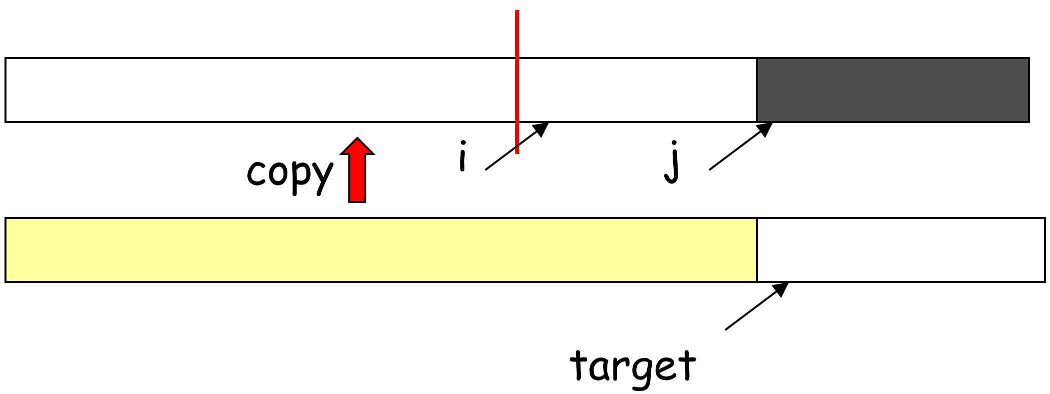
The merging requires an auxiliary array.



Merging

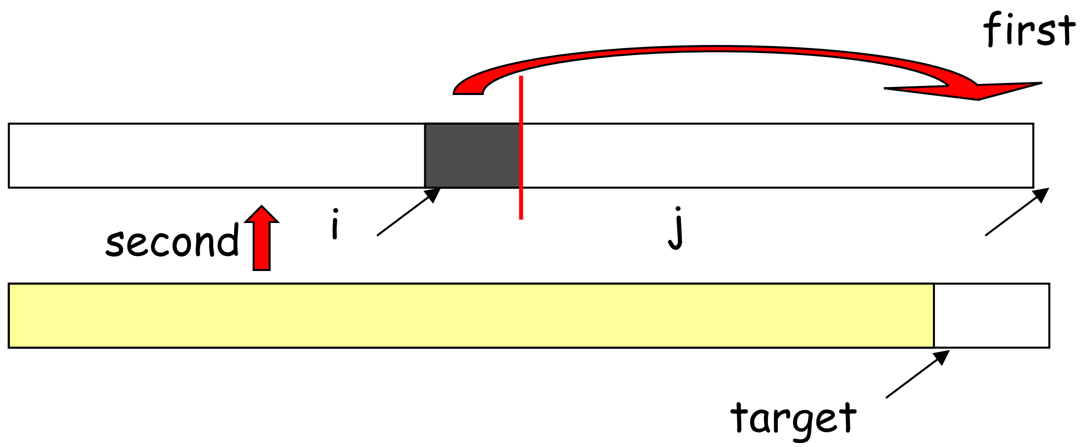


normal



Left completed
first

Merging



Right completed first

Merging

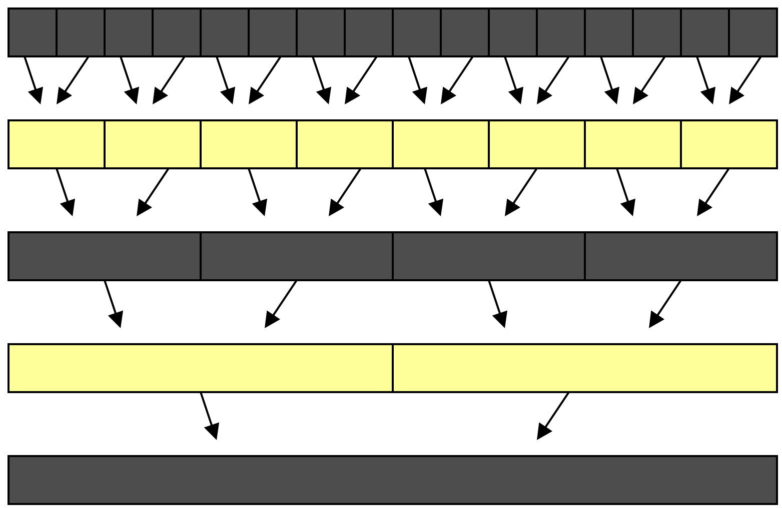
```
Merge(A[], T[] : integer array, left, right : integer) : {
  mid, i, j, k, l, target : integer;
  mid := (right + left)/2;
  i := left; j := mid + 1; target := left;
  while i ≤ mid and j ≤ right do
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;
      else T[target] := A[j]; j := j + 1;
    target := target + 1;
  if i > mid then //left completed//
    for k := left to target-1 do A[k] := T[k];
  if j > right then //right completed//
    k : = mid; l := right;
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;
    for k := left to target-1 do A[k] := T[k];
}
```

Recursive Mergesort

```
Mergesort(A[], T[] : integer array, left, right : integer) : {
  if left < right then
    mid := (left + right)/2;
    Mergesort(A,T,left,mid);
    Mergesort(A,T,mid+1,right);
    Merge(A,T,left,right);
}

MainMergesort(A[1..n]: integer array, n : integer) : {
  T[1..n]: integer array;
  Mergesort[A,T,1,n];
}
```

