Introduction to Algorithms

Divide and Conquer

Divide and Conquer

- Divide and Conquer algorithms consist of two parts:
  - ***Divide:*** Smaller problems are solved recursively (except, of course, the base cases).
  - ***Conquer:*** The solution to the original problem is then formed from the solutions to the subproblems.

Divide and Conquer

- Traditionally
  - Algorithms which contain at least 2 recursive calls are called *divide and conquer* algorithms, while algorithms with one recursive call are not.
- Classic Examples
  - Mergesort and Quicksort
- Examples of recursive algorithms that are not Divide and Conquer
  - **Findset** in a Disjoint Set implementation is not divide and conquer.
    - Mostly because it doesn't "divide" the problem into smaller sub-problems since it only has one recursive call.
  - Even though the recursive method to compute the Fibonacci numbers has 2 recursive calls
    - It's really not divide and conquer because it doesn't divide the problem.

This Lecture

- *Divide-and-conquer* technique for algorithm design. Example problems:
  - Integer Multiplication
  - Subset Sum Recursive Problem
  - Closest Points Problem
  - Strassen's Algorithm

Integer Multiplication

$$\begin{array}{r} 1011 \\ \underline{\times 1111} \\ 1011 \\ 10110 \\ 101100 \\ \underline{+1011000} \\ 10100101 \end{array}$$

- The standard integer multiplication routine of 2 n-digit numbers
  - Involves n multiplications of an n-digit number by a single digit
  - Plus the addition of n numbers, which have at most 2 n digits

		quantity	*time*
1)	*Multiplication n-digit by 1-digit*	n	$O(n)$
2)	*Additions 2n-digit by n-digit max*	n	$O(n)$

$$\textit{Total time} = n*O(n) + n*O(n) = 2n*O(n) = O(n^2)$$

Integer Multiplication

```
    1011
   x1111
    1011
   10110
  101100
+1011000
10100101
```

- Let's consider a Divide and Conquer Solution
  - Imagine multiplying an n-bit number by another n-bit number.
    - We can split up each of these numbers into 2 halves.
    - Let the 1st number be I, and the 2nd number J
    - Let the "left half" of the 1st number by I_h and the "right half" be I_l.
  - So in this example: I is 1011 and J is 1111
    - I becomes $10*2^2 + 11$ where $I_h = 10*2^2$ and $I_l = 11$.
    - and $J_h = 11*2^2$ and $J_l = 11$

Integer Multiplication

- So for multiplying any n-bit integers I and J
  - We can split up I into $(I_h * 2^{n/2}) + I_l$
  - And J into $(J_h * 2^{n/2}) + J_l$
- Then we get
  - $I \times J = [(I_h \times 2^{n/2}) + I_l] \times [(J_h \times 2^{n/2}) + J_l]$
  - $I \times J = I_h \times J_h \times 2^n + (I_l \times J_h + I_h \times J_l) \times 2^{n/2} + I_l \times J_l$
- So what have we done?
  - We've broken down the problem of multiplying 2 n-bit numbers into
    - 4 multiplications of n/2-bit numbers plus 3 additions.
  - $T(n) = 4T(n/2) + \theta(n)$
  - Solving this using the master theorem gives us…
    - $T(n) = \theta(n^2)$

Integer Multiplication

- So we haven't really improved that much,
  - Since we went from a $O(n^2)$ solution to a $O(n^2)$ solution
- Can we optimize this in any way?
  - We can re-work this formula using some clever choices…
  - Some clever choices of:

$$P_1 = (I_h + I_l) \text{ x } (J_h + J_l) = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l$$

$$P_2 = I_h \text{ x } J_h \text{, and}$$

$$P_3 = I_l \text{ x } J_l$$

  - Now, note that

$$P_1 - P_2 - P_3 = I_h x J_h + I_h x J_l + I_l x J_h + I_l x J_l - I_h x J_h - I_l x J_l$$

