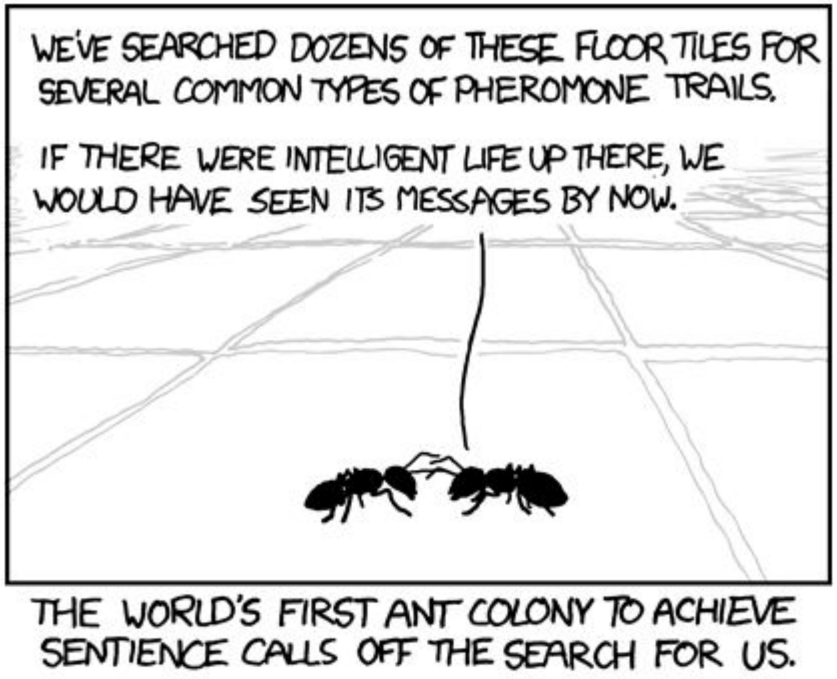
COSC 262 **Algorithms**

Computational Geometry

Part 2: Range Search

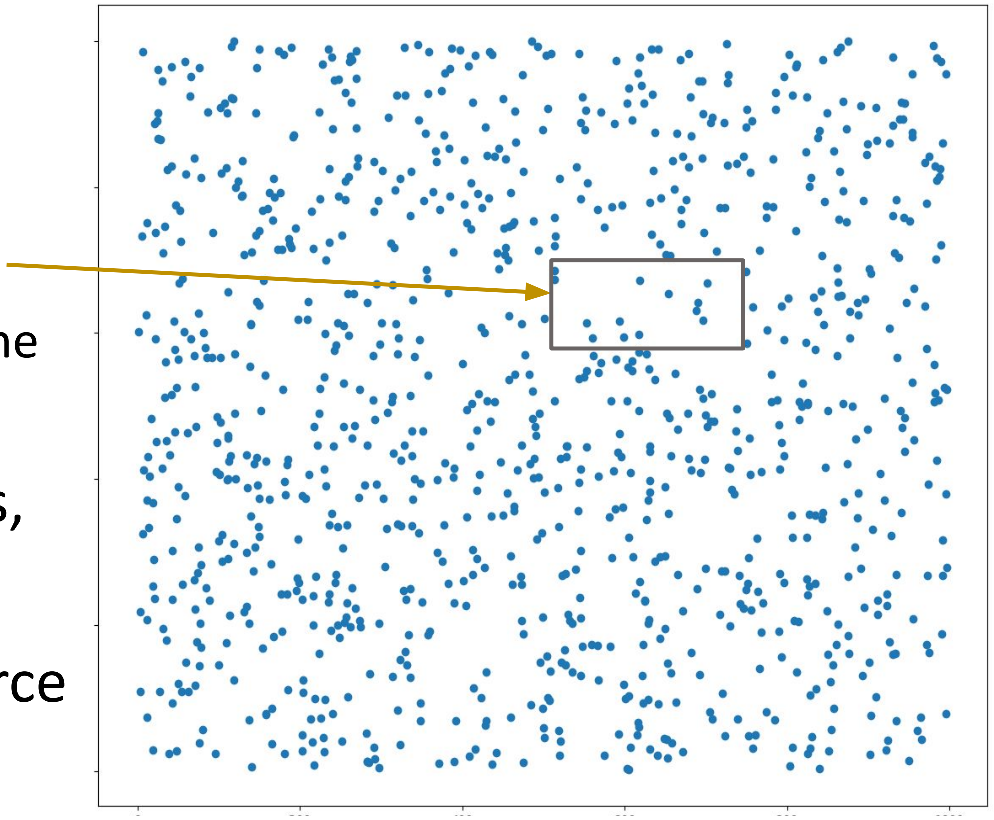

WE'VE SEARCHED DOZENS OF THESE FLOOR TILES FOR SEVERAL COMMON TYPES OF PHEROMONE TRAILS.
IF THERE WERE INTELLIGENT LIFE UP THERE, WE WOULD HAVE SEEN ITS MESSAGES BY NOW.
THE WORLD'S FIRST ANT COLONY TO ACHIEVE SENTIENCE CALLS OFF THE SEARCH FOR US.

https://xkcd.com/638/

Richard Lobb & R. Mukundan
richard.lobb@canterbury.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury

Range searching

- Suppose we have a large number of geometric objects
- How to find all objects intersecting a given area/volume?
  - Called a *range query*
- We will assume:
  - the objects are points
  - the range is rectangular
  - there are many queries on same point set
- Common problem in graphics, CAD, GIS, databases
- Want to improve on brute force "check all points"



First consider 1D Range Searching

- Given a set of *n* numbers, determine which of them lie within a given range [$r_{\min}$, $r_{\max}$].
- For simplicity, assume all numbers are distinct
- Example: which of the following lie in the range [30, 42]?

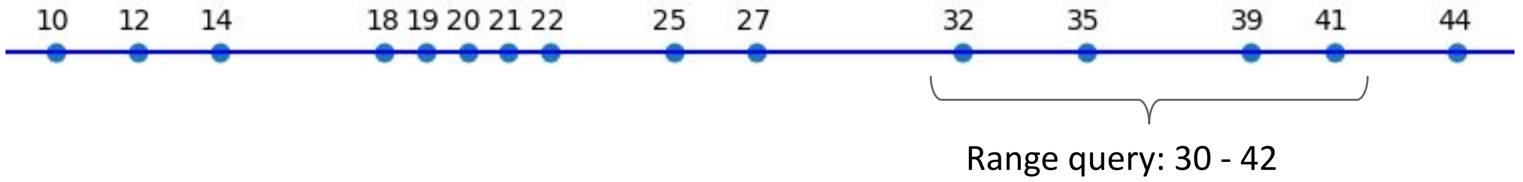
  22, 41, 19, 27, 12, 35, 14, 20, 39, 10, 25, 44, 32, 21, 18
- A single search takes ***O*(*n*)** time.

Python: `ans = [x for x in nums if rmin <= x <= rmax]`

- How can we improve on this if *n* is very large **and there are lots of queries**?

Sorting as a solution

Sort numbers



For each query:

```
Find smallest i such that nums[i] >= rmin
matches = []
while i < n and nums[i] <= rmax:
    matches.append(nums[i])
    i += 1
```

Binary search: O(log n)

Cost:

Initial O(*n log n)* for sort.

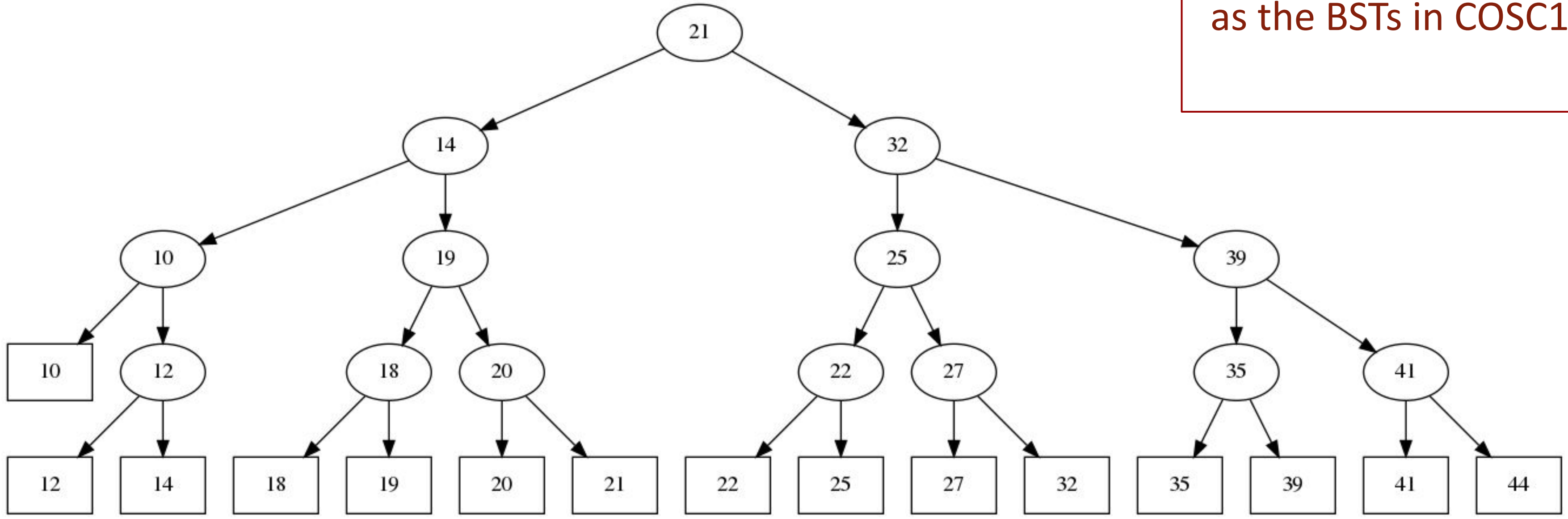
Each query O(log n + m) where *m* is the result set size.

Big win if m << n and lots of queries

Making a tree of it

- That works fine in 1D but doesn't generalise to higher dimensions.
- So: make a balanced binary search tree **with numbers at leaves**
- We'll see later that this is a 1D k-d tree
- Internal nodes are medians (**not true median** but a near-central value)
- Left subtree contains numbers <= median
- Right subtree contains numbers > median

NB: this is *not* the same as the BSTs in COSC122



Code to build tree

```
from            collections            import            namedtuple
Node = namedtuple("Node", ["value", "left", "right"])

def binary_search_tree(nums, is_sorted=False):
      """Return a balanced binary search tree with the given nums
          at the leaves. is_sorted is True if nums already sorted.
       Inefficient because of slicing but more readable.
    """
    if not is_sorted:
        nums = sorted(nums)
    n = len(nums)
    if n == 1:
        tree = Node(nums[0], None, None)  # A leaf
    else:
        mid = n // 2  # Halfway (approx)
        left = binary_search_tree(nums[:mid], True)
        right = binary_search_tree(nums[mid:], True)
        tree = Node(nums[mid - 1], left, right)
    return tree
```

Q1: rewrite it without using slicing.

Q2: what are the O complexities with and without slicing?

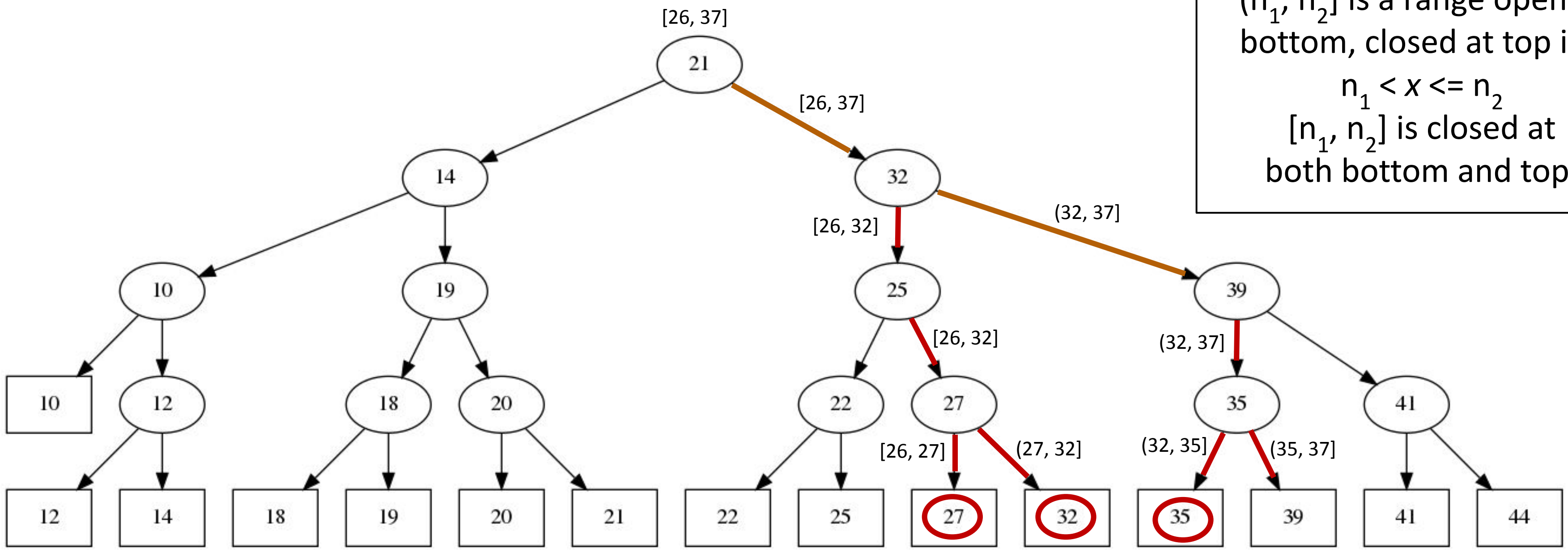
Searching the tree with a range (pseudocode)

```
function search(range, tree):
    if tree is a leaf node:
        return [tree.value] if value is in range else []
    else (it is an internal node):
        items = []
        if range.min <= node.value:
            items += search(range, tree.left)
        if range.max > node.value:
            items += search(range, tree.right)
        return items
```

$(n_1, n_2]$ is a range open at bottom, closed at top i.e.