Iterative Mergesort



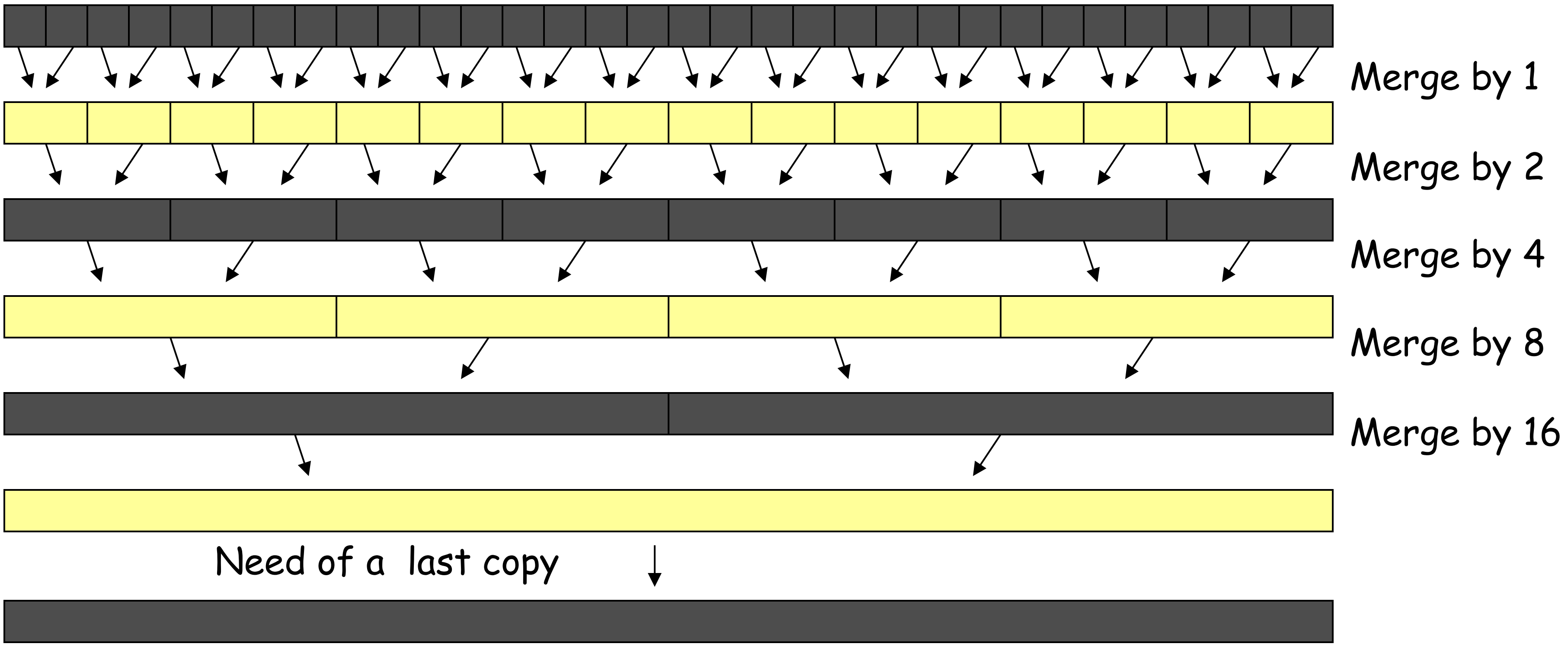
Merge by 1

Merge by 2

Merge by 4

Merge by 8

Iterative Mergesort



Iterative Mergesort

```
IterativeMergesort(A[1..n]: integer array, n : integer) : {
//precondition: n is a power of 2//
  i, m, parity : integer;
  T[1..n]: integer array;
  m := 2; parity := 0;
  while m ≤ n do
    for i = 1 to n - m + 1 by m do
       if parity = 0 then Merge(A,T,i,i+m-1);
         else Merge(T,A,i,i+m-1);
    parity := 1 - parity;
    m := 2*m;
  if parity = 1 then
    for i = 1 to n do A[i] := T[i];
}
```

How do you handle non-powers of 2?
How can the final copy be avoided?

Mergesort Analysis

Let $T(N)$ be the running time for an array of N elements
Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array

Each recursive call takes $T(N/2)$ and merging takes $O(N)$

Mergesort Recurrence Relation

The recurrence relation for T(N) is:

- $T(1) \leq a$
  - base case: 1 element array → constant time
- $T(N) \leq 2T(N/2) + bN$
  - Sorting N elements takes
    - the time to sort the left half
    - plus the time to sort the right half
    - plus an O(N) time to merge the two halves

$T(N) = O(n \log n)$ (see Lecture 5 Slide17)

Properties of Mergesort

Not in-place

- Requires an auxiliary array (O(n) extra space)

Stable

- Make sure that left is sent to target on equal values.

Iterative Mergesort reduces copying.

Related questions

https://leetcode.com/problems/merge-sorted-array/

https://leetcode.com/problems/sort-an-array/

- https://leetcode.com/problems/sort-an-array/discuss/329672/merge-sort

Quicksort

Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does

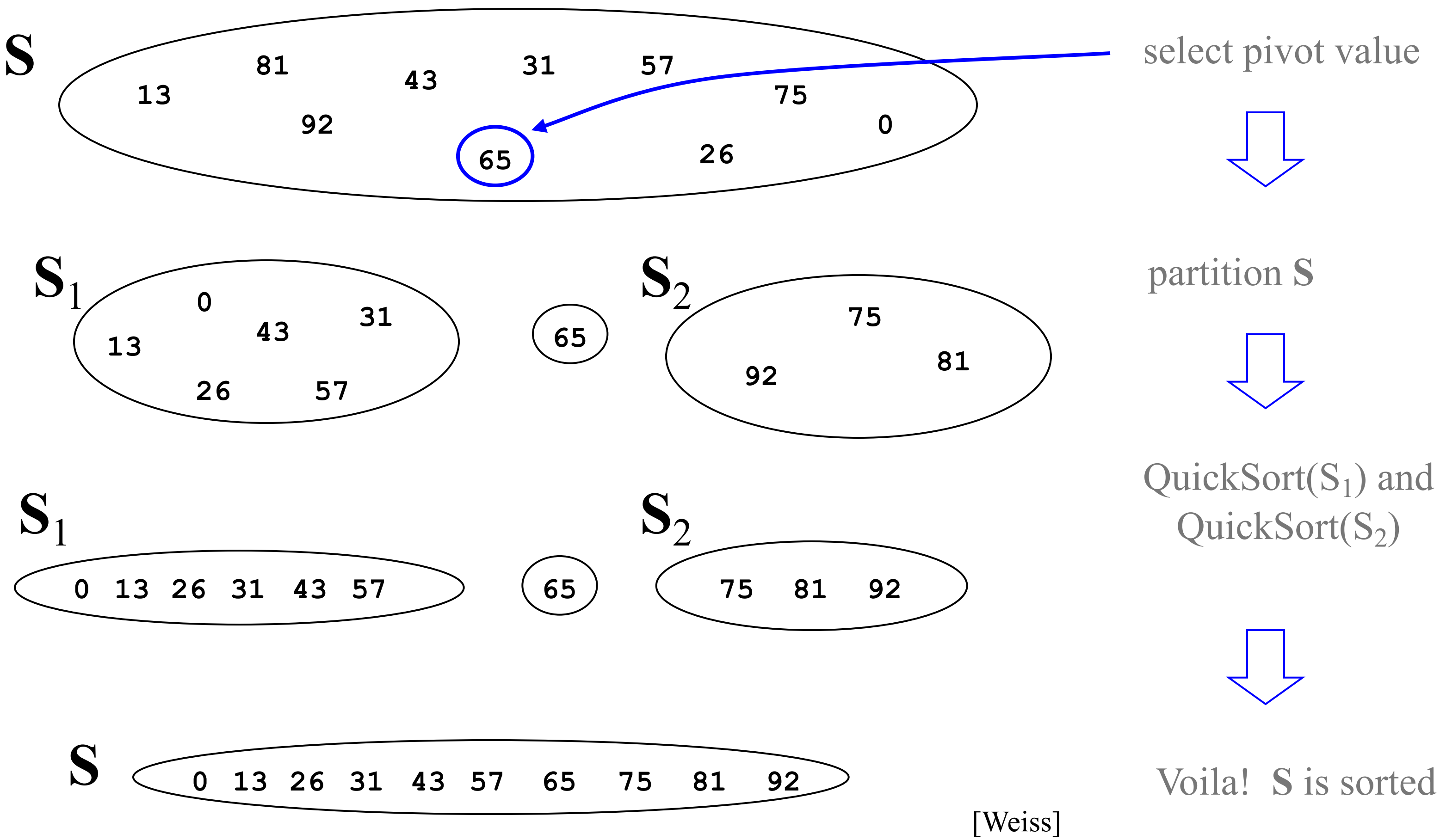
- Partition array into left and right sub-arrays
  - Choose an element of the array, called pivot
  - the elements in left sub-array are all less than pivot
  - elements in right sub-array are all greater than pivot
- Recursively sort left and right sub-arrays
- Concatenate left and right sub-arrays in O(1) time