$$= I_h x J_l + I_l x J_h$$

  - Then we can substitute these in our original equation:

$$I \text{ x } J = P_2 \text{ x } 2^n + [P_1 - P_2 - P_3] x \; 2^{n/2} + P_3.$$

Integer Multiplication

$$I \times J \quad = P_2 \times 2^n + [P_1 - P_2 - P_3] \times 2^{n/2} + P_3.$$

- Have we reduced the work?
  - Calculating P2 and P3 – take n/2-bit multiplications.
  - P1 takes two n/2-bit additions and then one n/2-bit multiplication.
  - Then, 2 subtractions and another 2 additions, which take O(n) time.
- This gives us : $T(n) = 3T(n/2) + \theta(n)$
  - Solving gives us $T(n) = \theta(n^{(\log_2 3)})$, which is approximately $T(n) = \theta(n^{1.585})$, a solid improvement.

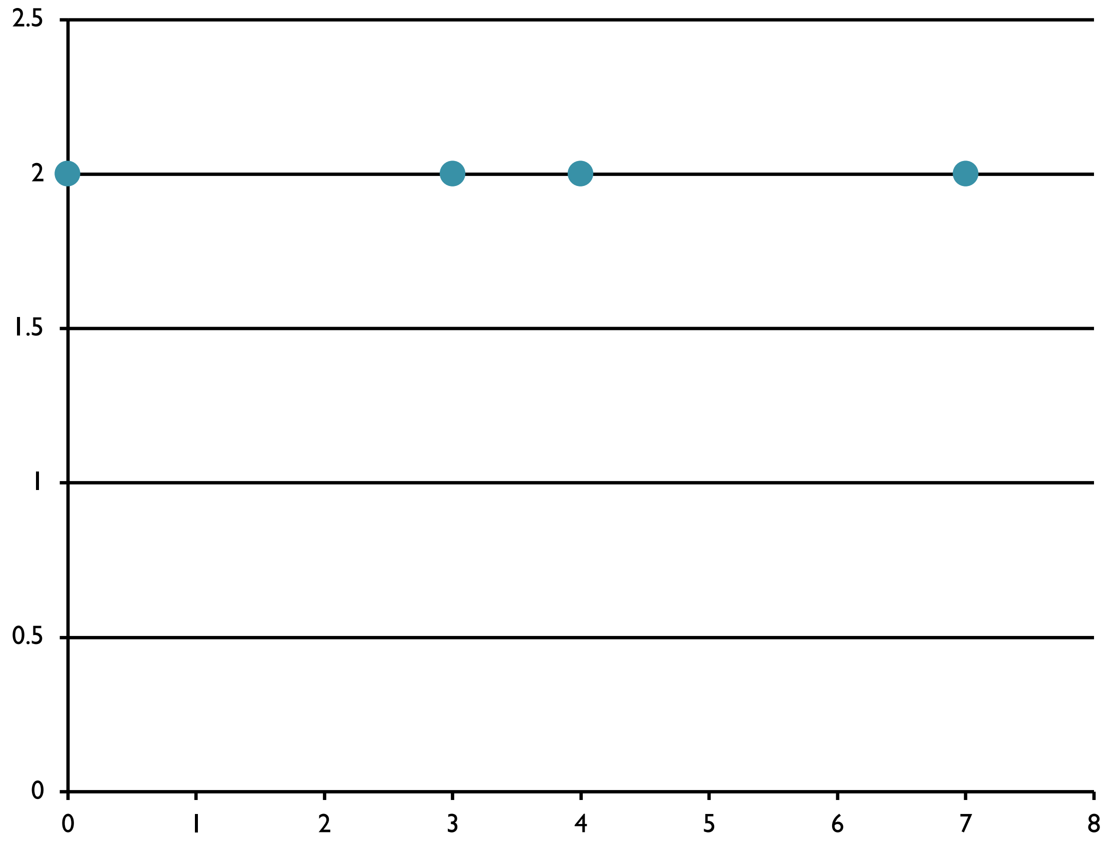
Integer Multiplication

- Although this seems it would be slower initially because of some extra pre-computing before doing the multiplications, ***for very large integers***, this will save time.