$n_1 < x <= n_2$

$[n_1, n_2]$ is closed at both bottom and top.



Range search code

```
def range_query(tree, low, high):
    """Return a list of the numbers in range min to max
       inclusive within the given binary search tree.
    """
    matches = []
    if tree.left is None and tree.right is None:  # Leaf?
        if low <= tree.value <= high:
            matches.append(tree.value)
    else: # Internal node
        if low <= tree.value and tree.left is not None:
            matches += range_query(tree.left, low, high)
        if high > tree.value and tree.right is not None:
            matches += range_query(tree.right, low, high)
    return matches
```

1D range search cost

- Algorithm visits at least 1 leaf node (the empty result case), so has a minimum cost of O(log *n*).
- It visits at most a further *m* leaf nodes where *m* is result set size.
- The internal nodes visited while reaching those leaves is (typically) m/2 + m/4 + … + 1 which is less than m.
- Hence the overall cost is O(log *n* + *m*)

Multi-Dimensional Data

- Each item has *d* aspects or coordinates (x_1, .. x_d)
  - Eg. Attributes of a product (price, size, power rating, ...)
  - Attributes of an employee (age, salary, years in service)
- **Multi-dimensional range search**: find all items whose aspects all fall within the given ranges.
  - Eg. Find all employees with (age, salary) in the query range

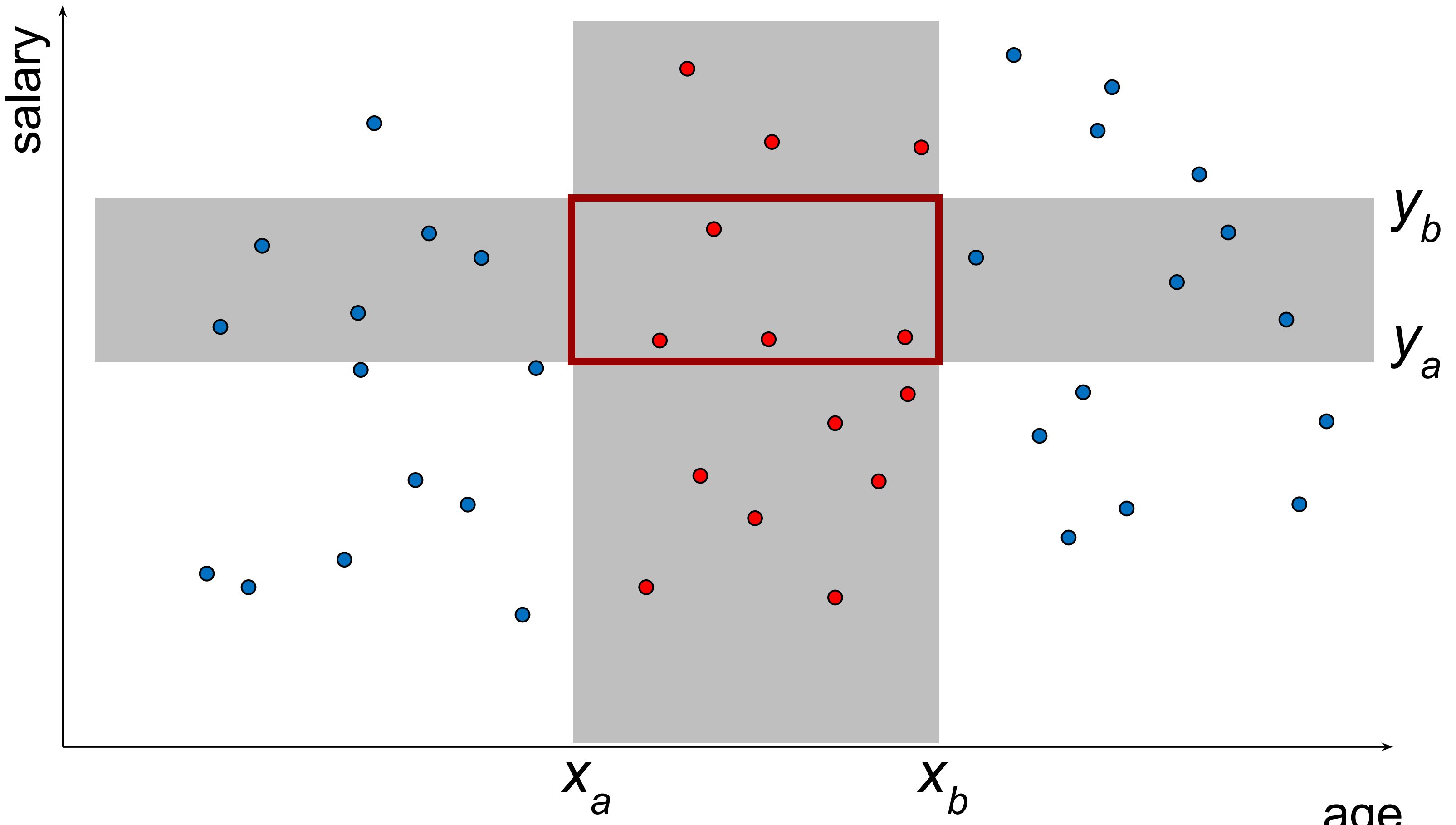
$$[(\underset{\substack{\uparrow\\ \text{amin}}}{25}, \underset{\substack{\uparrow\\ \text{smin}}}{55000}), \quad (\underset{\substack{\uparrow\\ \text{amax}}}{35}, \underset{\substack{\uparrow\\ \text{smax}}}{85000})]$$

Python brute force code:

```
result = [x for x in employees  if amin <= x[0] <= amax  and  smin <= x[1] <= smax]
```

2D Range Searching

A 2D query range $[(x_a, y_a), (x_b, y_b)]$ can be viewed as the intersection of strips formed by two 1-D intervals



2D Range Searching

Implementation options:

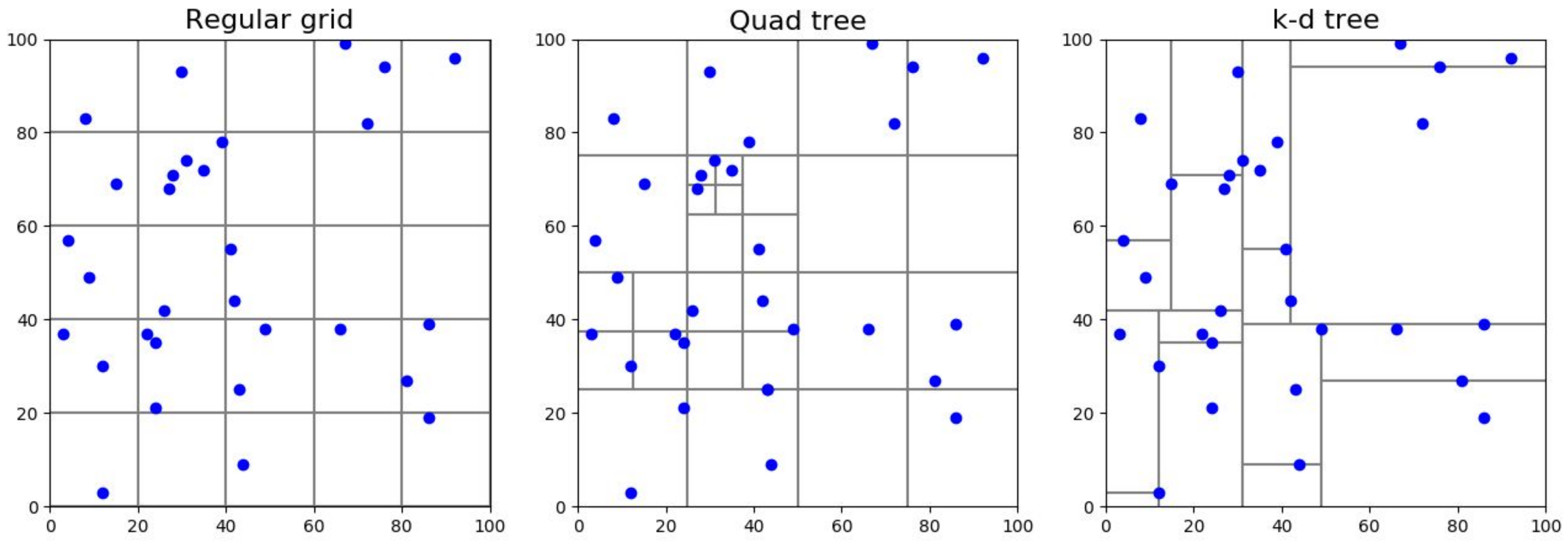
1. **Brute force:**
   - Check each point against the range. O(*n*).
   - OK if not many queries
2. **Projection:**
   - Do a range search on one axis.
   - Using the data returned from the previous search, do a range search on the other axis.
     - But **inefficient**. Can build a tree for first search but second search depends on output from first, so can't pre-process.
   - Or have two copies of the data, sorted on different axes. Do two range searches + set intersection.
     - Not a bad option, except for need for two copies. Can we do better?

2D Range Searching (cont'd)

3. Spatial subdivision

- Break search space into sub-areas, each with its own point set.
- Search only the sub-areas overlapping search range.
- Types:
  - Regular grid
    - Easy
    - Works well if point set fairly uniformly distributed, small range-query rectangles
    - Arbitrarily poor otherwise
  - Quadtree
    - Recursively subdivide space into 4 sub-rectangles until a sufficiently small number of points is in each area (or maximum depth reached)
  - k-d tree
    - Recursively subdivide point set in two, alternating between x and y axes