"Four easy steps"

To sort an array S

1. If the number of elements in S is 0 or 1, then return. The array is sorted.
2. Pick an element v in S. This is the *pivot* value.
3. Partition S-$\{v\}$ into two disjoint subsets, S_1 = {all values $x \leq v$}, and S_2 = {all values $x \geq v$}.
4. Return QuickSort(S_1), v, QuickSort(S_2)

The steps of QuickSort



Details, details

Implementing the actual partitioning

Picking the pivot

- want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible

Dealing with cases where the element equals the pivot

Quicksort Partitioning

Need to partition the array into left and right sub-arrays

- the elements in left sub-array are ≤ pivot
- elements in right sub-array are ≥ pivot

How do the elements get to the correct partition?

- Choose an element from the array as the pivot
- Make one pass through the rest of the array and swap as needed to put elements in partitions

One implementation (there are others)

- median3 finds pivot and sorts left, center, right
  - Median3 takes the median of leftmost, middle, and rightmost elements
  - An alternative is to choose the pivot randomly (need a random number generator; "expensive")
  - Another alternative is to choose the first element (but can be very bad. Why?)
- Swap pivot with next to last element

Partitioning in-place

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit elmt A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

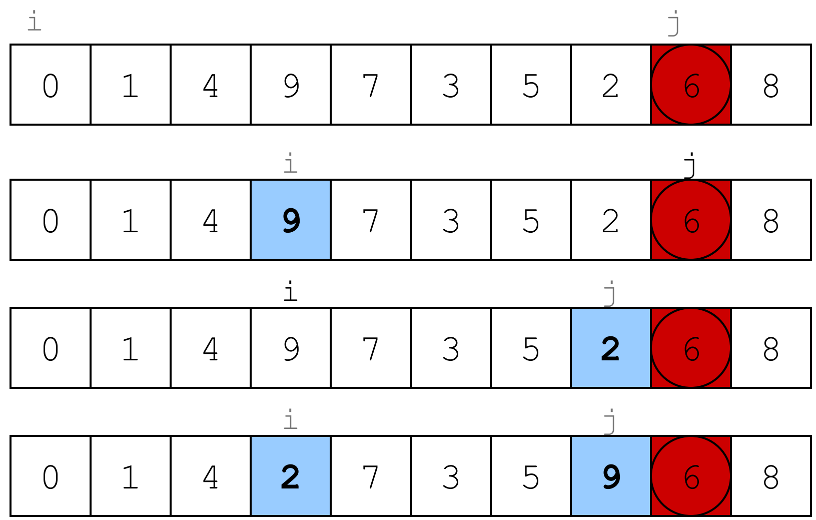
0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

Median of 0, 6, 8 is 6. Pivot is 6

0	1	4	9	7	3	5	2	6	8
i								j	

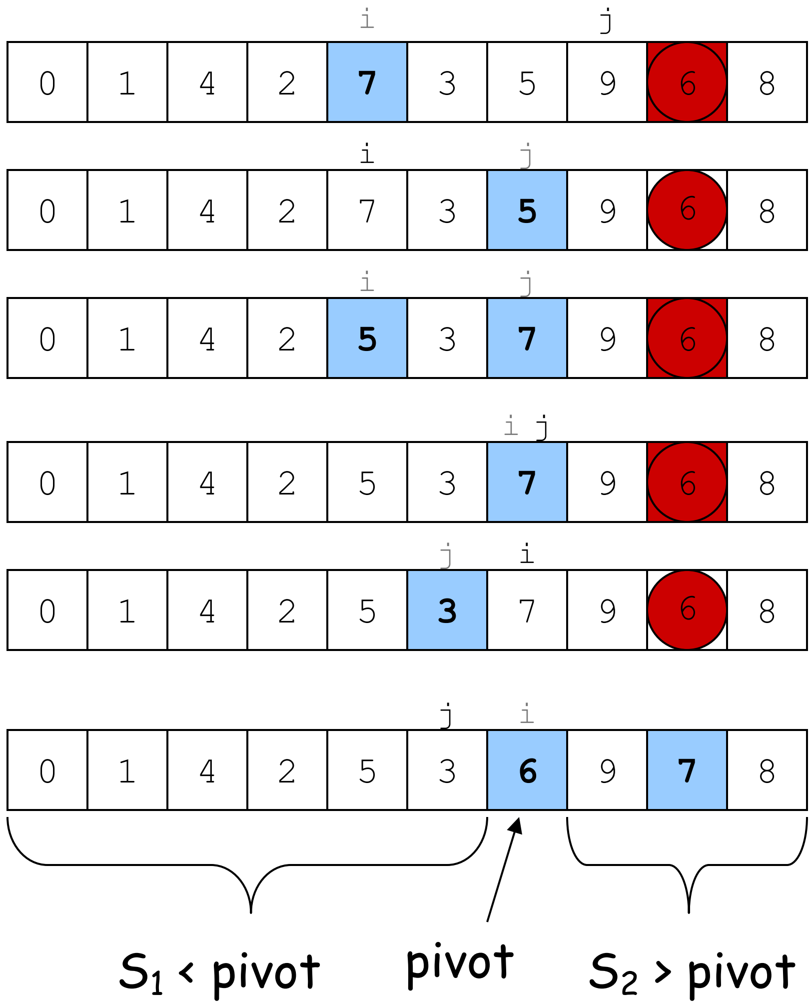
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example



Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example



Cross-over i > j

Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don’t use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Quicksort Best Case Performance

Algorithm always chooses best pivot and splits sub-arrays in half at each recursion

- $T(0) = T(1) = O(1)$
  - constant time if 0 or 1 element
- For $N > 1$, 2 recursive calls plus linear time for partitioning
- $T(N) = 2T(N/2) + O(N)$
  - Same recurrence relation as Mergesort
- $T(N) =$ $O(N \log N)$

Quicksort Worst Case Performance

Algorithm always chooses the worst pivot – one sub-array is empty at each recursion

- $T(N) \leq a$ for $N \leq C$
- $T(N) \leq T(N-1) + bN$
- $\leq T(N-2) + b(N-1) + bN$
- $\leq T(C) + b(C+1) + \ldots + bN$
- $\leq a + b(C + (C+1) + (C+2) + \ldots + N)$
- $T(N) = O(N^2)$

Fortunately, *average case performance* is O(N log N) (see text for proof)

Properties of Quicksort

Not stable because of long distance swapping.
No iterative version (without using a stack).
Pure quicksort not good for small arrays.
"In-place", but uses auxiliary storage because of recursive call ($O(\log n)$ space).
$O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

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.