- Q: Why won't this save time for small multiplications?
  - A: The hidden constant in the $\theta(n)$ in the second recurrence is much larger. It consists of 6 additions/subtractions whereas the $\theta(n)$ in the first recurrence consists of 3 additions/subtractions.

Finding the Closest Pair of Points

- Problem:
  - Given n ordered pairs $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, find the distance between the two points in the set that are closest together.



Closest-Points Problem

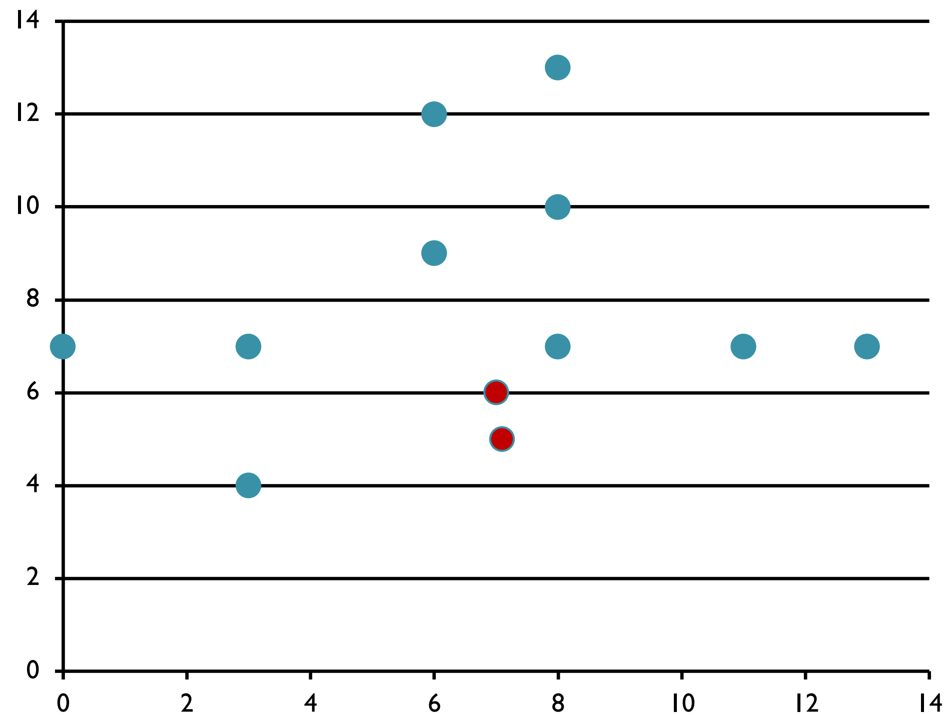
- Brute Force Algorithm
  - Iterate through all possible pairs of points, calculating the distance between each of these pairs. Any time you see a distance shorter than the shortest distance seen, update the shortest distance seen.

Since computing the distance between two points takes O(1) time,

And there are a total of $n(n-1)/2 = \theta(n^2)$ distinct pairs of points,

It follows that the running time of this algorithm is $\theta(n^2)$.