Spatial subdivision types



25 cells
0 - 5 points / cell

28 cells
0 - 2 points / cell

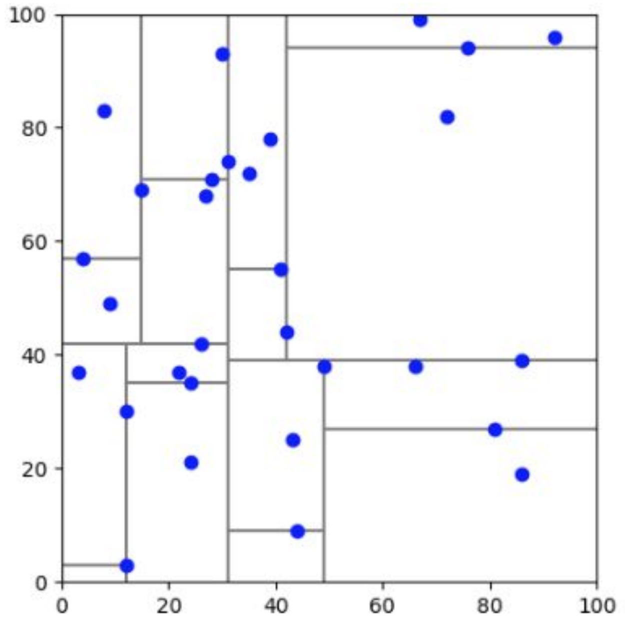
16 cells
1 - 2 points / cell

Easier to implement and update ←

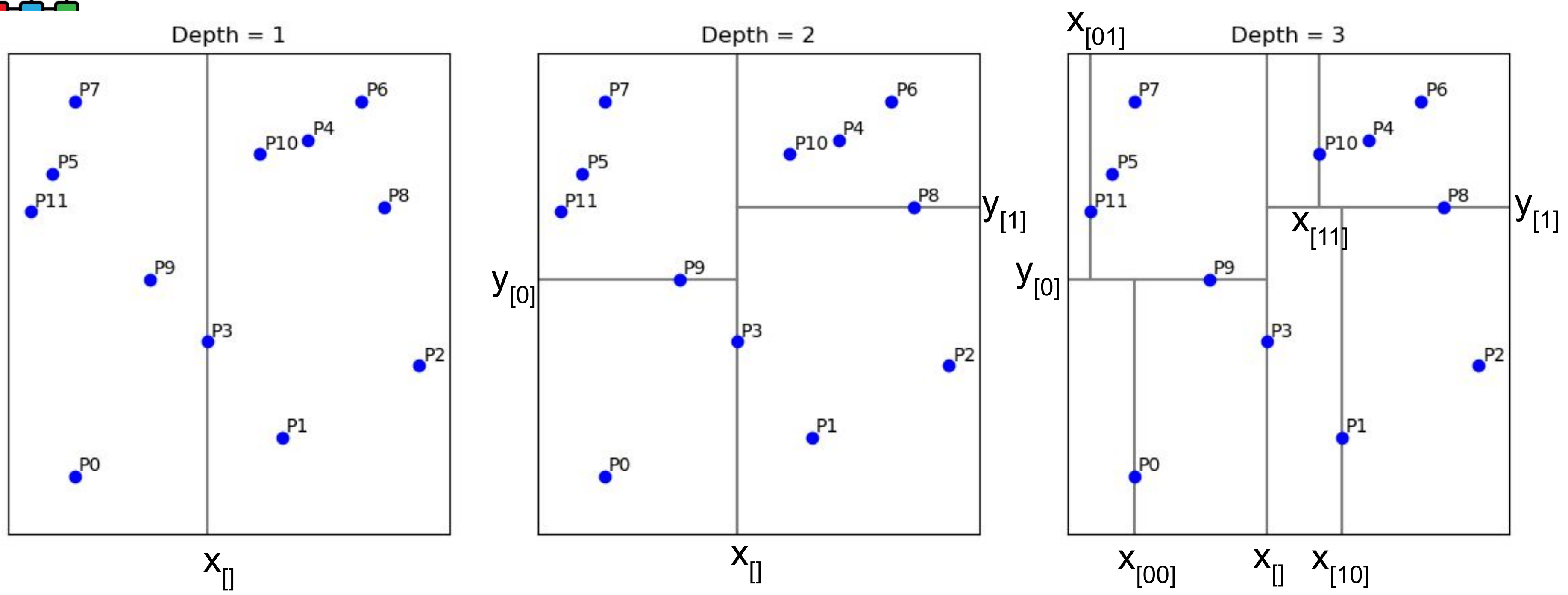
More efficient with clustered data and/or large range searches →

k-d trees (“Axis-aligned Binary Space Partioning Trees”)

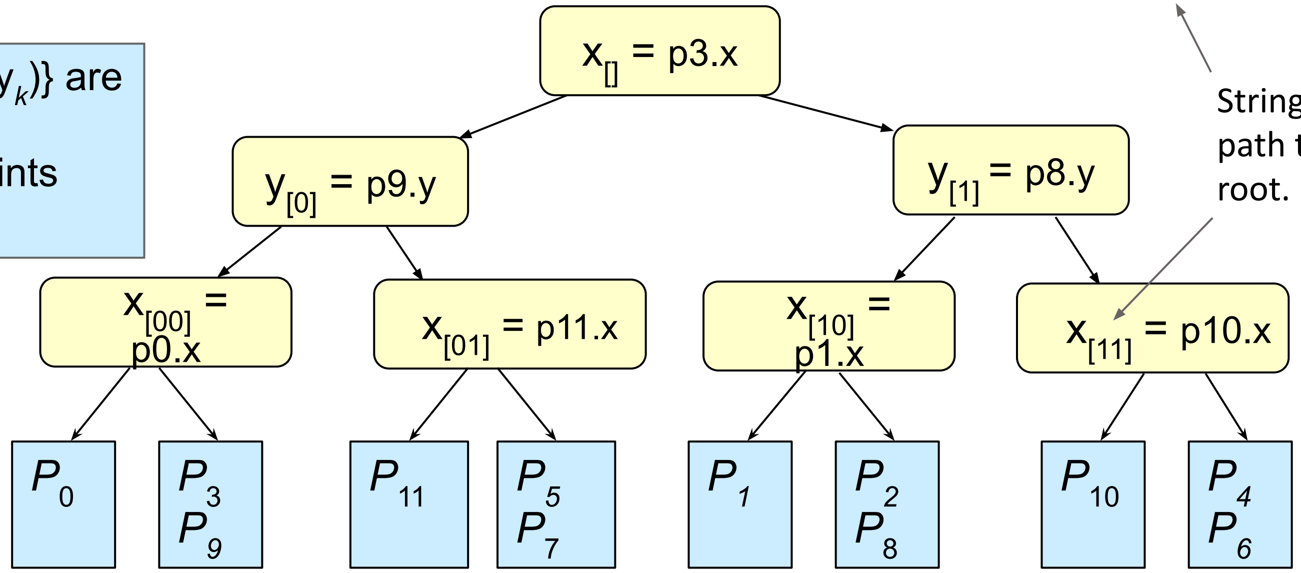
- A 2D k-d tree recursively splits a two-dimensional region into sub-regions using alternating horizontal and vertical lines.
- Usually partition at or near median
- Earlier binary search tree was a 1D k-d tree.
- Generalising to higher dimensional data sets, a *k*-d tree is a binary partitioning tree where the split operation is performed cyclically along each axis direction using axis-aligned hyper-planes.



K-d tree example



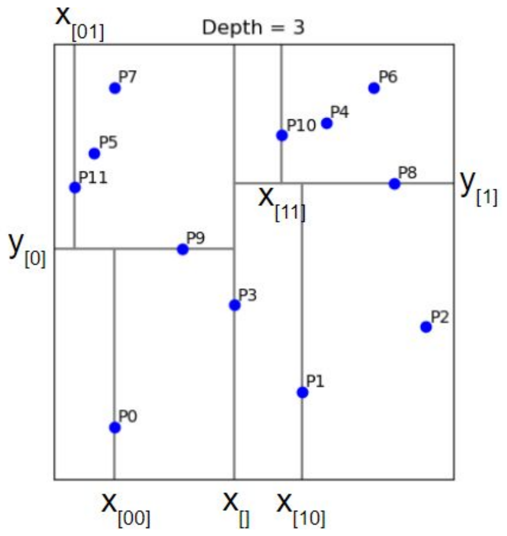
Points $\{(x_k, y_k)\}$ are all at leaves (multiple points per leaf).



Strings in brackets are the path to the node from the root. Empty for root.

k-d trees: splitting planes

- $x_{[]}$: The median of points in the x-direction
- $y_{[0]}$: The median along y-direction, of points on the left side of $x_{[]}$
- $y_{[1]}$: The median along y-direction of points on the right side of $x_{[]}$
- The sub-regions are further split until
  - a region has less than some number of points, or
  - a maximum depth is reached



NOTE: Computation of median requires that the tree be restructured whenever a point is added/deleted/moved. However, a *k*-d tree is usually restructured only if there is a significant change in the underlying data.

Remember: "median" is being used very loosely throughout to mean "sort-of middle point"

2D k-d tree building algorithm

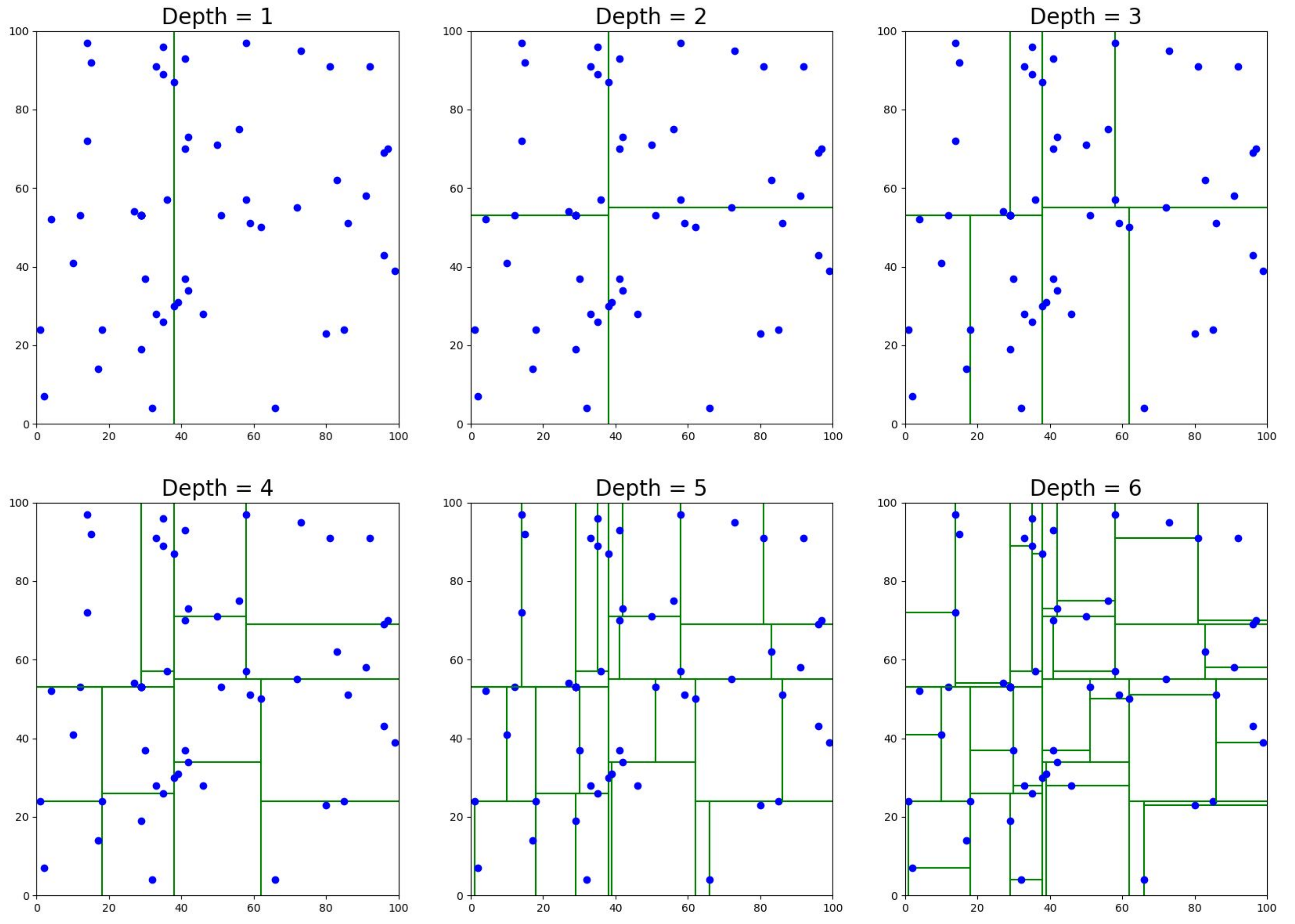
A relatively simple extension of the 1D k-d tree building algorithm.

```
function kd_tree(points, depth=0):
    if “sufficiently few” points, or maximum depth reached:
        return Leaf(points)
    else:
        axis = depth % 2  # 0 for x, 1 for y
        sort points along axis
        mid = len(points) // 2
        xy = points[mid - 1][axis] # x or y value of divider
        left = kd_tree(points[:mid], depth + 1)
        right = kd_tree(points[mid:], depth + 1)
        return Node(axis, xy, left, right)
```

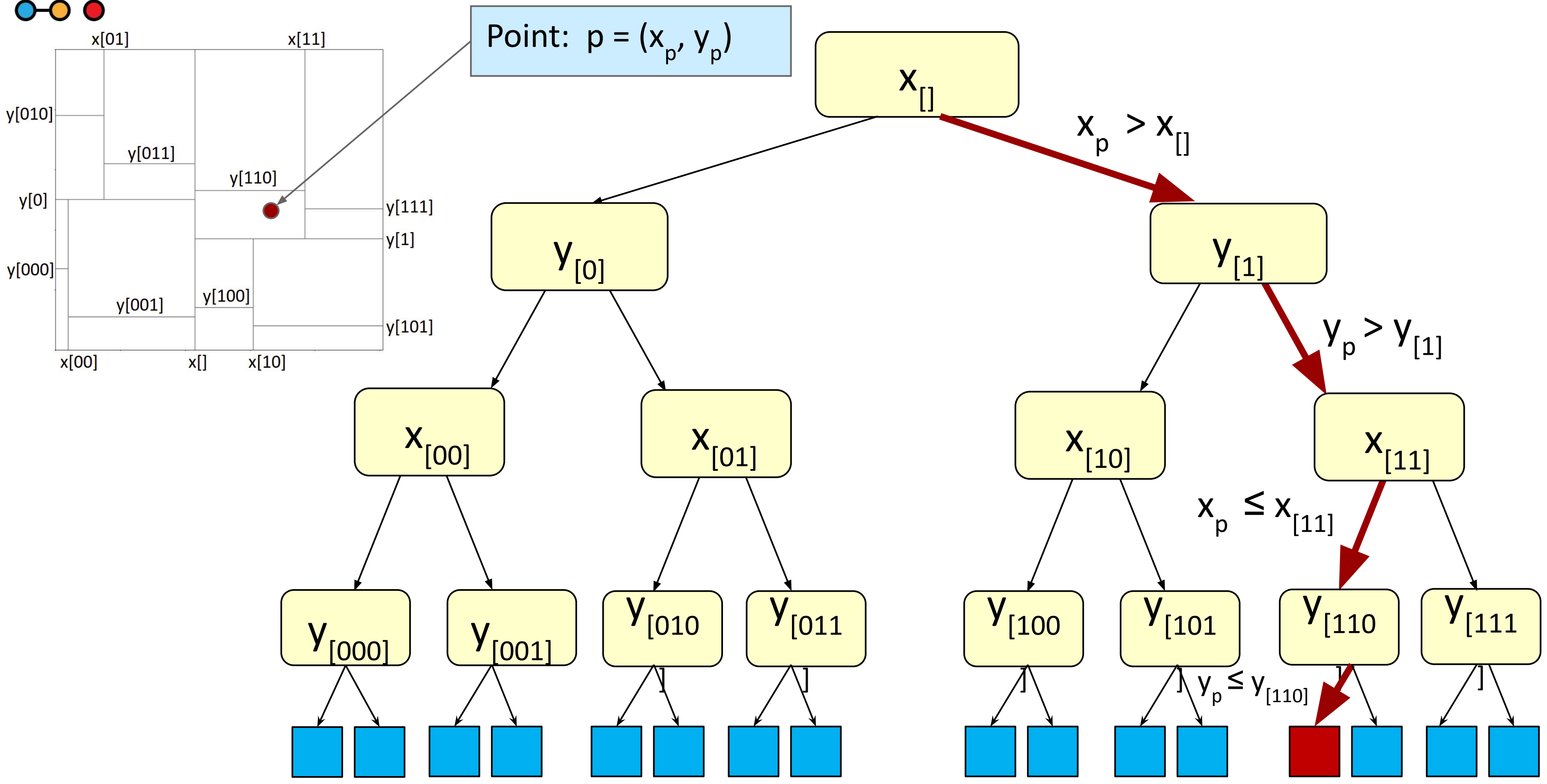
Questions:

1. What is the order of complexity?
2. Can you see how to improve it in average case?

A larger example



Locating a specific point in a *k*-d tree

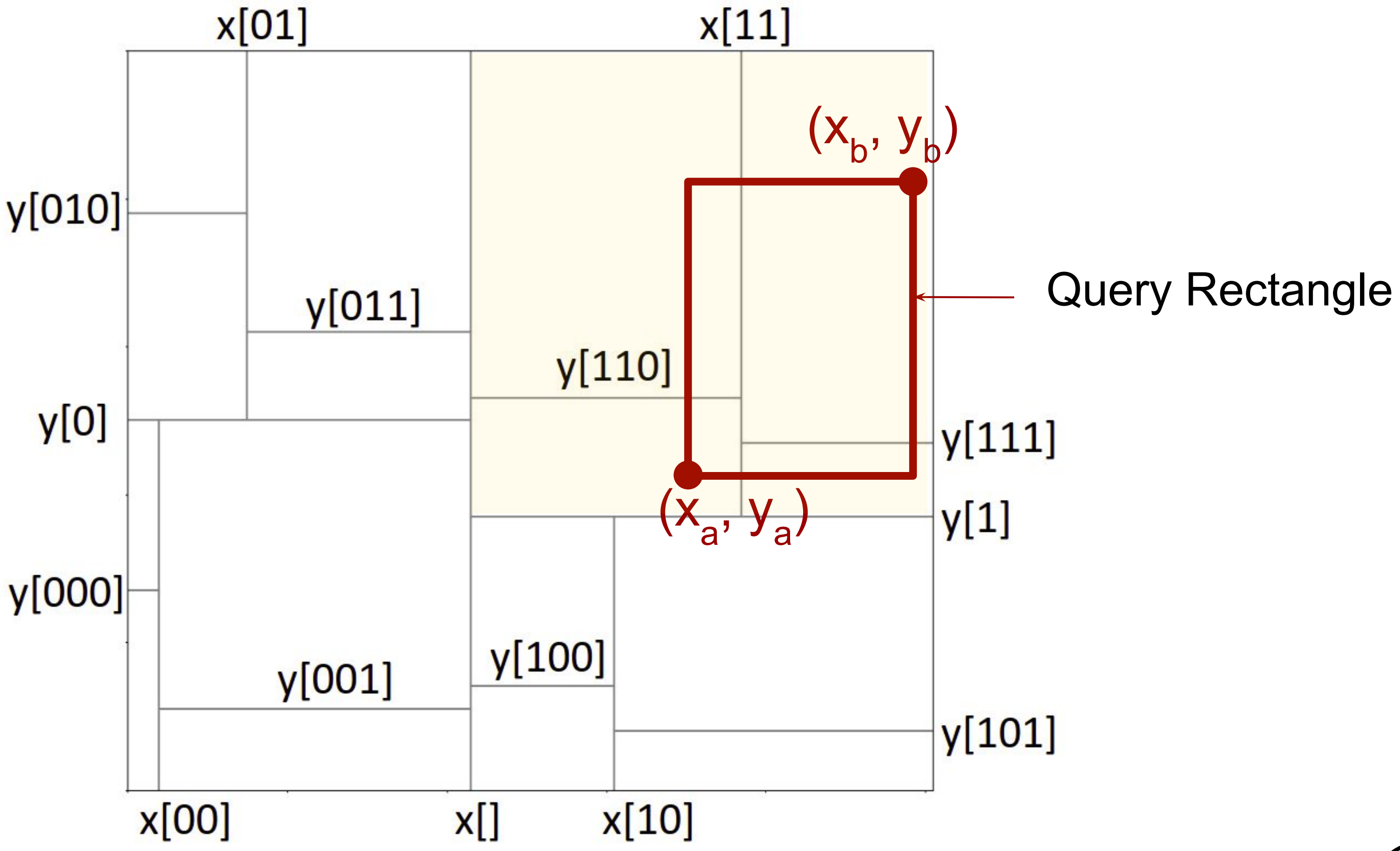


Compare the value at a node with the corresponding coordinate of the point. If the point's value is larger than the node value, go to right child; otherwise go to left child. At leaf, check all points for a match.

2D range searching using a *k*-d tree

Range: $[(x_a, y_a), (x_b, y_b)]$

How do we limit the search to only the overlapping nodes of the *k*-d tree?



Range search algorithm

A relatively simple extension of the 1D k-d tree search.

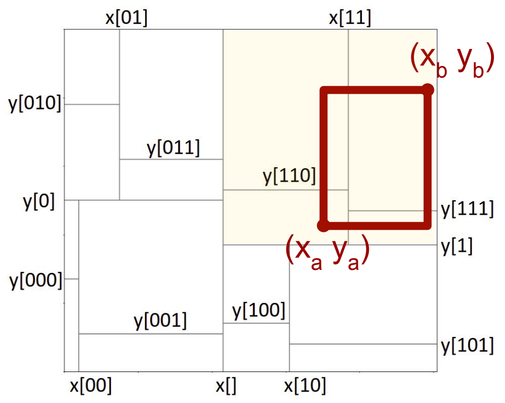
```
function points_in_range(node, query_rectangle):
    if node is leaf:
        matches = all points in node.points lying within rectangle
    else:
        matches = []
        if rectangle intersects left (or bottom) half space:
            matches += points_in_range(node.left, query_rectangle)
        if rectangle intersects right (or top) half space:
            matches += points_in_range(node.right, query_rectangle)
    return matches
```

A “half-space” is all points on one side of a given line. So for a node with a horizontal divider line, we define the bottom half space as all points **on or below** that line, the top half space as all points **on or above** it. Similarly for a vertical divider with left and right half spaces.

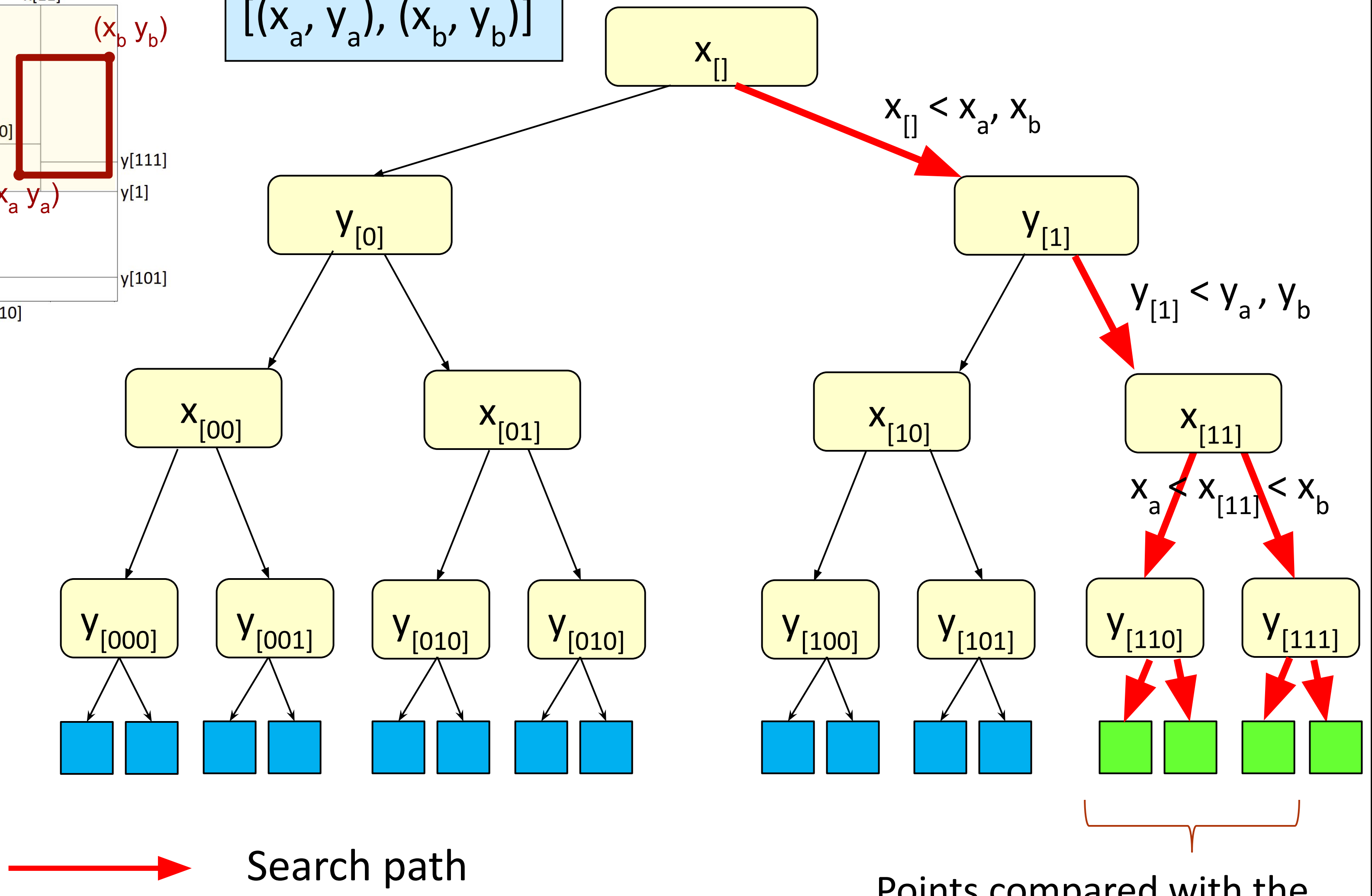
“On-the-line” points

- With the binary search tree, we assumed no duplicate values.
- We could then ensure that values less than or equal to the node value were in the left sub-tree, while values greater than the node value were in the right sub-tree.
- With 2D points we still assume no duplicates but there may be points with identical x or y values.
  - In the worst case, all points may have same x or y value.
- So some values lying exactly on a split line may land up in one subtree while others land in the other subtree.
- Hence the range search much check both sides of the tree when a boundary of the range rectangle lies on a split line.
  - That’s why the half-space definition has “on the line” points in both half spaces.