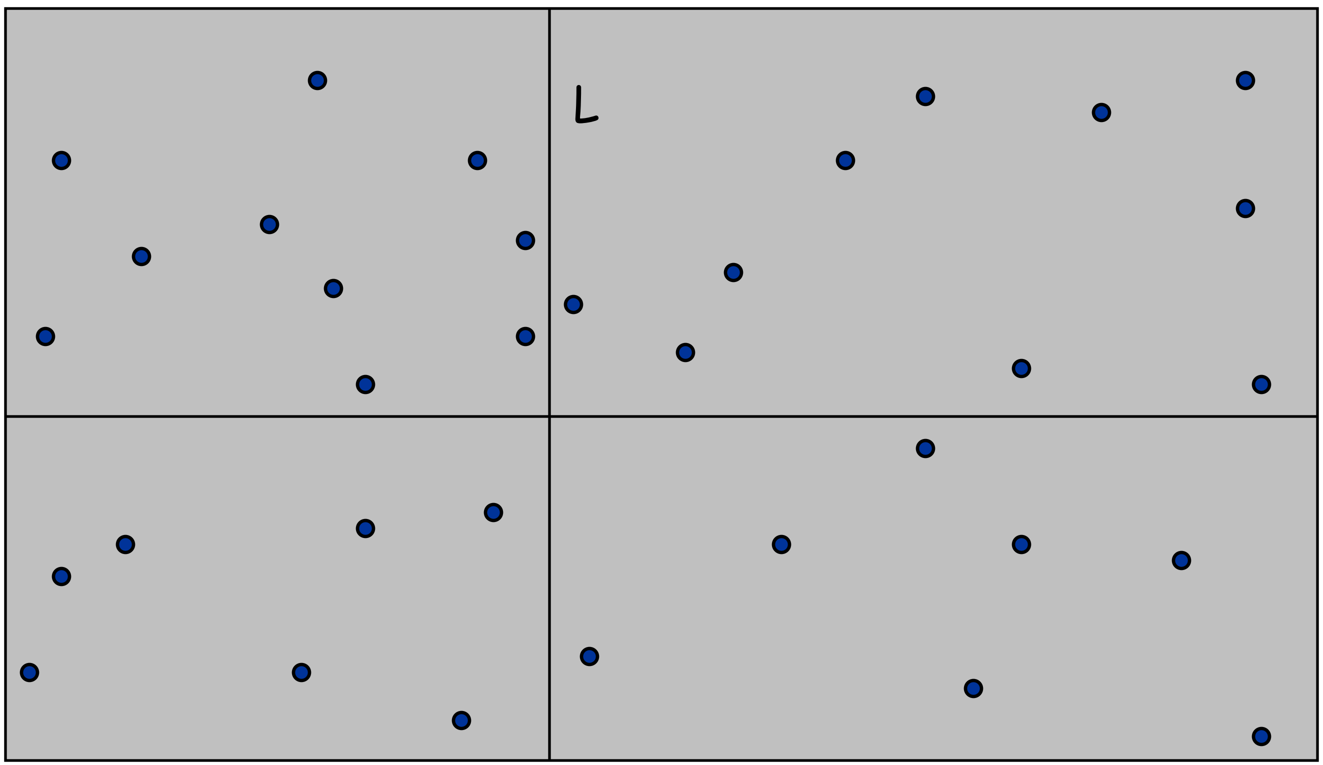
1-D version. $O(n \log n)$ easy if points are on a line.

Assumption. No two points have same x coordinate.

to make presentation cleaner

Closest Pair of Points: First Attempt

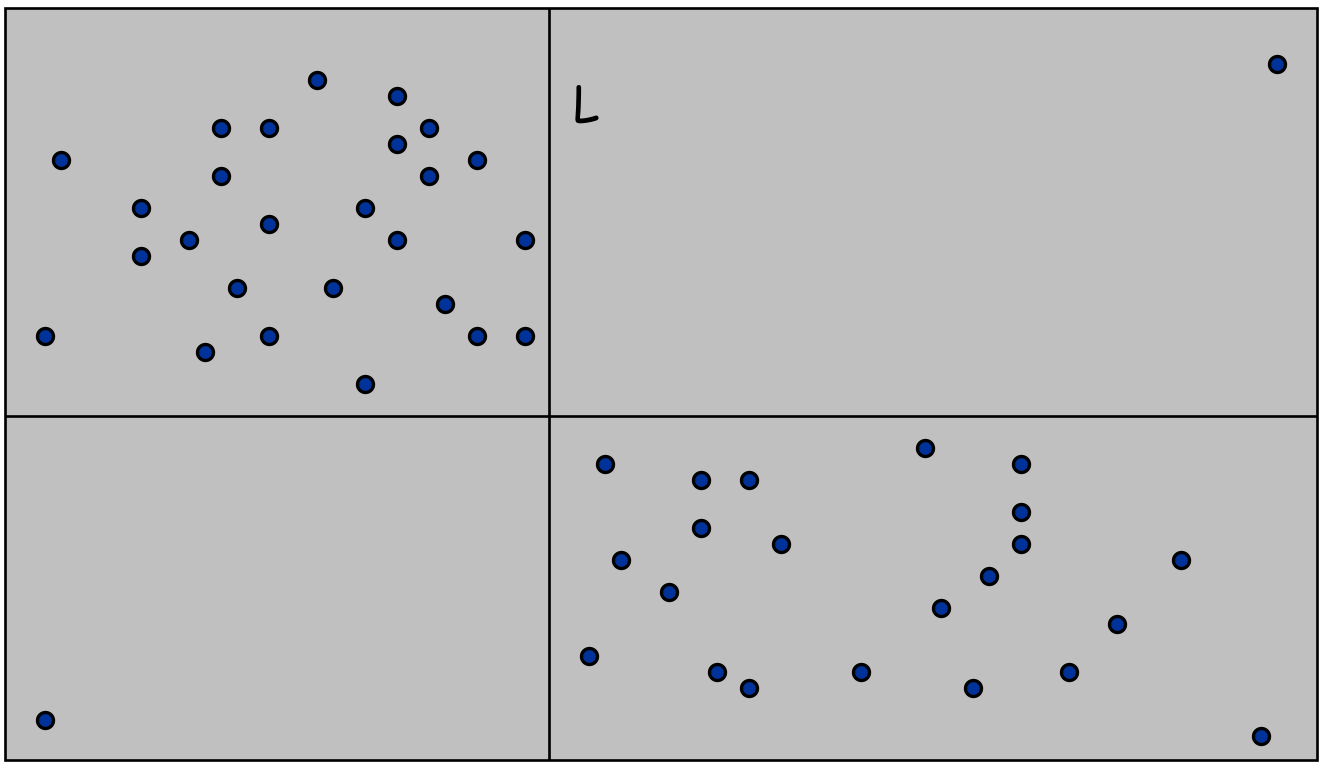
Divide. Sub-divide region into 4 quadrants.



Closest Pair of Points: First Attempt

Divide. Sub-divide region into 4 quadrants.