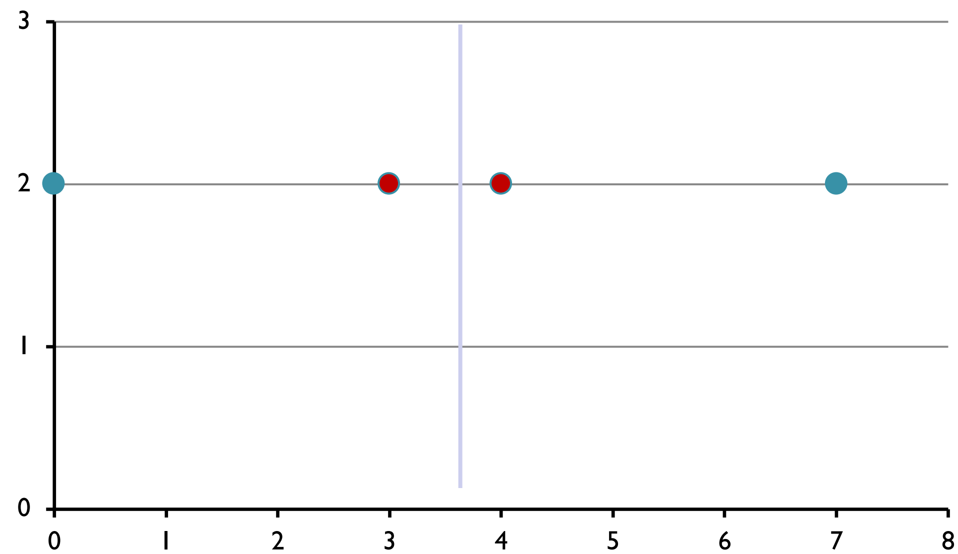
Can we do better?



Closest-Points Problem

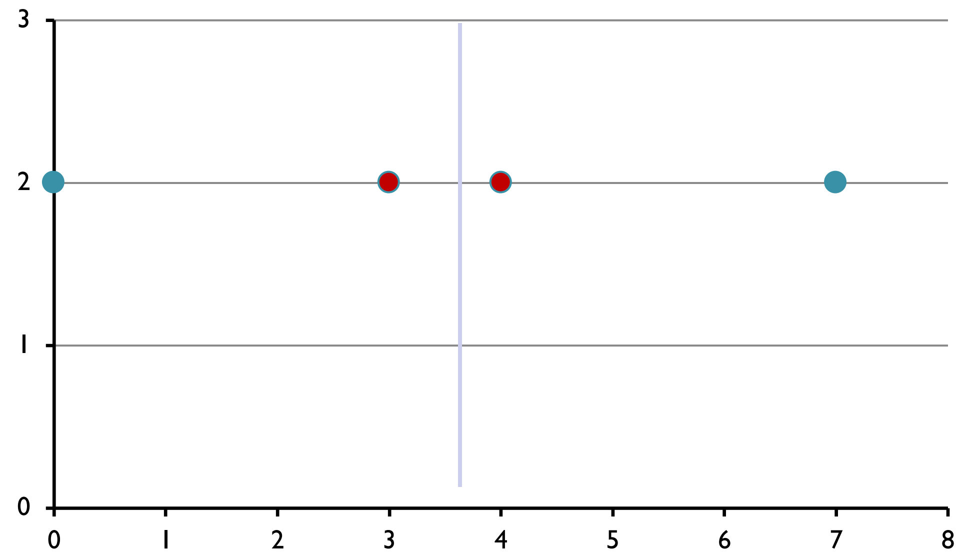
- Here's the idea:
  1) Split the set of n points into 2 halves by a vertical line.
     - Do this by sorting all the points by their x-coordinate and then picking the middle point and drawing a vertical line just to the right of it.
  2) Recursively solve the problem on both sets of points.
  3) Return the smaller of the two values.

- ***What's the problem with this idea?***



Closest Points Problem

- The problem is that the actual shortest distance between any 2 of the original points MIGHT BE between a point in the 1st set and a point in the 2nd set! Like in this situation:
  - *So we would get a shortest distance of 3, instead of 1.*