Example



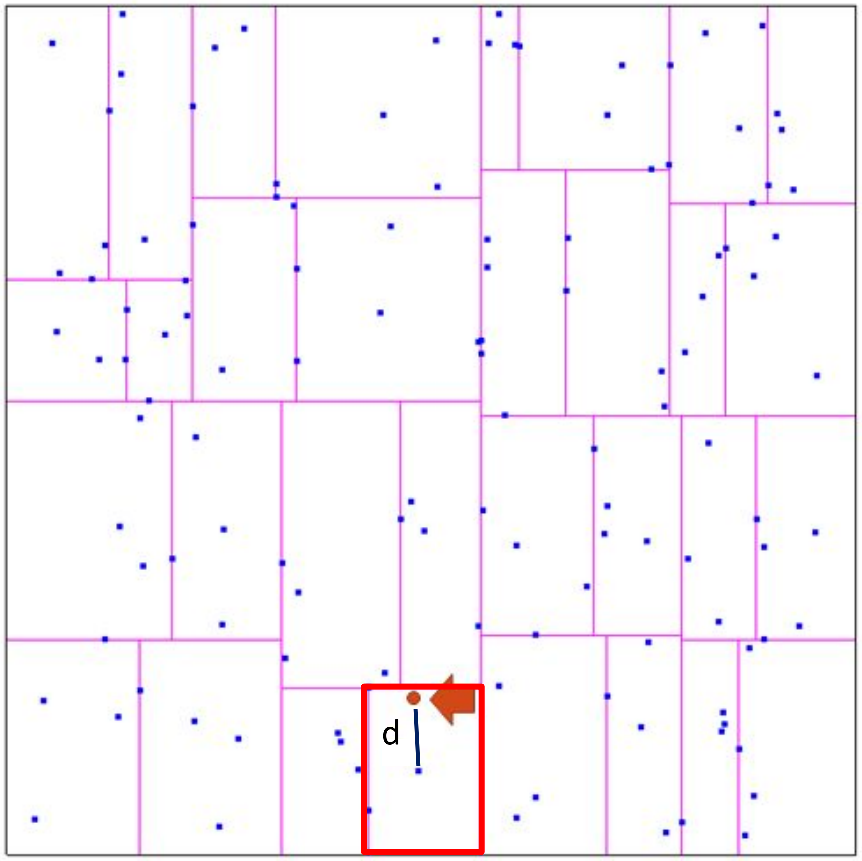
$[(x_a, y_a), (x_b, y_b)]$



Points compared with the range and reported.

k-d Trees: Finding the closest neighbour

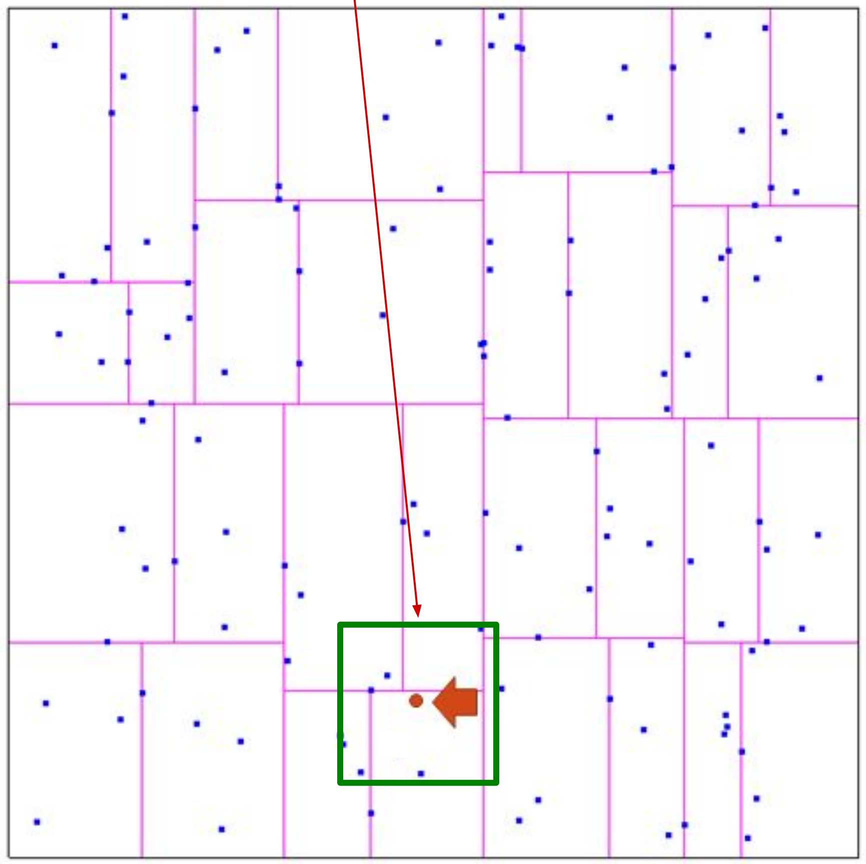
- Problem: Given a point (x_p, y_p) find its closest neighbour in a space partitioned by a *k*-d tree.
- First, find the leaf node containing the point, and compute the distance *d* from the closest neighbour **within that leaf**.



Algorithm continues on next slide

Finding the closest neighbour (cont'd)

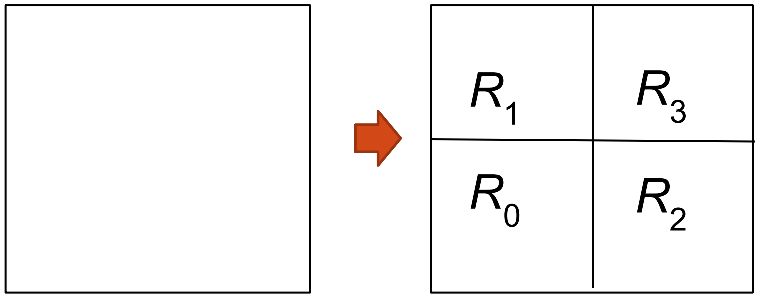
- Use *points_in_range* to find all points within the query square $[(x_p\text{-}d, y_p\text{-}d), (x_p+d, y_p+d)])$
- Linearly search that list.

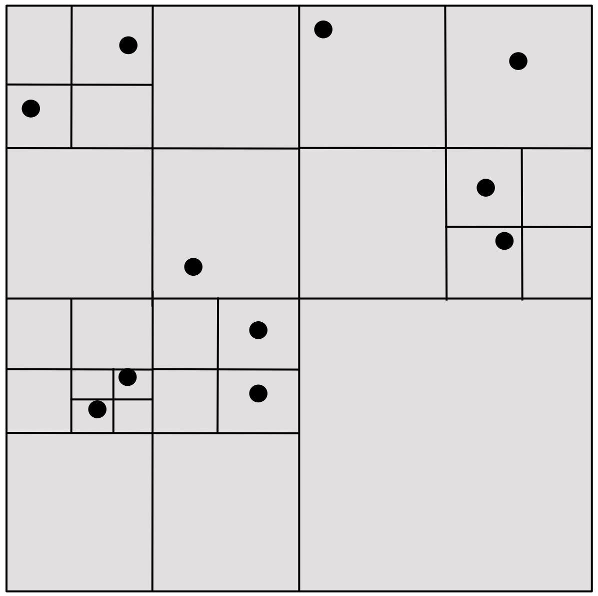


Quadtrees

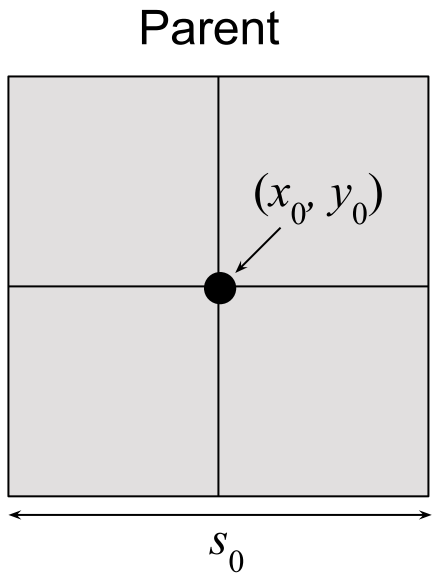
A quadtree is a spatial partition tree constructed as follows:

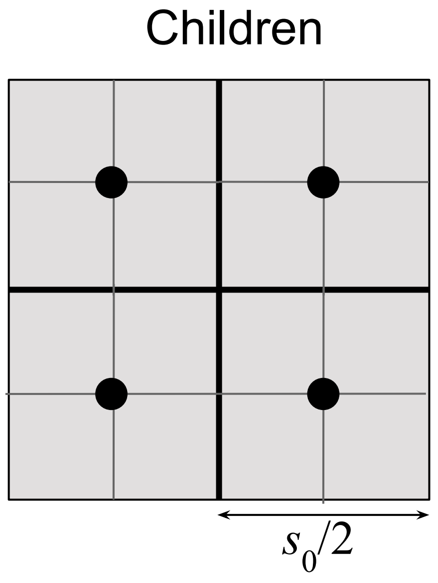
- The root node represents a square region containing all the points.
- Each region R is split into 4 equal squares R_0, R_1, R_2, R_3. The smaller regions are represented by the 4 children of the parent node.
- Regions are recursively subdivided until
  - a region contains no more than a specified number of points, or
  - the maximum depth is reached.



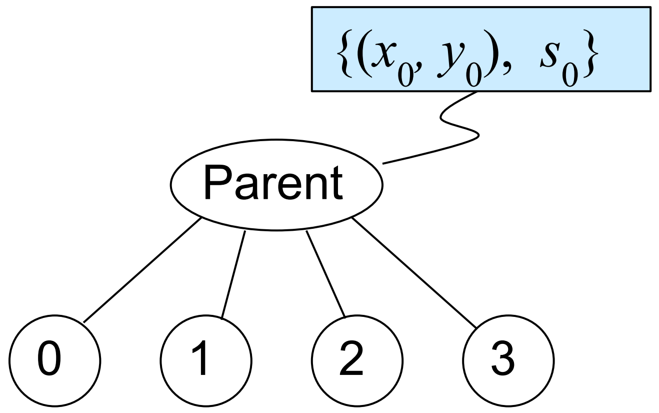


Quadtree Construction





A quadtree node is represented by its coordinates (e.g. centre) and size. These require floating point calculations.



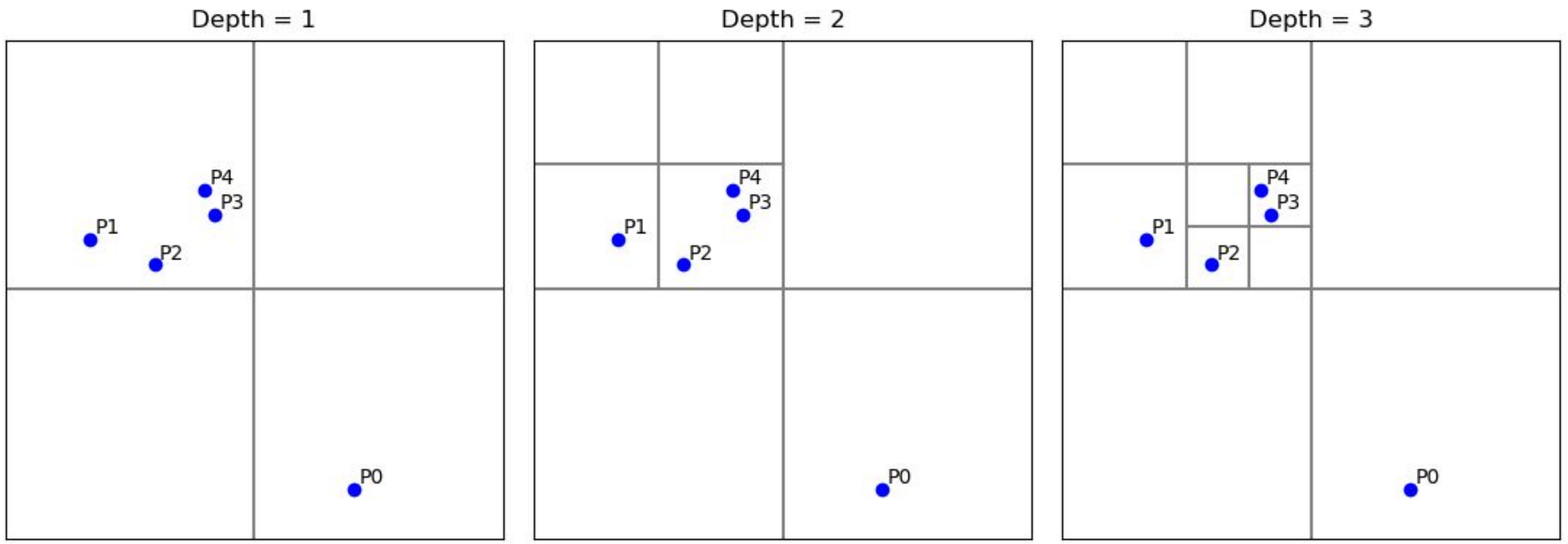
0 (bottom left):	$\{(x_0 - s_0/4,\ y_0 - s_0/4),\quad s_0/2\}$
1 (top left):	$\{(x_0 - s_0/4,\ y_0 + s_0/4),\quad s_0/2\}$
2 (bottom right):	$\{(x_0 + s_0/4,\ y_0 - s_0/4),\quad s_0/2\}$
3 (top right):	$\{(x_0 + s_0/4,\ y_0 + s_0/4),\quad s_0/2\}$

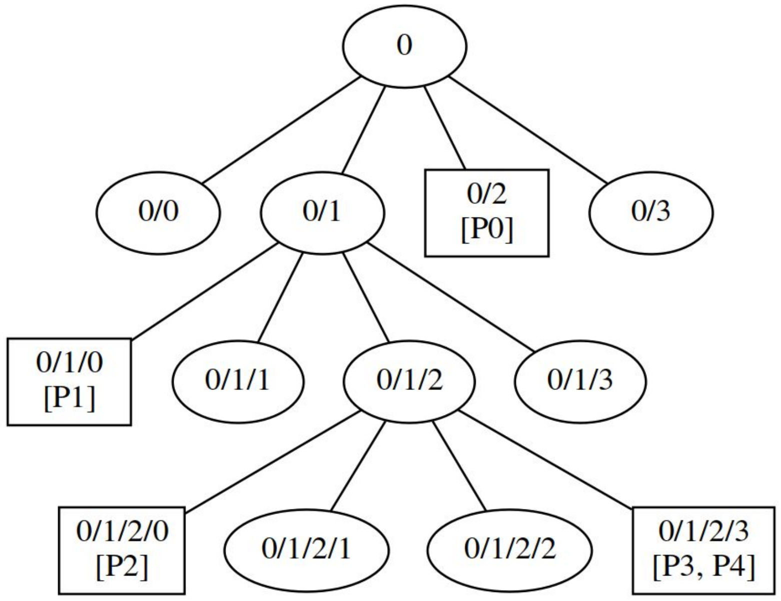
Quadtree build algorithm

```
function quad_tree(points, centre, size, depth=0, max_leaf_points=2):
    points = [p for p in points if in_square(p, centre, size)]
    children = []
    if len(points) > max_leaf_points and depth < max_depth:
        for i in range(4):
            compute child_centre, child_size for child i
            child = quad_tree(
                    points, child_centre, child_size, depth + 1)
            children.append(child)
    return Node(points, centre, size, children)
```

where *in_square* checks if p lies within the given box, including bottom and left boundaries *but not top and right*.

