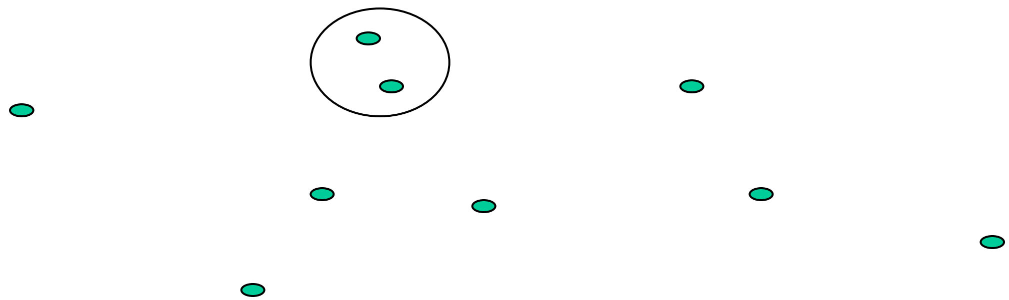
Problem 4: Finding a closest pair of points

Problem: given a set P of n points in the plane, find a pair whose distance from each other is as small as possible.

e.g.



- **Naïve algorithm**
  - check each pair - $O(n^2)$ complexity
- **Divide and conquer algorithm**
  - $O(n \log n)$ complexity
  - Based on sorting

Example of a divide and conquer algorithm: MergeSort

```
int [] a = {...};
```

```
public void mergeSort(int i, k) {
   if (i < k) {
      int j = (i+k)/2;
      mergeSort(i,j);
      // a[i..j] sorted
      mergeSort(j+1,k);
      // a[j+1..k] sorted
      merge(i,j,k);
      /* a[i..j] merged with a[j+1..k]
       * i.e. a[i..k] sorted */
   }
}
```

divide

recursive calls

combine

conquer!

To sort an array `a`, make the call `mergeSort(0, a.length-1)`

```
/** Merges sorted segments a[i..j] and
 *  a[j+1..k] to give a[i..k] sorted */
private void merge(int i, int j, int k) {

  int index1, index2, index3;
  int [] temp = new int[a.length];

  index1 = i;
  index2 = j+1;
  index3 = i;
  while ( index1 ≤ j && index2 ≤ k )
    if (a[index1] ≤ a[index2])
      temp[index3++] = a[index1++];
    else
      temp[index3++] = a[index2++];
  if ( index1 ≤ j )
    temp[index3..k] = a[index1..j];
  else if ( index2 ≤ k )
    temp[index3..k] = a[index2..k];
  a[i..k] = temp[i..k];
}
```

Analysis of mergesort

Let $f(n)$ be worst-case complexity. Then

$$f(n) \le 2\, f(n/2) + cn \qquad (n > 1)$$

$$f(1) = d$$

recursive calls

merge

where c and d are constants.