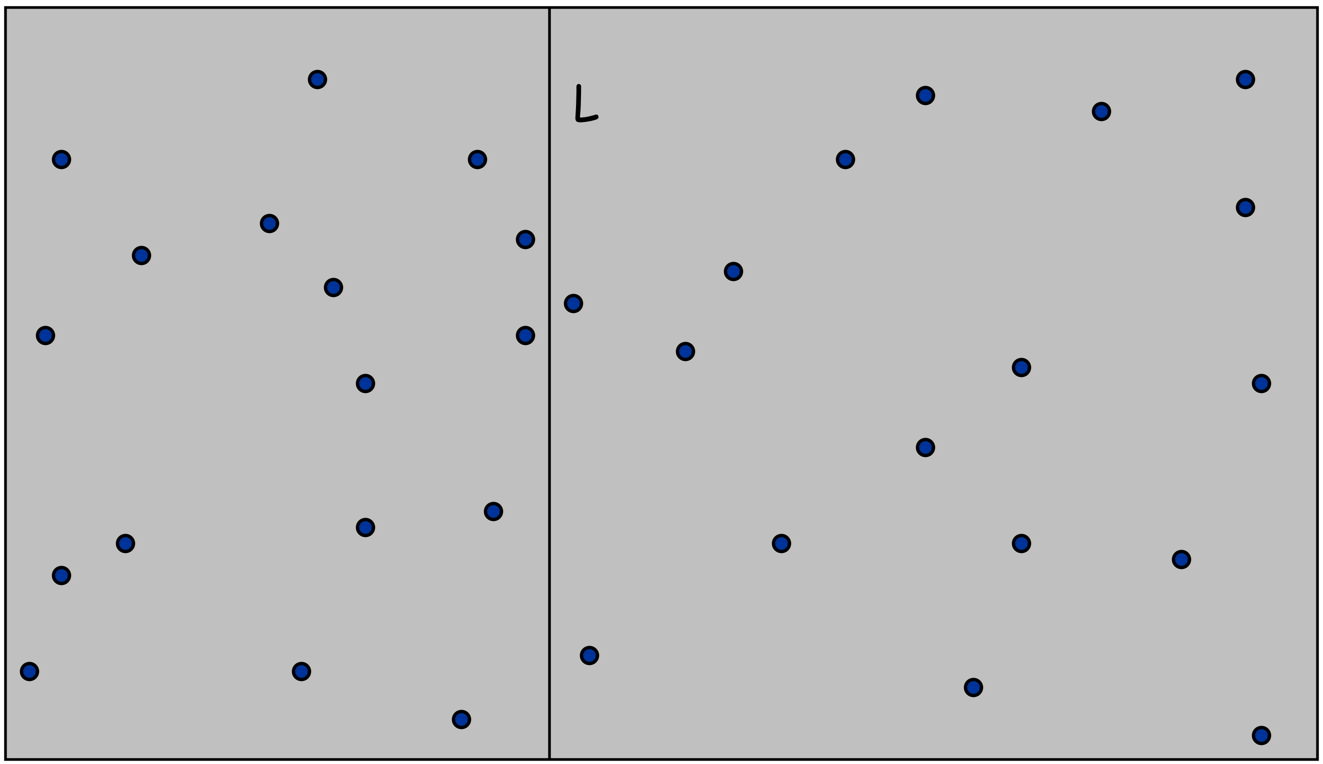
Obstacle. Impossible to ensure n/4 points in each piece.



Closest Pair of Points

Algorithm.

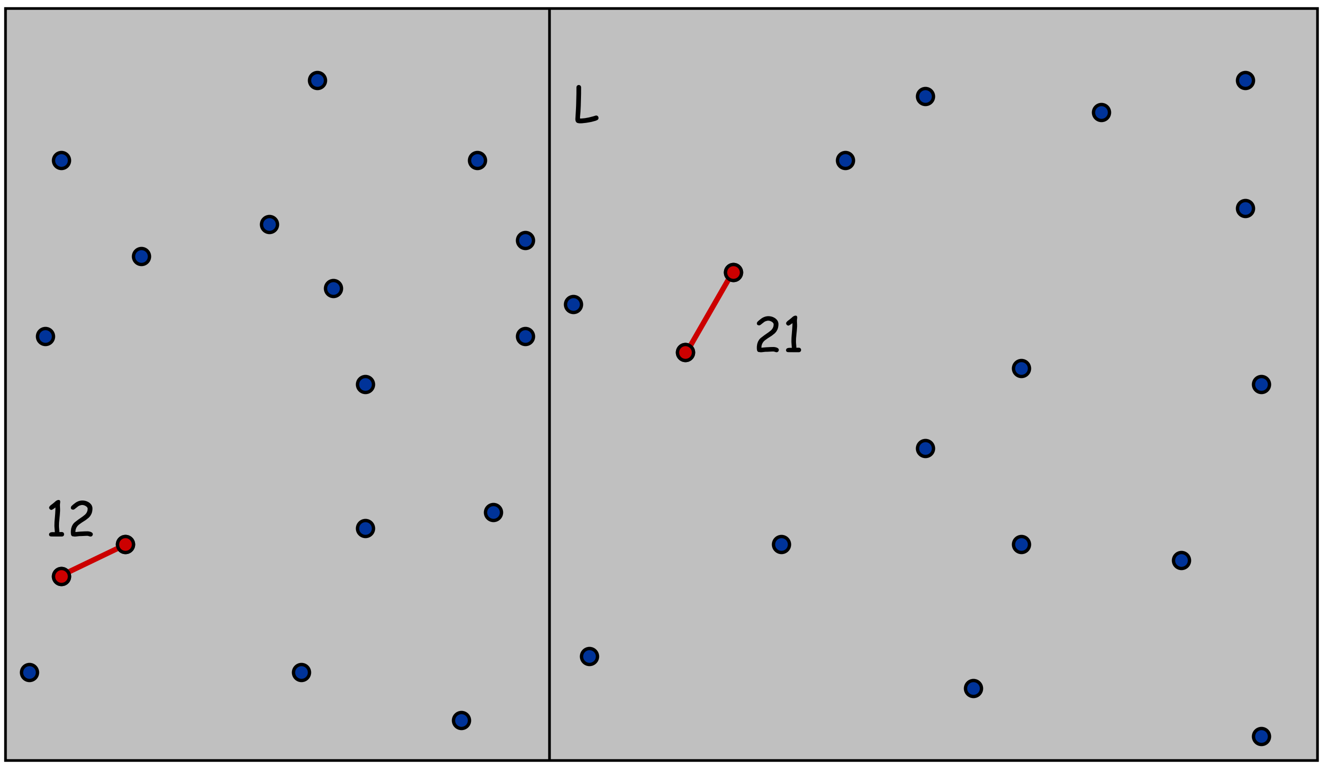
- Divide: draw vertical line L so that roughly $\frac{1}{2}$n points on each side.



Closest Pair of Points

Algorithm.