Quadtree example





Node labels are *paths* from root.
Children are numbered as below.
Non-empty leaves are drawn as boxes.

1	3
0	2

Quadtree build complexity

Best case

- Points uniformly distributed
- Results in balanced tree
- T(n) = n + 4 T(n / 4) => O(?)

Worst case

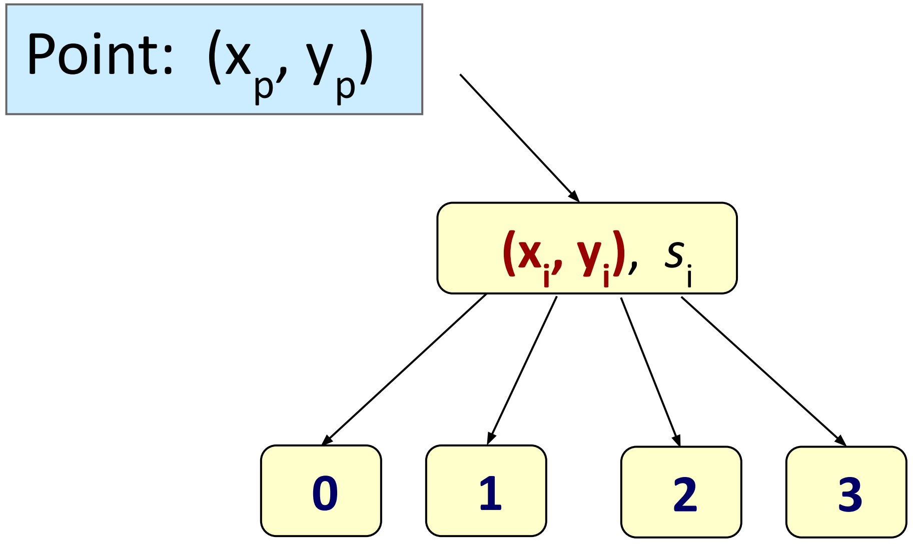
- All points in a very tight cluster
- All points land in same leaf node
- Recursion stops when hits max depth *d*
- Complexity is thus O(?)

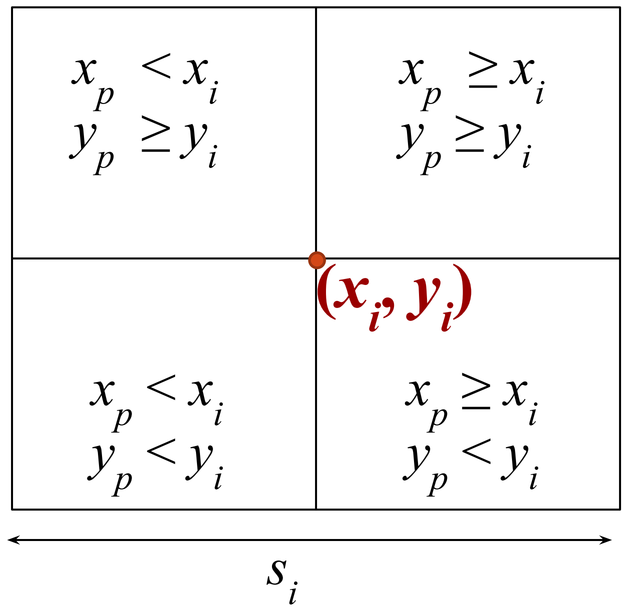
Exercises: see lab

BUT … the “best case” scenario would be better handled by a regular grid.

Locating/Inserting a point in a quadtree

- Traverse the quadtree from the root node. At each node (x_i, y_i), compare the coordinates of the point with x_i, y_i to determine the index of the child node.
- Check for or insert the point when the leaf node is reached.




$x_p < x_i$, $y_p \geq y_i$
$x_p \geq x_i$, $y_p \geq y_i$
(x_i, y_i)
$x_p < x_i$, $y_p < y_i$
$x_p \geq x_i$, $y_p < y_i$
s_i

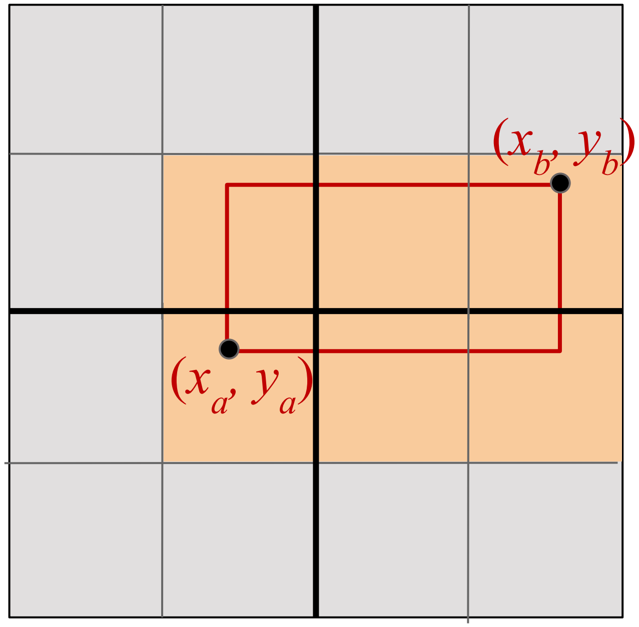
2D Range Search Using Quadtrees

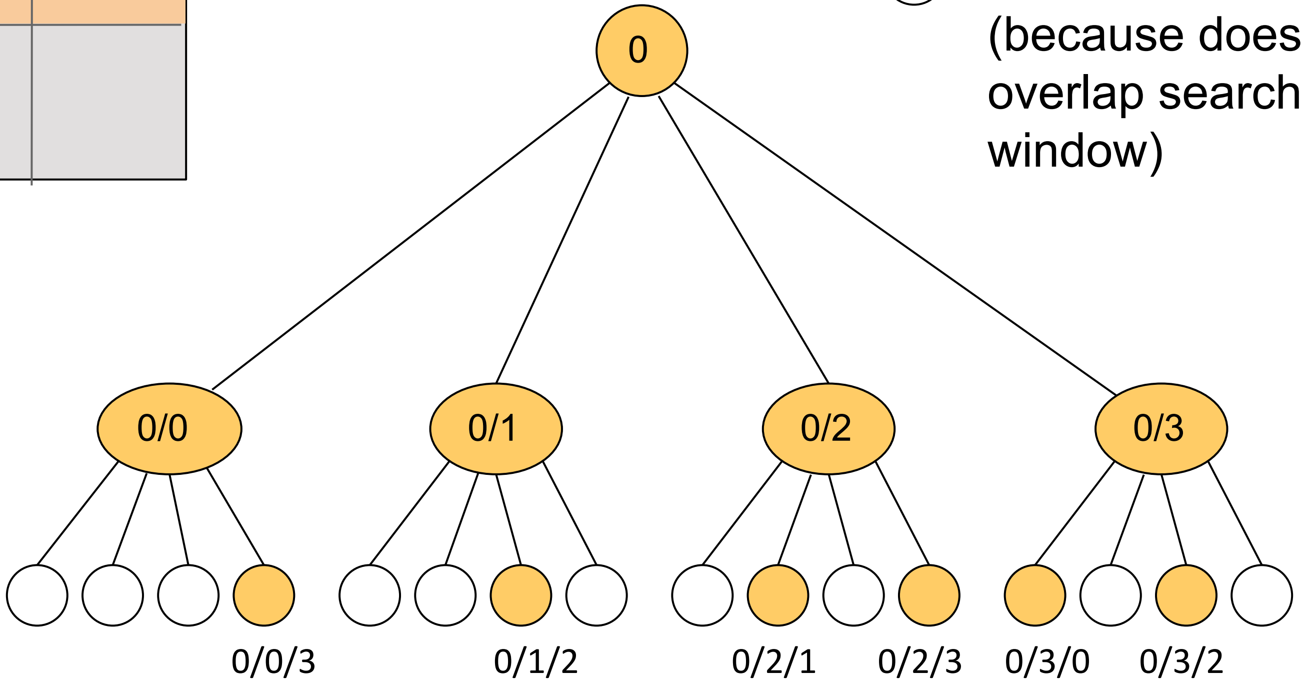
Algorithm:

Starting from root of tree.

```
function points_in_range(node, search_window):
    if node is leaf:
        return [p for p in node.points if p inside search_window]
    else:
        points = []
        for each child:
            if child rectangle overlaps search_window:
                points += points_in_range(child, search_window)
        return points
```

Quadtree Search: Example





● Visited node (because it overlaps search window)

○ Unvisited node (because doesn't overlap search window)

Quadtrees versus k-d trees

- **Quadtrees are 2D only!**
- k-d trees extend trivially to any dimension
- Quadtrees are slightly easier to program (?)
- K-d trees much more efficient with clustered data
  - Shallower
  - Fewer nodes
- Quadtrees are worse than useless in worst case
  - All points in one deep leaf
- Adding points:
  - **Easy to add points to quadtree; expand nodes if necessary**
  - k-d trees need to be rebuilt from scratch when they get too unbalanced

COSC 262 **Algorithms**

Computational Geometry

Part 3: Line Sweep

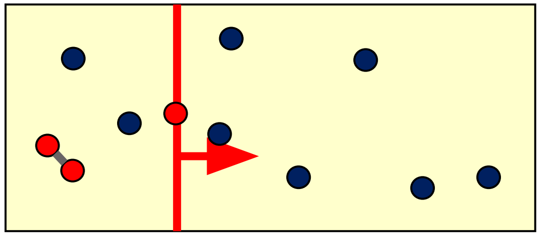
Nice store. How do you keep the floors so clean?
Oh, we hired this dude named Kepler, he's really good. Hard worker. Doesn't mind the monotony. Sweeps out the same area every night.

https://xkcd.com/21/

Richard Lobb & R. Mukundan
richard.lobb@canterbury.ac.nz
Department of Computer Science and Software Engineering
University of Canterbury

Line-sweep algorithms

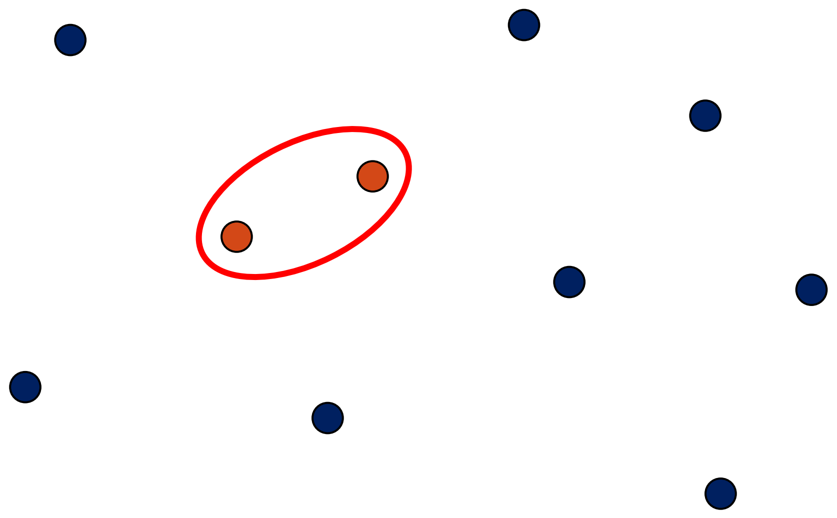
- A powerful approach for solving problems involving geometric objects on a plane.
- *Simulates* the sweeping of a vertical line across objects on the *xy*-plane in one direction (no backing up).
    - This is done by **sorting** the points in ascending order of *x*-coordinates, and sequentially visiting them from left to right.
- At any stage in the sweep we know the solution to the problem for all points to the left of the line.
- As each point is added, we check if the new point changes the existing solution.



