Strassen's Matrix Multiplication Algorithm

- The standard method of matrix multiplication of two $n \times n$ matrices takes $O(n^3)$ operations.
- Strassen's algorithm is a *Divide-and-Conquer* algorithm that is asymptotically faster, i.e. $O(n^{\lg 7})$.
- The usual multiplication of two 2×2 matrices takes 8 multiplications and 4 additions. Strassen showed how two 2×2 matrices can be multiplied using only 7 multiplications and 18 additions.

Motivation

- For 2×2 matrices, there is no benefit in using the method.
- To see where this is of help, think about multiplication two $(2k) \times (2k)$ matrices.
- For this problem, the scalar multiplications and additions become matrix multiplications and additions.
- An addition of two matrices requires $O(k^2)$ time, a multiplication requires $O(k^3)$.
- Hence, multiplications are much more expensive and it makes sense to trade one multiplication operation for 18 additions.

Algorithm

Imagine that A and B are each partitioned into four square sub-matrices, each submatrix having dimensions $\frac{n}{2} \times \frac{n}{2}$.

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

, where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Strassen's algorithm

Strassen "observed" that:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{bmatrix}$$

, where

$$\begin{aligned}
P_1 &= A_{11}(B_{12} - B_{22}) \\
P_2 &= (A_{11} + A_{12})B_{22} \\
P_3 &= (A_{21} + A_{22})B_{11} \\
P_4 &= A_{22}(B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
P_7 &= (A_{11} - A_{21})(B_{11} + B_{12})
\end{aligned}$$

Complexity

- $T(n) = 7T(\frac{n}{2}) + cn^2$, where c is a fixed constant. The term cn^2 captures the time for the matrix additions and subtractions needed to compute $P_1, ..., P_7$ and $C_{11}, ..., C_{22}$.
- The solution works out to be:

$$T(n) = \Theta(n^{lg7}) = O(n^{2.81}).$$

- Currently, the best known algorithm was given by Coppersmith and Winograd and has time complexity $O(n^{2.376})$.

Closest Pair Problem

- Given a set of n points in the plane, determine the two points that are closest to each other.
- An attempt at a simple solution:
  - Project the points onto a line.
  - Sort the points along the line to find the smallest distance.
  - Problem: Projection changes the distance.
- Brute Force Algorithm: Compute the distances $d(p, q)$ for all possible vertex pairs, and select the minimum distance.
- Complexity: $\Theta(n^2)$.

A Divide and Conquer Solution

Closest-Pair (PointSet)

- Split PointSet in half with a vertical line so that half are on left and half are on right;
- Recursively determine closest pair in each half;
- Let d be smallest of those two distances;
- Search along the boundary between the two halves to see if there are any pairs closer than d;

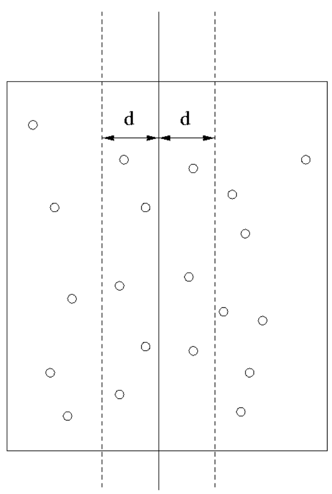
Time Complexity

- Analysis
  - The last step appears to require time $\Theta(n^2)$.
  - Recurrence for total time is $T(n) = 2T(n/2) + \Theta(n^2)$ and $T(1) = 1$.
  - Solution: $T(n) = \Theta(n^2)$.
- The algorithm's last step is the problem.
- How can we do the last step in linear time?

We don't have to examine all the points

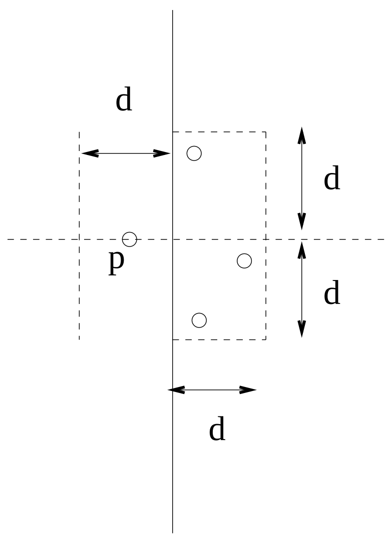
- When we search along the boundary, we don't have to look at all the points in each half.
  - We can ignore any point farther than d from the boundary line. Why?

- But each side may still have $\Theta(n)$ point within distance d from the boundary.



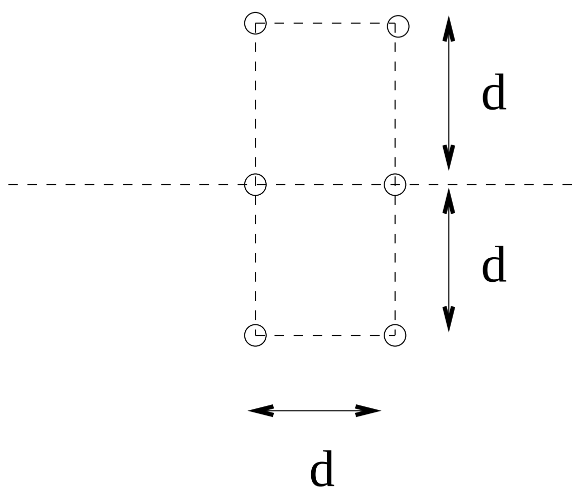
There are just a few points to check on the other side

- For a point q on the right to be close to a point p on the left:
  - q must be within distance d of p.
  - q must fall within a rectangle of size d by $2d$.



Cont'd

- How many points (on the right) can fit into such a rectangle?
  - Any two points on the right are distance d or more apart.
  - Thus, there are at most 6 points in the rectangle.



Closest Pair Algorithm (Expanded)

Close-Pair(PointSet):

- Step 1: Split PointSet in half with a vertical line so that half are on left and half are on right;
- Step 2: Recursively determine the closest pair in each half and let d be smallest of the two distances.
- Step 3: Let L (on the left) and R (on the right) be the sets of points that are within distance d of the dividing line;
- Step 4: Sort L and R by y-coordinates;

Cont'd

- Step 5: For each point p of L, inspect the points of R with y-coordinate within distance d of p's y-coordinate to determine if there is a point within distance d of p;

  /* The L pointer always advances. */

  /* The R pointer may oscillate, but never by more than 6; */

- Step 6: Return the shortest distance found.

Analysis

- Step 1: Median + Partition: $O(n)$;
- Step 2: 2T(n/2);
- Step 3: $O(n)$;
- Step 4: $O(nlgn)$;
- Step 5: $O(n)$;
- Step 6: $O(1)$.
- Running time recurrence $T(n) = 2T(n/2) + O(nlgn)$ and $T(1) = 1$. This does not solve to $T(n) = O(nlgn)$.

Final Trick: Presorting

- Sort the set of points by y-coordinate before we start.
- Whenever we split a point set, we can run through the list sorted by y-coordinate and create a new list for each part, sorted by y-coordinates.
- Recurrence becomes $T(n) = 2T(n/2) + n$ and $T(1) = 1$.
- Solution: $T(n) = O(nlgn)$.
- It's possible to show that the closest pair can be found in $O(nlgn)$ time for any number of dimensions.