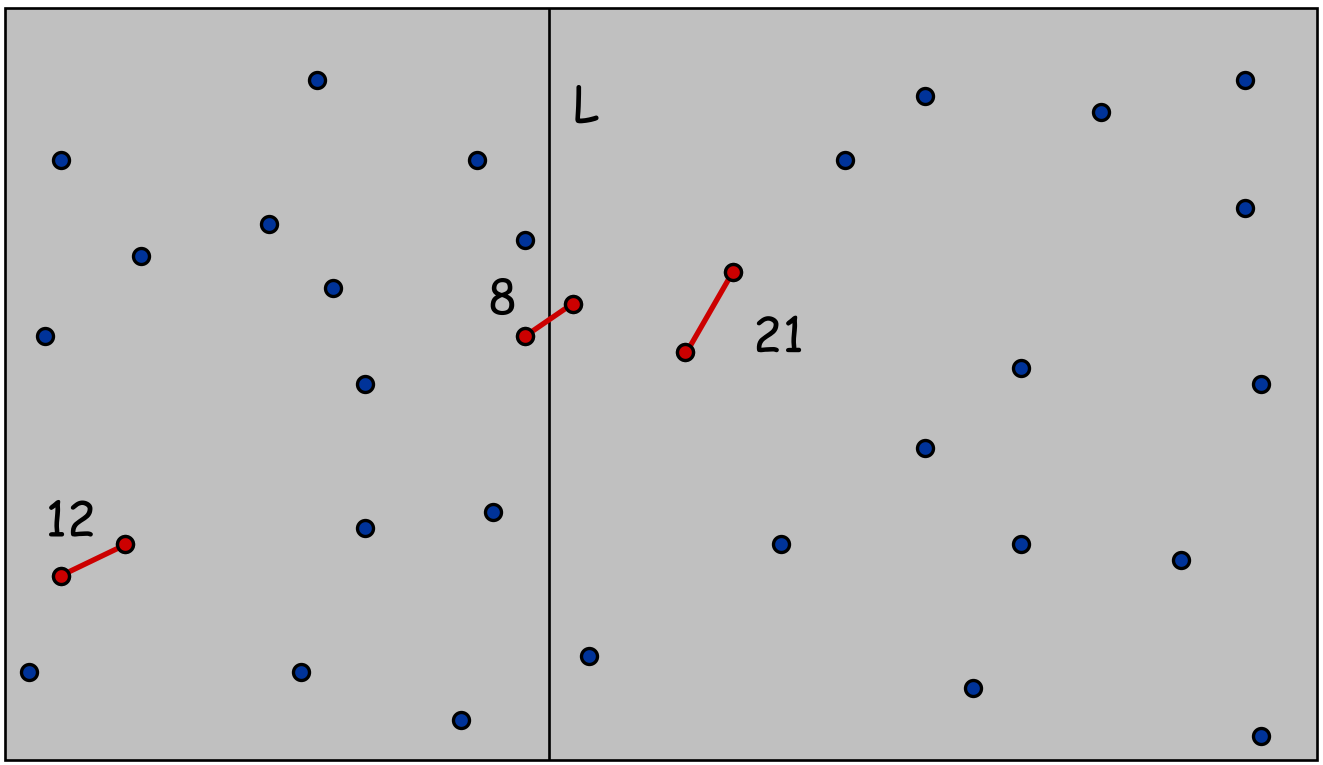
- Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
- Conquer: find closest pair in each side recursively.



Closest Pair of Points

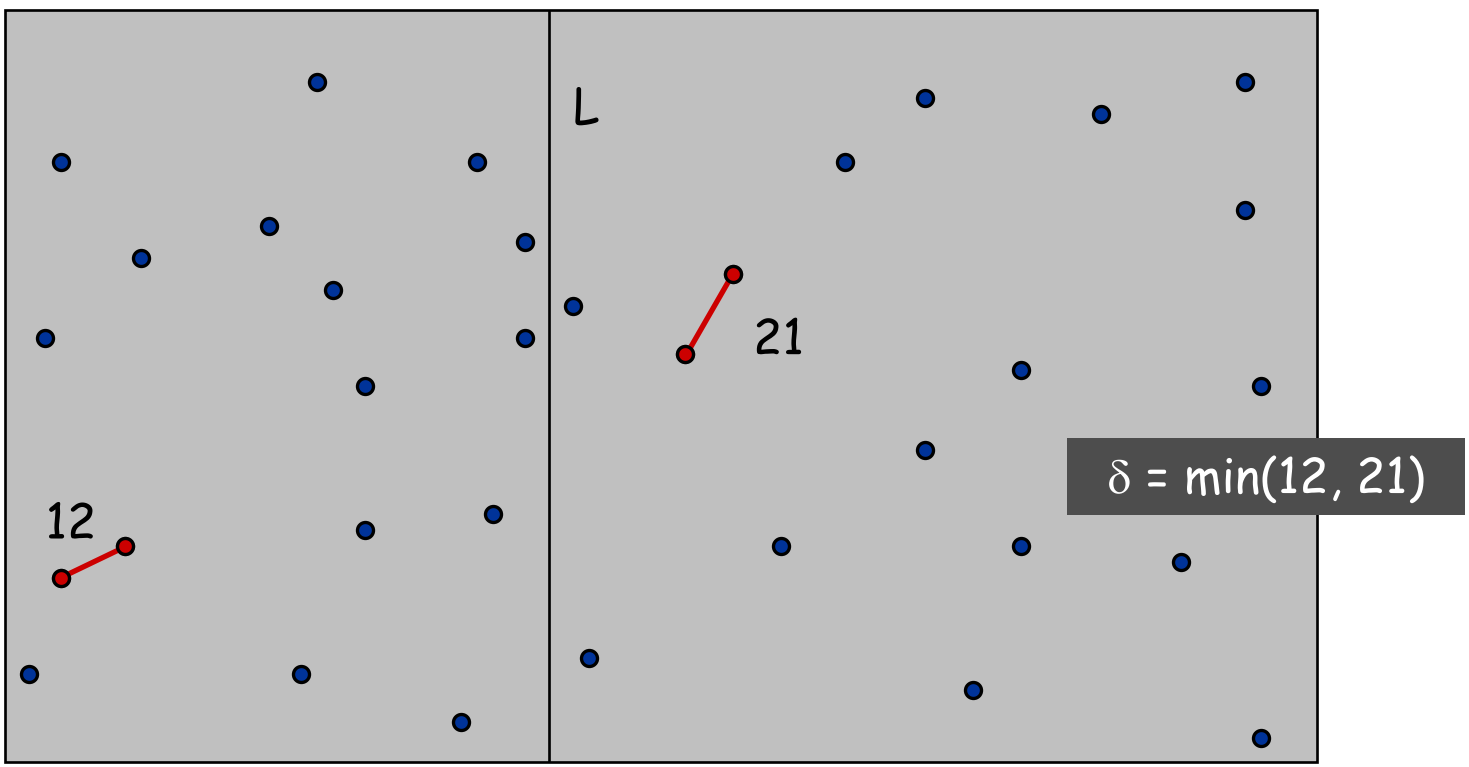
Algorithm.