We look at just one example, ...

The closest point-pair problem

Given a set of *n* points on a plane, find the pair of points *P* and *Q* that are at a minimum distance from each other.



Brute-Force Algorithm: Compare every point with every other point. Number of comparisons = $n(n-1)/2 = O(n^2)$.
Is it necessary to compare every point with all other points?

The 1D closest pair problem

- Consider a one-dimensional problem:

  51 2 45 30 15 37 25 8 59

- The naive approach will take $O(n^2)$ time.
- If we sort the list in ascending order, we can very easily find the closest pair.

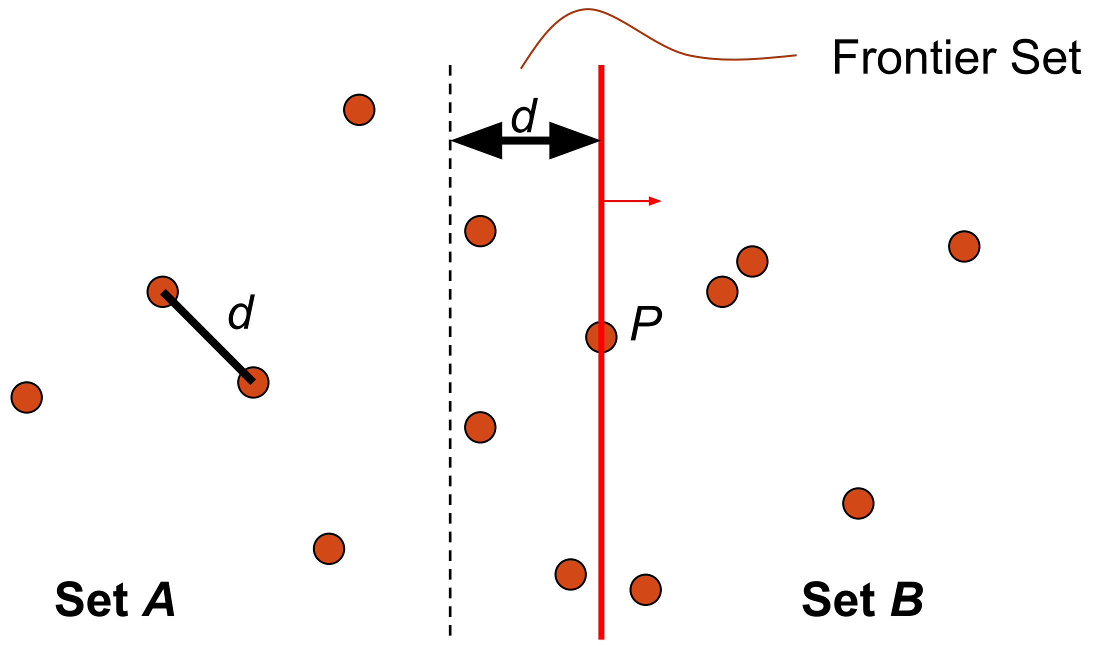
  2 8 15 25 30 37 45 51 59

  - Sorting takes only $O(n \log n)$ time.
- We now just traverse the list from left to right, compute the difference between the current value and the next value, and store the minimum difference found so far. We also store the index of the value that gave the minimum difference.

2D closest pair problem

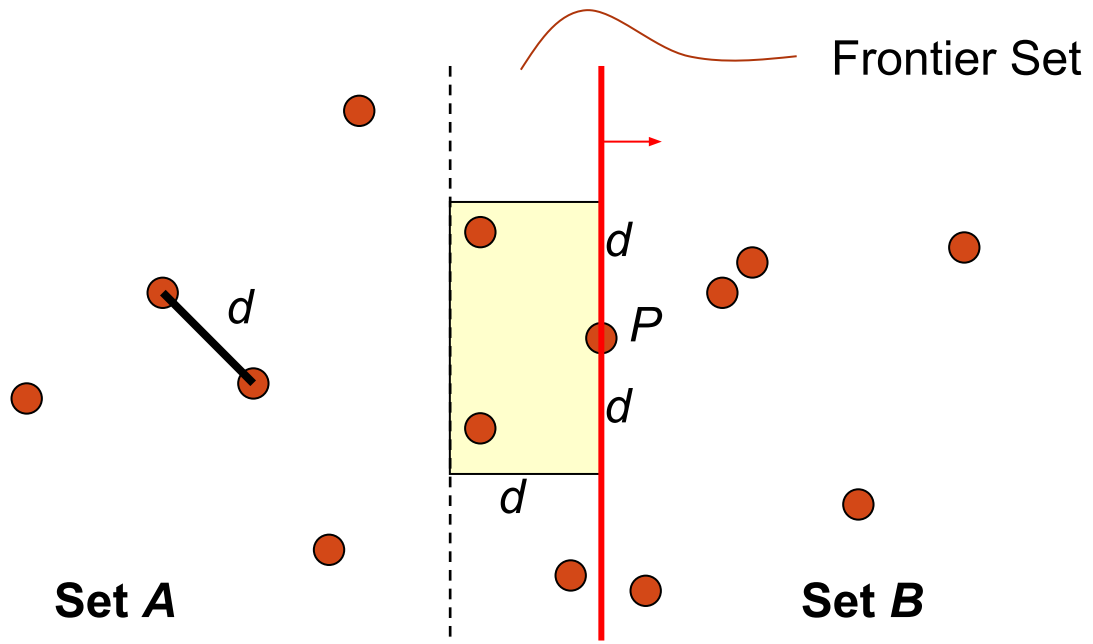
Let us visit the points from left to right. Assume that the closest distance found so far was *d*. Suppose that we are now at a point *P*.

- Imagine a vertical line through *P*, and a "frontier set" defined by horizontal distance *d* on the left side of the line. On the left of the frontier set is "Set *A*", and on its right "Set *B*"



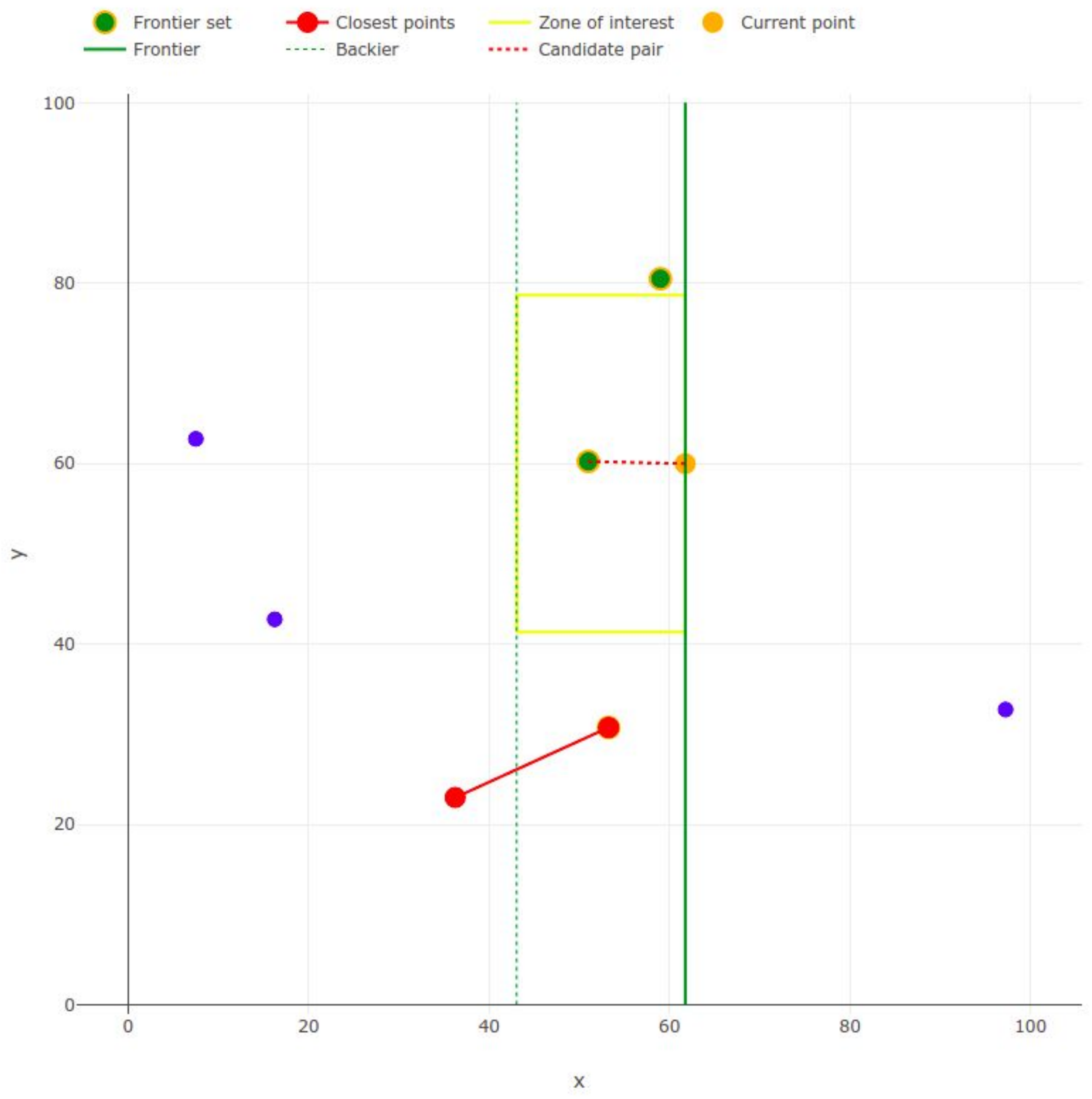
2D closest pair problem: key ideas

- At all times we want the solution involving only points on or to the left of the sweep line.
- As we add each point P, we need to check against only the points inside a 2*d* × *d* rectangle within the frontier set.



Closest pair algorithm visualiser

See https://lobb.nz/closestpointpairvisualiser/closestpointpair.html



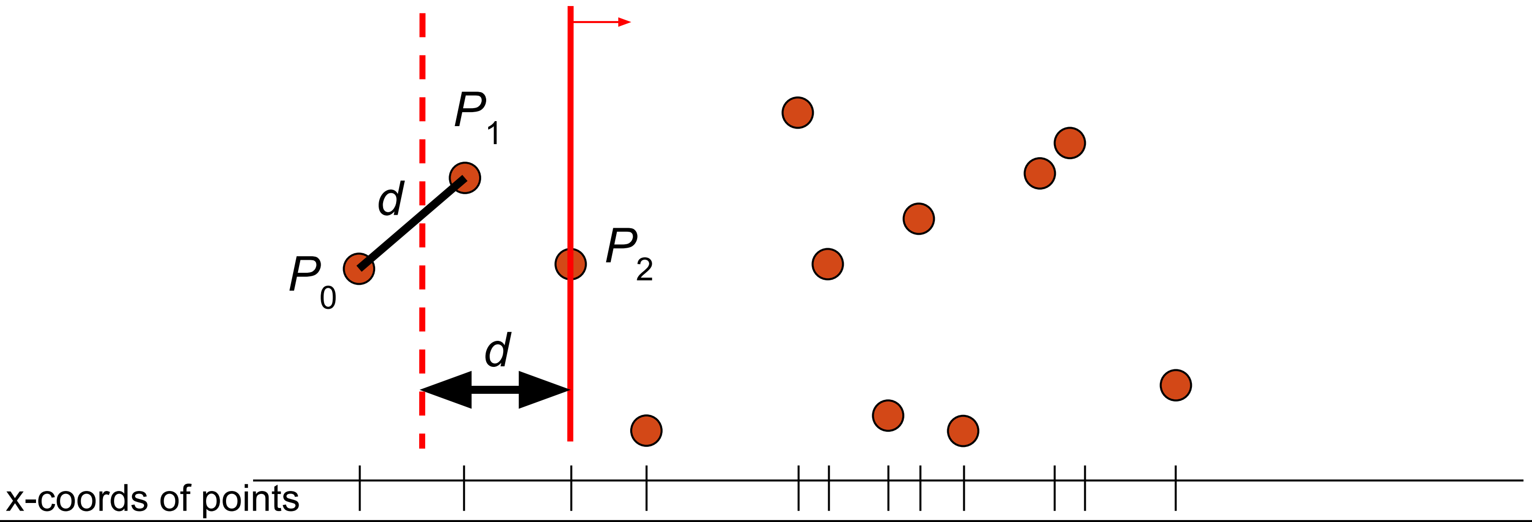
Data structures

Point list:

- Points sorted by their *x*-coords in ascending order.