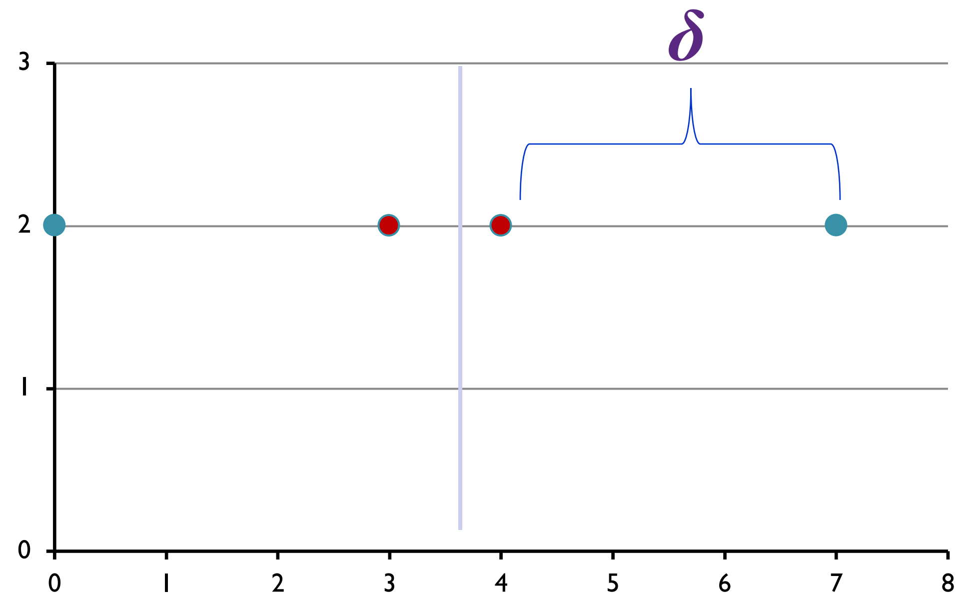


- *Original idea:*
  1) *Split the set of n points into 2 halves by a vertical line.*
     - *Do this by sorting all the points by their x-coordinate and then picking the middle point and drawing a vertical line just to the right of it.*
  2) *Recursively solve the problem on both sets of points.*
  3) *Return the smaller of the two values.*

We must adapt our approach:

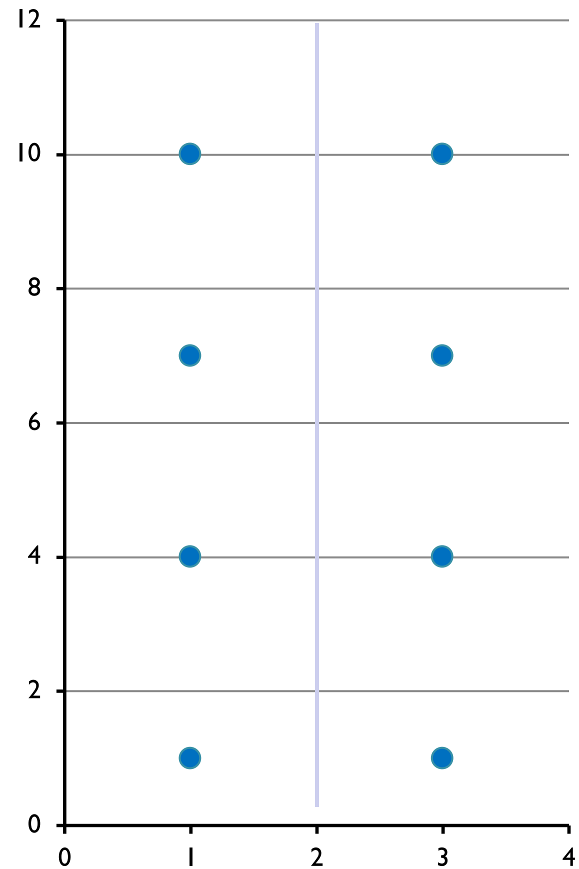
- *In step 3, we can "save" the smaller of the two values (called **δ**), then we have to check to see if there are points that are closer than **δ** apart.*

- *Do we need to search thru all possible pairs of points from the 2 different sides?*
  - *NO, we can only consider points that are within **δ** of our dividing line.*