Solving the recurrence relation

Assume $n = 2^k$ for simplicity, so $k = \log_2 n$

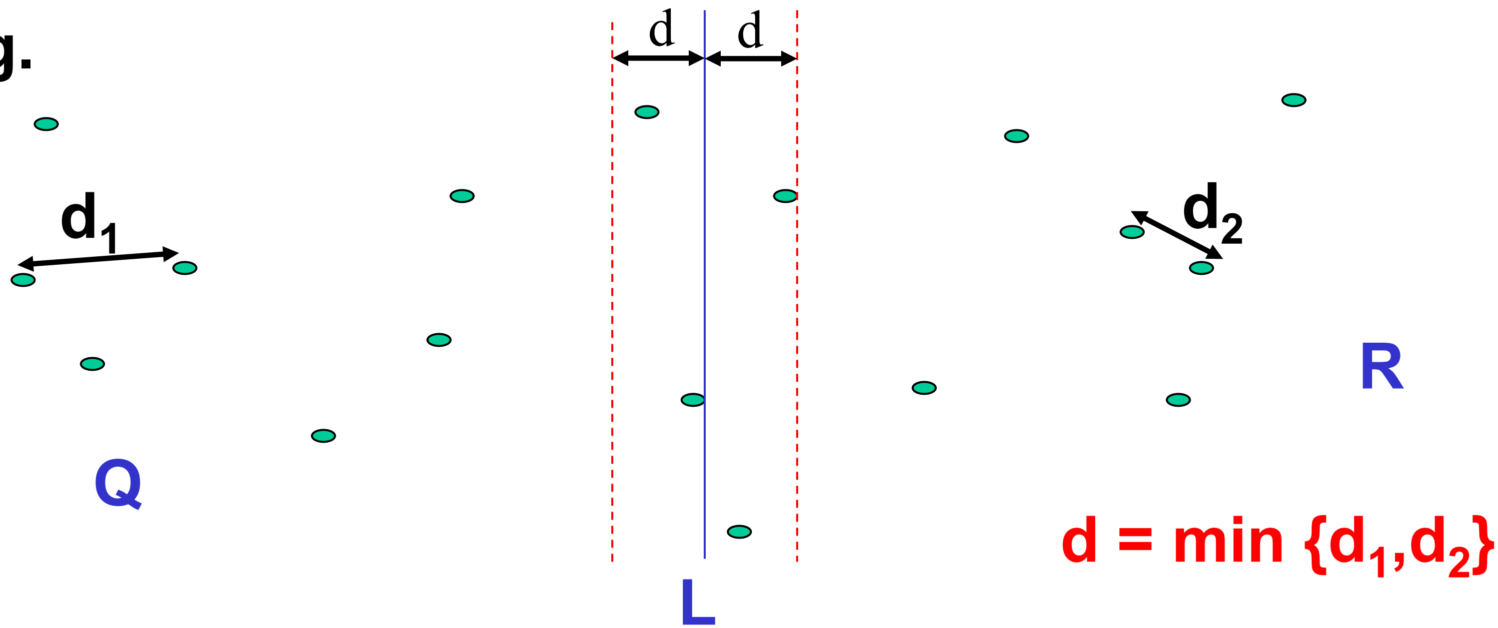
$$
\begin{aligned}
f(n) &\le 2\, f(n/2) + cn \\
&\le 2^2 f(n/2^2) + 2cn \\
&\le 2^3 f(n/2^3) + 3cn \\
&\quad \cdots\cdots\cdots \\
&\le 2^k f(n/2^k) + kcn \\
&= d\, n + c\, n \log_2 n \\
&= O(n \log n)
\end{aligned}
$$

Closest pair algorithm - basic idea

- sort P by x-coordinate once at the outset
- divide P into two equal-sized subsets Q, R based on x-coordinate
- solve for each subset recursively
- combine: closest pair (x,y) satisfies either

(i) $x \in Q, y \in Q$ (solved already) or (ii) $x \in R, y \in R$ (solved already) or
(iii) $x \in Q, y \in R$ (distance apart $\leq d$)

e.g.

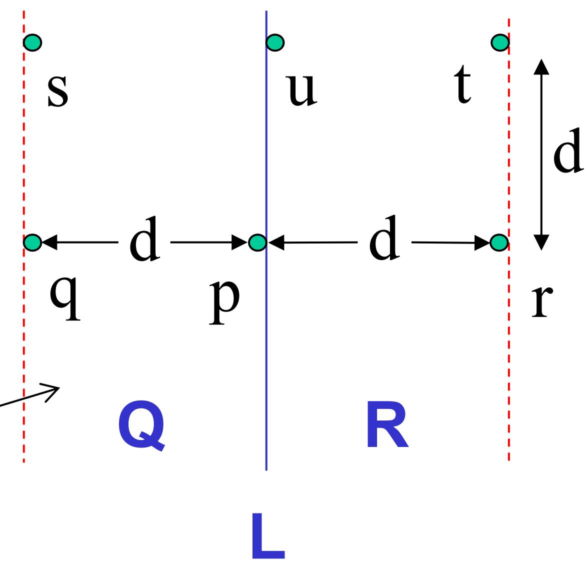


$d = \min \{d_1, d_2\}$

Taking care of case (iii): Closest pair (x,y) in P satisfies $x \in Q$, $y \in R$