- 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

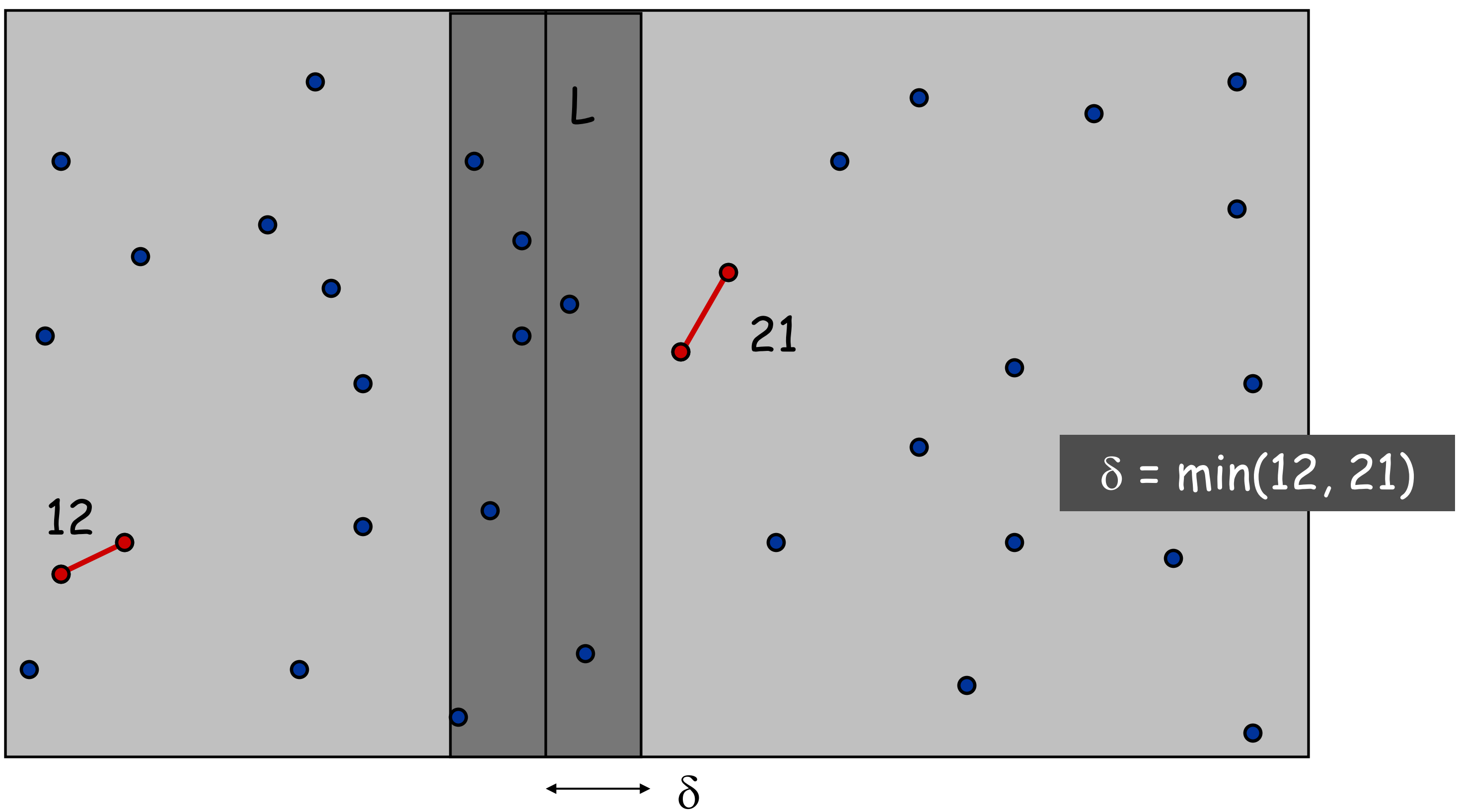
Find closest pair with one point in each side, assuming that distance < δ.



Closest Pair of Points

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

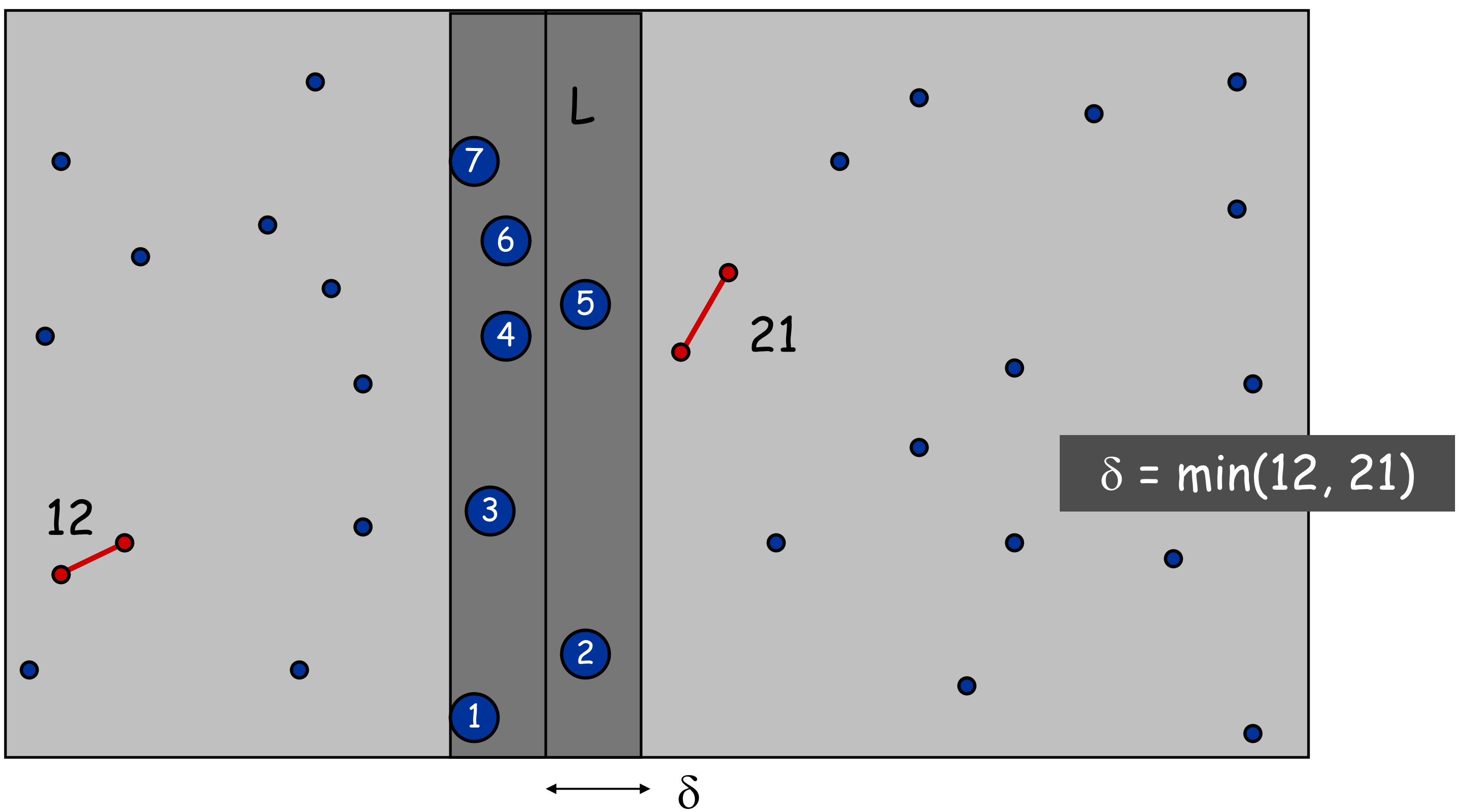
- Observation: only need to consider points within δ of line L.