Closest Points Problem

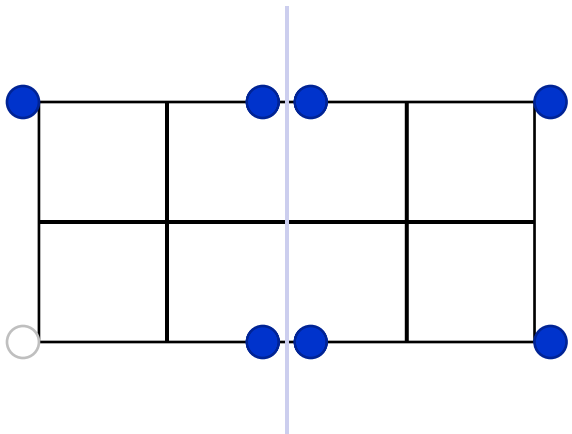
- However, one could construct a case where ALL the points on each side are within δ of the vertical line:



- *So, this case is as bad as our original idea where we'd have to compare each pair of points to one another from the different groups.*
- *But, wait!! Is it really necessary to compare each point on one side with every other point on every other side???*

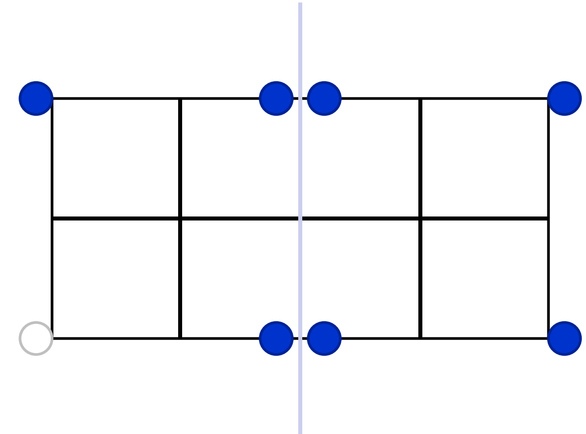
Closest Points Problem

- Consider the following rectangle around the dividing line that is constructed by eight δ/2 x δ/2 squares.



  - *Note that the diagonal of each square is $\delta/\sqrt{2}$, which is less than δ.*
  - *Since each square lies on a single side of the dividing line, at most one point lies in each box*
    - *Because if 2 points were within a single box the distance between those 2 points would be less than δ.*
  - *Therefore, there are at MOST 7 other points that could possibly be a distance of less than δ apart from a given point, that have a greater y coordinate than that point.*
    - *(We assume that our point is on the bottom row of this grid; we draw the grid that way.)*

Closest Points Problem



- Now we have the issue of how do we know ***which 7 points*** to compare a given point with?

- *The idea is:*
  - *As you are processing the points recursively, **SORT** them based on the **y-coordinate**.*
- *Then for a given point within the strip, you only need to compare with the next 7 points.*

Closest Points Problem

- Now the Recurrence relation for the runtime of this problem is:
  - $T(n) = 2T(n/2) + O(n)$
  - Which is the same as Mergesort, which we've shown to be O(n log n).

Subset Sum Recursive Problem

- Given n items and a target value, T, determine whether there is a subset of the items such that their sum equals T.
  - Determine whether there is a subset S of {1, …, n} such that the elements of S add up to T.

- Two cases:
  - Either there is a subset S in items {1, …, n-1} that adds up to T.
  - Or there is a subset S in items {1,…, n-1} that adds up to T – n, where S U {n} is the solution.