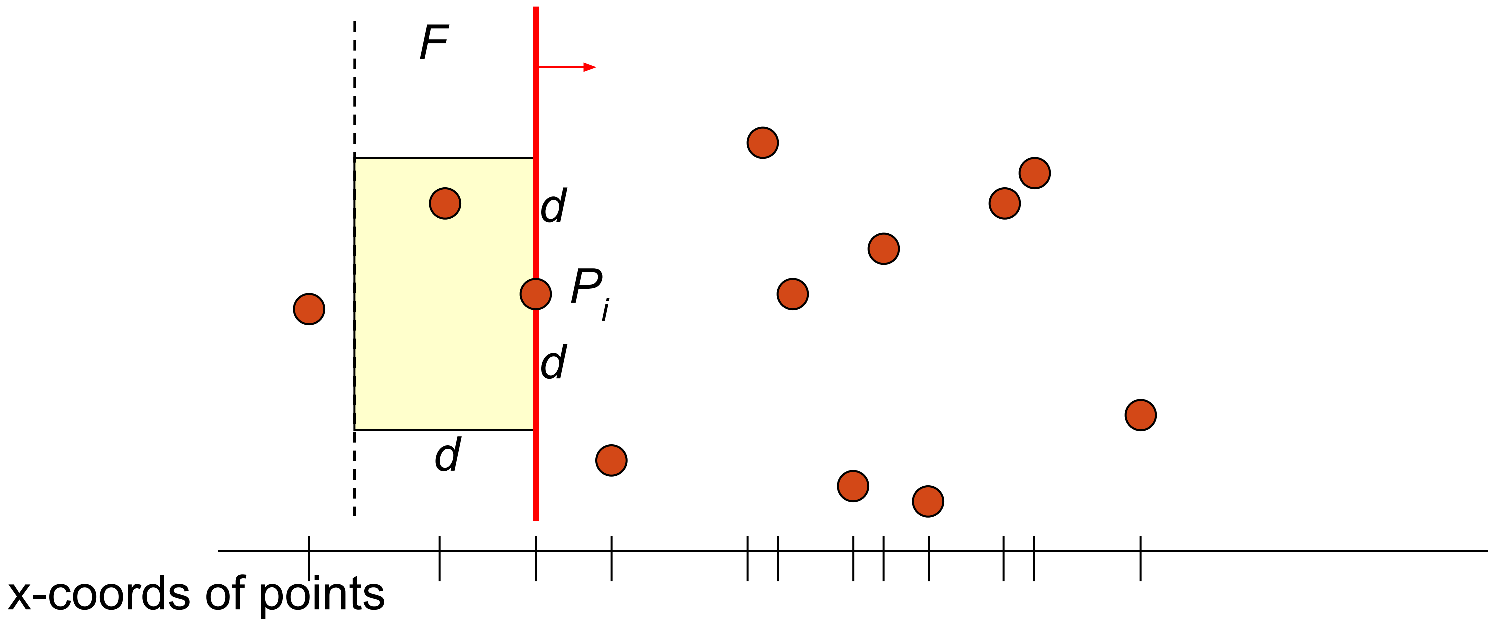
State to maintain:

- The current sweep line position (index into point list). Initialised to P_2
- The best point pair so far, initialised to (P_1, P_0), and the distance d between those points. [Need floating point numbers here]
- The frontier set, *F*, including the current point; initialized to {P_0, P_1 , P_2}. In the example below, P_0 will be removed after distance comparison.



Actions at each step in the algorithm

- Get the current point P_i from the sorted point list and add it to the frontier set F. ($F \leftarrow F \cup \{P_i\}$)
- Remove from F all points whose horizontal distance from the current point P_i is greater than d.
- Compare the current point P_i with points in F within a vertical distance d (see next slide). If a closer point is found, update the solution set and the d value.

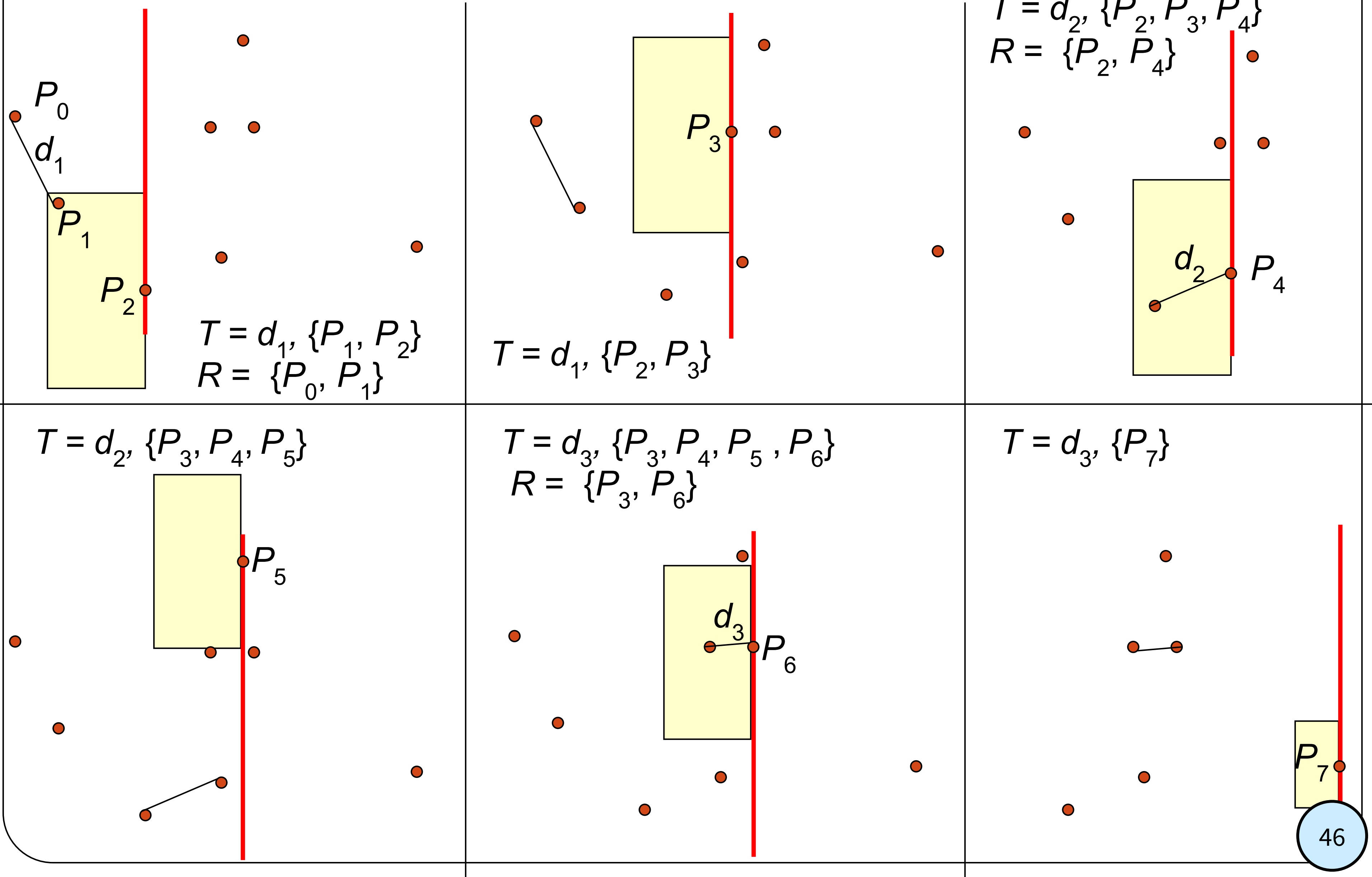


Representing the frontier set *F*

- Previous slide contains the step:
    - "Compare the current point P_i with points in *F* within a vertical distance *d*"
- To make this efficient, F must be sorted by y.
- F must also support efficient addition and removal of points
    - Inserting/deleting in a simple sorted list is O(n). Want O(log n)
- A self-balancing binary tree (e.g. AVL tree) is one solution.
- Python has no such data structure built in
    - I used the 3rd party *SortedContainers* module
        - Includes a *SortedList* class with (approx) desired behaviour
    - See http://www.grantjenks.com/docs/sortedcontainers/
    - `pip install sortedcontainers`

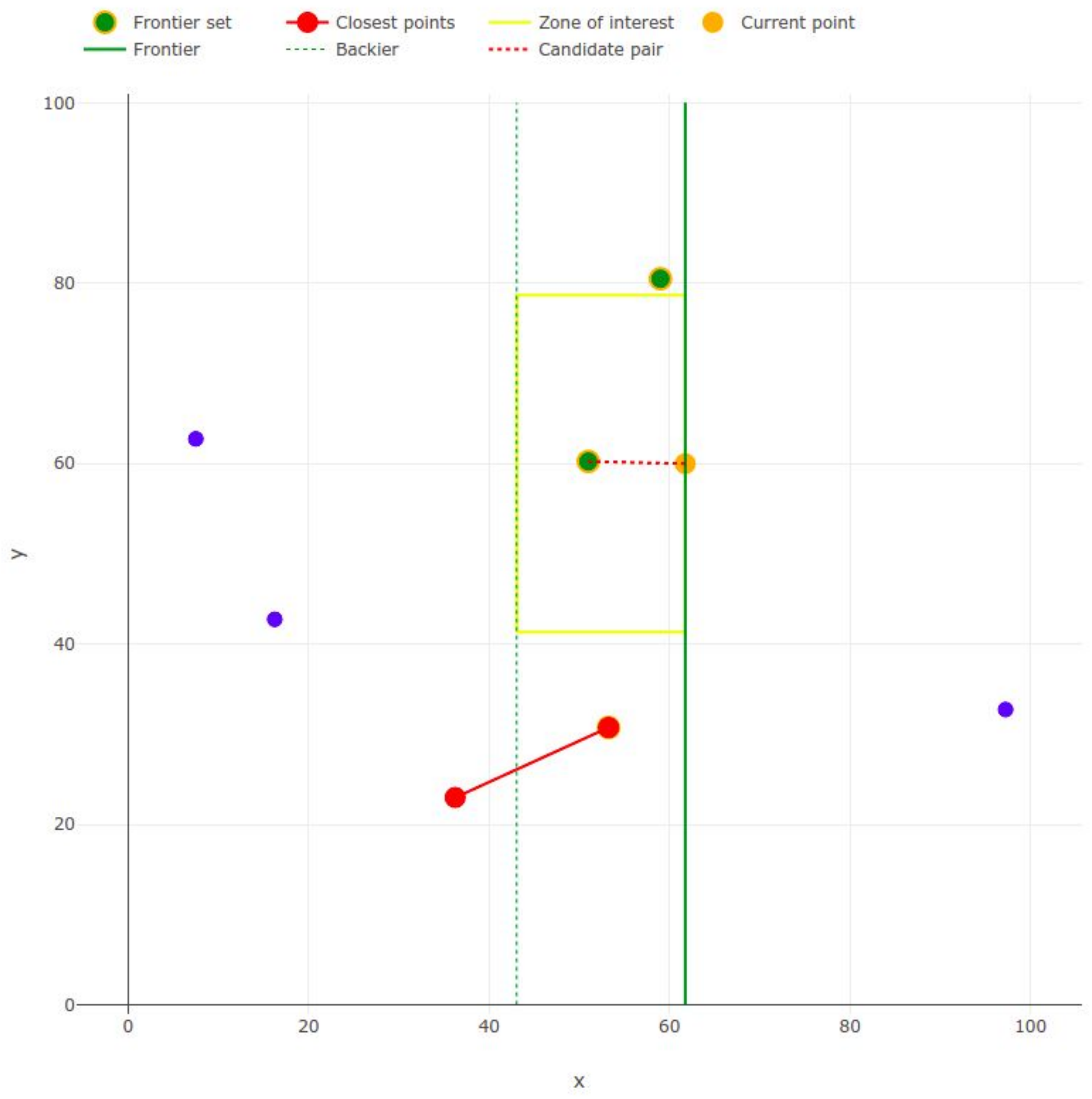
Closest Pair: Example

See https://lobb.nz/closestpointpairvisualiser/closestpointpair.html



Closest pair algorithm visualiser

See https://lobb.nz/closestpointpairvisualiser/closestpointpair.html



Algorithm complexity

- Sorting the points by their *x*-coordinate values takes *O*(*n* log *n*) time.
- Each point is added once and removed once from the frontier set *F:* takes *O*(*n* log *n*) time in total.
- The search for the closest pair at each step takes *O*(log *n*) time
  - Time to select range of points within ±d of P in *y* is O(log n)
  - Number of points within that range is O(1)
    - Tricky to prove mathematically but results from the fact that excluding P itself, no pair of points within the frontier set are closer than *d*
  - Hence total time for searching is also *O*(*n* log *n*)
- Thus we can conclude that the closest pair problem can be solved using a line-sweep algorithm in *O*(*n* log *n*) time.

Line-sweep algorithms in general

- Line-sweep algorithms are useful in significantly reducing the number of comparisons between geometric primitives using information gathered at each position of the sweep line.
- Other examples include line-segment intersection (Bentley-Ottmann algorithm) and Voronoi Diagram computation (Fortune’s algorithm)
- The pre-processing step often involves the sorting of points taking $O(n \log n)$ time. Remaining computations are generally of similar or lower complexity.
- Although not normally classified as such, Graham Scan can be thought of as a line-sweep algorithm in polar coordinates rather than cartesian coordinates.
    - And there’s a variant called Andrew’s monotone chain algorithm that sorts in x - definitely a line-sweep algorithm.