Closest Pair of Points

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

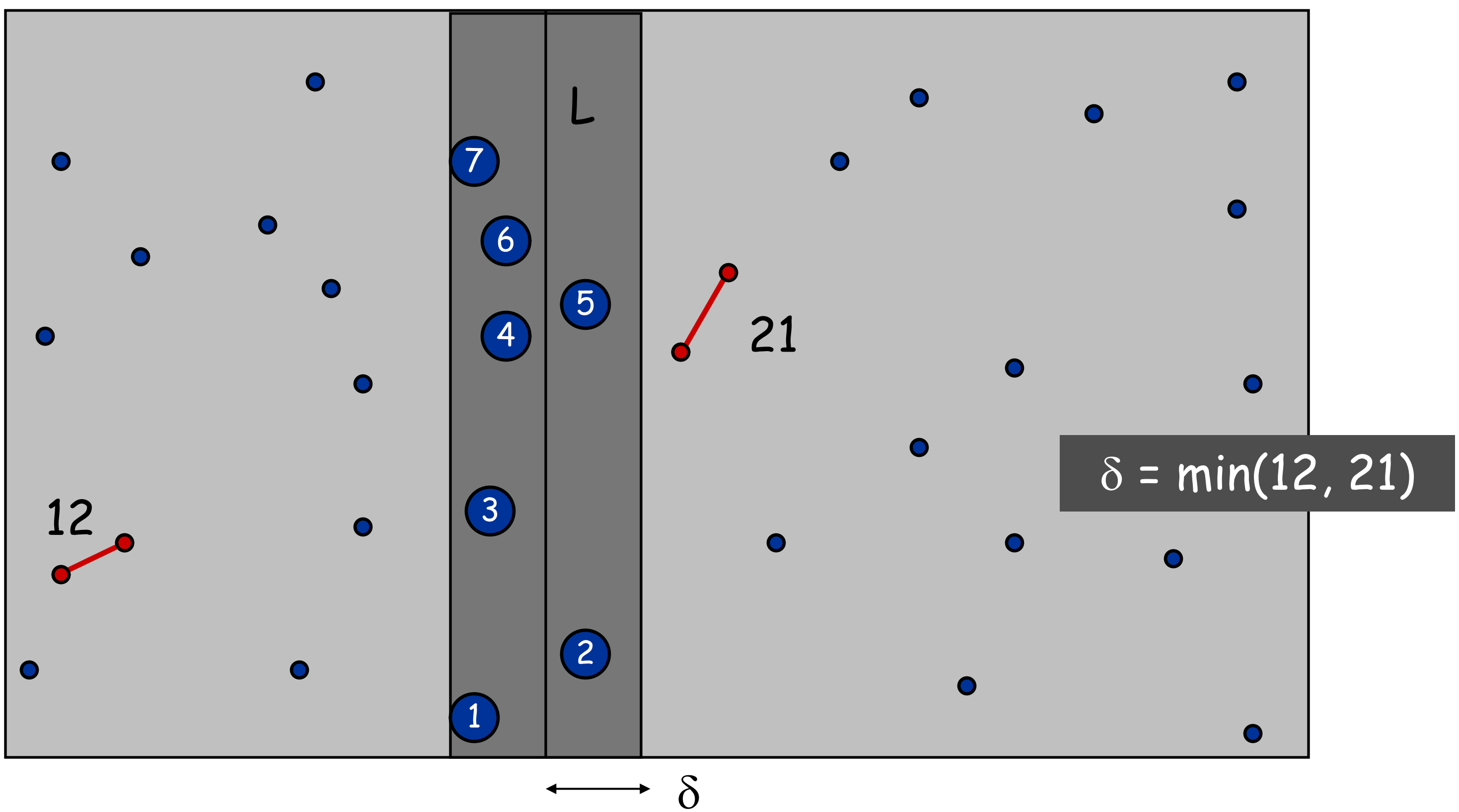
- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.



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

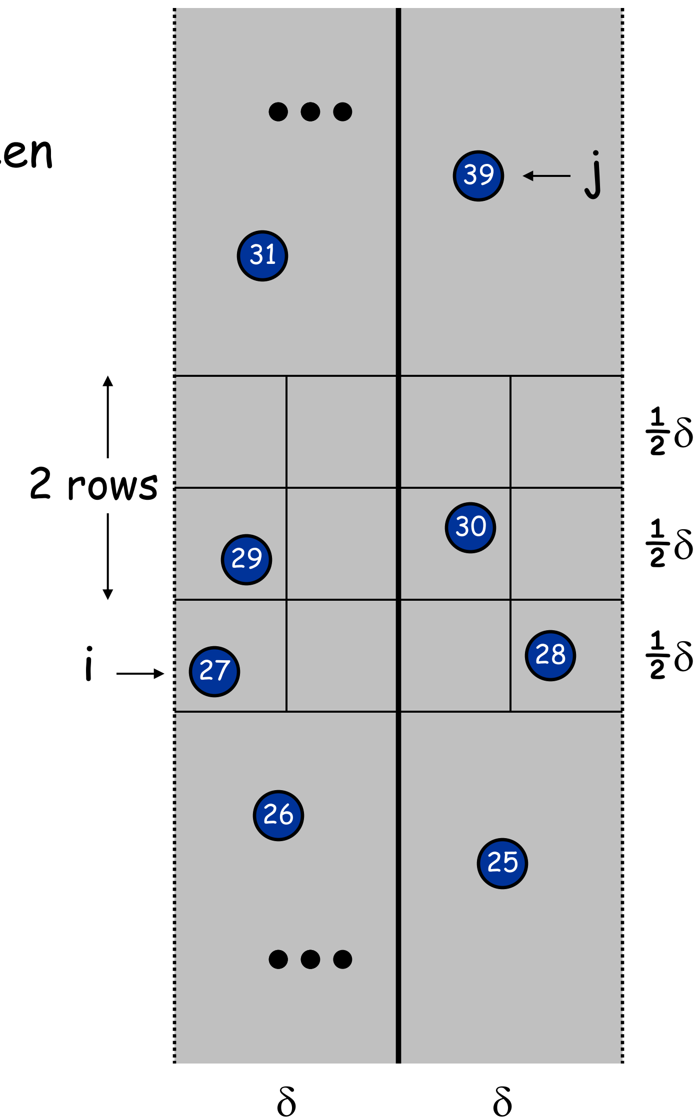
Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.

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     O(n log n)
   are on one side and half on the other side.

   δ1 = Closest-Pair(left half)                             2T(n / 2)
   δ2 = Closest-Pair(right half)
   δ  = min(δ1, δ2)

   Delete all points further than δ from separation line L  O(n)

   Sort remaining points by y-coordinate.                   O(n log n)

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

   return δ.
}
```

Closest Pair of Points: Analysis

Running time?

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.

Related questions

https://leetcode.com/problems/k-closest-points-to-origin/