- The divide-and-conquer algorithm based on this recursive solution has a running time given by the recurrence:
  - $T(n) = 2T(n-1) + O(1)$

Strassen's Algorithm

- A fundamental numerical operation is the multiplication of 2 matrices.
  - The standard method of matrix multiplication of two n x n matrices takes T(n) = O(n3).

$$\begin{bmatrix} \mathbf{a_{11}} & \mathbf{a_{12}} & \mathbf{a_{13}} & \mathbf{a_{14}} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} X \begin{bmatrix} \mathbf{b_{11}} & b_{12} & b_{13} & b_{14} \\ \mathbf{b_{21}} & b_{22} & b_{23} & b_{24} \\ \mathbf{b_{31}} & b_{32} & b_{33} & b_{34} \\ \mathbf{b_{41}} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} \mathbf{c_{11}} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

The following algorithm multiples n x n matrices A and B:

```
// Initialize C.
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
          C [i, j] += A[i, k] * B[k, j];
```

Strassen's Algorithm

- We can use a Divide and Conquer solution to solve matrix multiplication by separating a matrix into 4 quadrants:

$$\left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \times \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right] = \left[\begin{array}{cc|cc} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ \hline c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{array}\right]$$

- *Then we know have:*

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

if $C = AB$***, then we have the following:***

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$
$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$
$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$
$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

*8 n/2 * n/2 matrix multiples + 4 n/2 * n/2 matrix additions*

$T(n) = 8T(n/2) + O(n^2)$

If we solve using the master theorem we still have $O(n^3)$

Strassen's Algorithm

- Strassen showed how two matrices can be multiplied using only 7 multiplications and 18 additions:
  - Consider calculating the following 7 products:
    - $q_1 = (a_{11} + a_{22}) * (b_{11} + b_{22})$
    - $q_2 = (a_{21} + a_{22}) * b_{11}$
    - $q_3 = a_{11} * (b_{12} – b_{22})$
    - $q_4 = a_{22} * (b_{21} – b_{11})$
    - $q_5 = (a_{11} + a_{12}) * b_{22}$
    - $q_6 = (a_{21} – a_{11}) * (b_{11} + b_{12})$
    - $q_7 = (a_{12} – a_{22}) * (b_{21} + b_{22})$
  - It turns out that
    - $c_{11} = q_1 + q_4 – q_5 + q_7$
    - $c_{12} = q_3 + q_5$
    - $c_{21} = q_2 + q_4$
    - $c_{22} = q_1 + q_3 – q_2 + q_6$

Strassen's Algorithm

- Let's verify one of these:

  Given: $A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$ $B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$ $C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$

  if $C = AB$ ***, we know:*** $c_{21} = a_{21}b_{11} + a_{22}b_{21}$

- *Strassen's Algorithm states:*
  - $\boldsymbol{c_{21} = q_2 + q_4}$,
    where $\boldsymbol{q_4 = a_{22} * (b_{21} - b_{11})}$ *and* $\boldsymbol{q_2 = (a_{21} + a_{22}) * b_{11}}$

Strassen's Algorithm

	Mult	Add	Recurrence Relation	Runtime
Regular	8	4	$T(n) = 8T(n/2) + O(n^2)$	$O(n^3)$
Strassen	7	18	$T(n) = 7T(n/2) + O(n^2)$	$O(n^{\log_2 7}) = O(n^{2.81})$