- **Can eliminate points distance > d from L**
  - **but in the worst case this might not eliminate any points!**
- **If we sort the remaining points on their y coordinate, any such point can be at distance $\leq$ d from only a small number of its successors in this list**
  - **question: how many successors do we need to check?**
  - **answer: only 5 - see diagram**

With points ordered p,q,r,s,t,u this is the only way a point 5 ahead of p could be a candidate. A point 6 ahead in the list could never be.



call this region the 'strip'

Sorting points in the strip by y-coordinate

- Need to sort all points in the strip in increasing order of their y-coordinates
- If we do this every time we "combine", it leads to an $O(n \log^2 n)$ algorithm – but we can do better than this
- Trick: mimic mergesort!
  - Assume that the points in Q are sorted in increasing order of y-coordinate (when we solve Closest Pair on Q)
  - Assume that the points in R are sorted in increasing order of y-coordinate (when we solve Closest Pair on R)
  - At the "Combine" step, merge the two sorted sets to get all points in the strip sorted in increasing order of y-coordinate
- This leads to a faster algorithm

Closest pair algorithm (to find shortest distance)

```
public double closestPair(PointSet p) {
/** Input: a PointSet p
  * Output: d, the distance between
  * the closest pair of points in p */
  sortOnXCoord(p); // sort points in p on x-coordinate
  return cPRec(p, 0, p.length-1);
}
```

```
private double cPRec (PointSet p, int i, int k) {
/** assumes p[i..k] sorted on x-coordinate;
  * returns the distance between a
  * closest pair of points in p[i..k];
  * also returns, in p[i..k], the points
  * initially in p[i..k] sorted on y coordinate
  */

  double d;
```

```
  if (i == k) d = Double.MAX_VALUE;
  else
  { int j = (i+k)/2;    // mid-point of p[i..k]
    double mid =(p[j].x + p[j+1].x))/2.0;
                        // x coord of mid-line
    double d1 = cPRec(p, i, j);
                        // p[i..j] sorted on y coord
    double d2 = cPRec(p, j+1, k);
                        // p[j+1..k] sorted on y coord
    merge(p, i, j, k); // p[i..k] sorted on y coord
    d = Math.min(d1, d2);
    PointSet s = filter(p, i, k, d, mid);
                        // the points in the "strip"
    int m = s.length;   // no. of points in s
    for (int a=0; a < m-1; a++)
      for (int b=a+1; b <= Math.min(a+5,m-1); b++)
        if ( dist(s[a], s[b]) < d )
          d = dist(s[a], s[b]);
  }
  return d;
}
```

Subsidiary methods

```
private PointSet filter(PointSet p,
                        int i, int k,
                        double d, double z);
/** returns a PointSet containing points in p[i..k] with
  * x-coord within d of z; preserves relative order */


private double dist(Point2D.Double a, Point2D.Double b)
/** returns the distance between the points a and b */
```

Returning an actual closest pair of points

Every time `d` is updated, store the two points `s[a]` and `s[b]` that were responsible for the update being made

Closest pair algorithm - analysis

- Initial sort on x-coordinate is $O(n \log n)$
- When dealing with an array of length n, `merge` and `filter` are both $O(n)$
- The nested for loops contribute $O(n)$
  - the outer loop is executed m ($\leq n$) times, and for each of these, the inner loop is executed ≤ 5 times
- Let $f(n)$ be worst-case complexity. Then

$$f(n) \leq 2\, f(n/2) + cn \quad (n > 1)$$
$$f(1) = d$$

recursive calls

merge, filter, etc.

where c and d are constants.

- So $f(n) = O(n \log n)$ (as for Mergesort)