Strassen's Algorithm

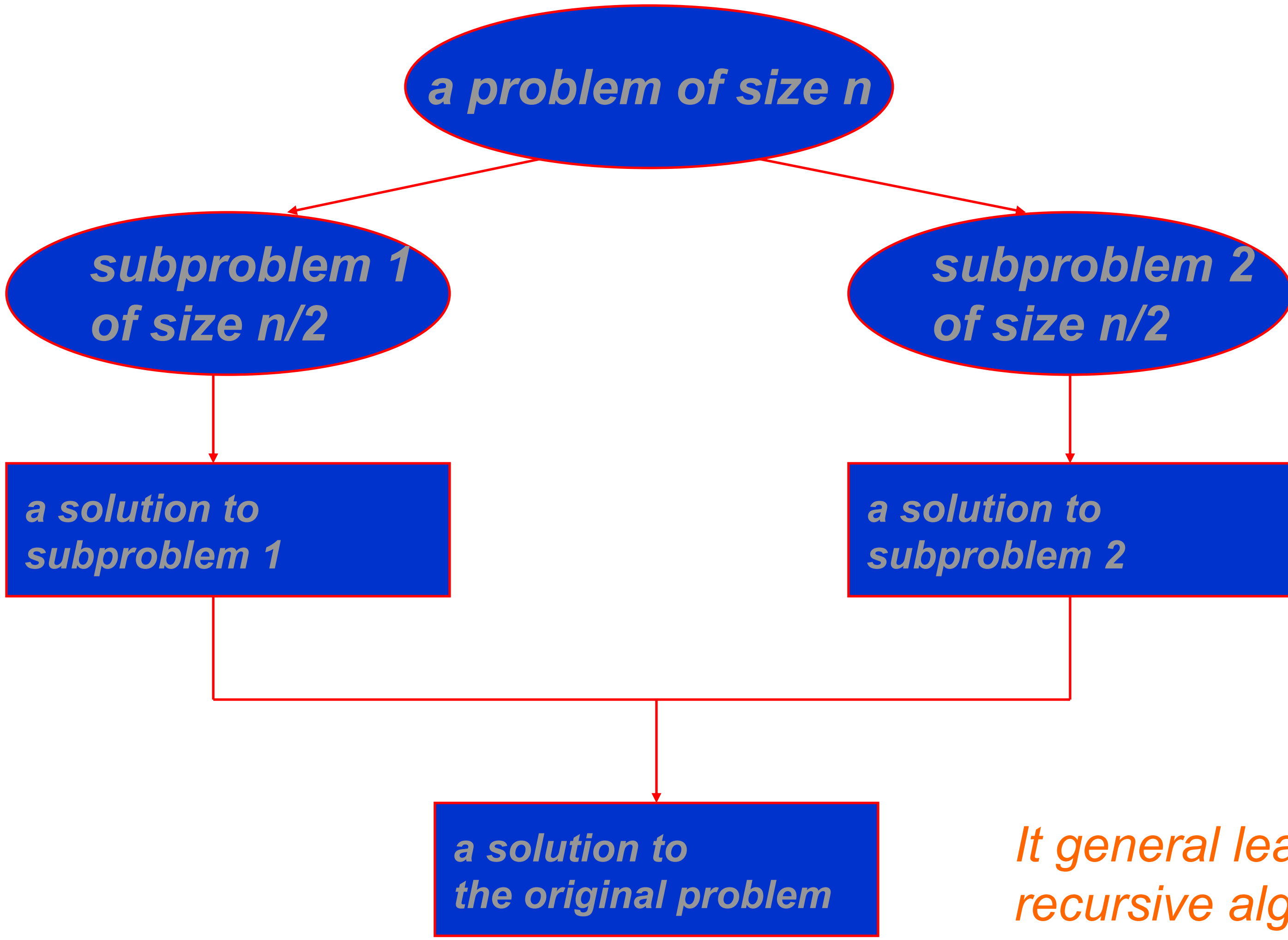
- I have no idea how Strassen came up with these combinations.
  - He probably realized that he wanted to determine each element in the product using less than 8 multiplications.
    - From there, he probably just played around with it.
- If we let T(n) be the running time of Strassen's algorithm, then it satisfies the following recurrence relation:
  - $T(n) = 7T(n/2) + O(n^2)$
    - It's important to note that the hidden constant in the $O(n^2)$ term is larger than the corresponding constant for the standard divide and conquer algorithm for this problem.
  - However, for large matrices this algorithm yields an improvement over the standard one with respect to time.

Divide-and-Conquer Summary

The most-well known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide-and-Conquer Technique



It general leads to a recursive algorithm!

Divide-and-Conquer Examples

- Sorting: mergesort and quicksort

- The Algorithms we've reviewed:
  - Integer Multiplication
  - Closest Pair of Points Problem
  - Subset Sum Recursive Problem
  - Strassen's Algorithm for Matrix Multiplication