Divide and conquer

1) Divide your problem into subproblems

2) Solve the subproblems recursively, that is,
run the same algorithm on the subproblems
(when the subproblems are very small, solve them from scratch)

3) Combine the solutions to the subproblems into a solution of the original problem

Divide and conquer

Recursion is “top-down” start from big problem, and make it smaller

Every divide and conquer algorithm can be written without recursion, in an iterative “bottom-up” fashion:
solve smallest subproblems, combine them, and continue

Sometimes recursion is a bit more elegant

```
Merge sort (low, high) {
   if (high-low <= 1) return;  //Smallest subproblems

   //Divide into subproblems low..split and split..high
   split = (low+high) / 2;

   MergeSort(low, split);    //Solve subproblem recursively
   MergeSort(split, high);  //Solve subproblem recursively

   //Combine solutions
     merge sorted sequences a[low..split] and a[split ..high]
     into the single sorted sequence a[low..high]
}
```

```
Merge sort (low, high) {
  if (high-low <= 1) return;
  split = (low+high) / 2;
  MergeSort(low, split);
  MergeSort(split, high);

  Merge
}
```

```
copy a[low ... split-1] to
                  scratch array;
m1 = 0;
m2 = split;
i = low;
while (i < m2 && m2 < high)
  if (scratch[m1] <= a[m2])
    a[i++]=scratch[m1++];
  else
    a[i++]=a[m2++];
while (i < m2)
  a[i++]=scratch[m1++];
```

Merge sort:

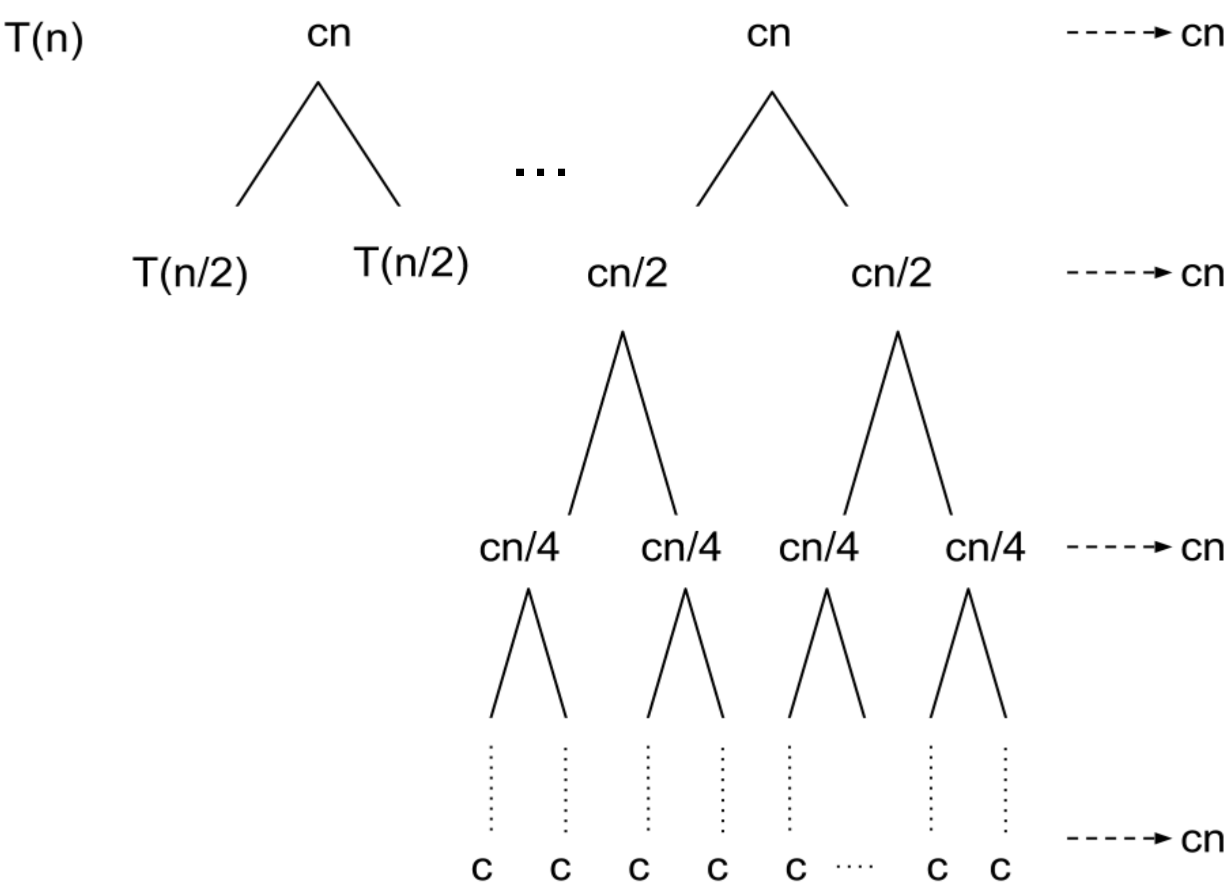
demo

Analysis of running time T(n)

$T(n) = 2\,T(n/2) + c\,n$ (time cn includes merge etc.)

At level i we have $2^i\, cn/2^i = cn$

Numbers of levels is log(n) ⇨ $T(n) = cn \log n$




```
MergeSort(low, high){
if (high-low <= 1) return;
   split = (low+high) / 2;
   MergeSort(low, split);
   MergeSort(split, high);
   Merge
   a[low..split] and a[split ..high]
   into a[low..high] }
```

Analysis of space

How many extra array elements we need?

O(n) because we need a scratch array of size O(n) to be able to merge the two sorted sequences into one sorted sequence.

```
MergeSort(low, high){
if (high-low <= 1) return;
   split = (low+high) / 2;
   MergeSort(low, split);
   MergeSort(split, high);
   Merge the two sorted sequences
a[low..split] and a[split ..high] into the
single sorted sequence a[low..high] }
```

Quick sort:

```
QuickSort(low, high)
  {
  if (high-low <= 1) return;
  partition(low, high) and return split;
  QuickSort(low, split);
  QuickSort(split+1, high);
}
```

Partition rearranges the input array a[low..high] into two (possibly empty) sub-arrays a[low.. split] and a[split+1.. high]

each element in a[low.. split] is ≤ a[split],

each element in a[split.. high] is ≥ a[split].

Quick sort:

```
QuickSort(low, high)
  {
  if (high-low <= 1) return;
  partition(low, high) and return split,
  QuickSort(low, split);
  QuickSort(split+1, high);
  }
```

The choice of split determines the running time of Quick sort. If the partitioning is balanced, Quick sort is as fast as Merge sort, if the partitioning is unbalanced, Quick sort is as slow as Bubble sort.

Quick sort(low, high)

```
if (high-low <= 1) return;
pivot = a[high-1];
split = low;
for (i=low; i<high-1; i++)
   if (a[i] <pivot) {
      swap a[i] and a[split];
      split++;
   }
swap a[high-1] and a[split];
QuickSort(low, split);
QuickSort(split+1, high);
Return;
```

Partition w.r.t. last element

Analysis of running time

T(n) = worst-case number of comparisons in Quick sort on an arrays of length n.

T(n) depends on the choice of

the pivot element split.

- Choosing pivot deterministically
- Choosing pivot randomly

```
QuickSort(low, high)
  {
  if (high-low <= 1) return;
  partition(low, high) and
return split,
  QuickSort(low, split);
  QuickSort(split+1, high);
  }
```

Analysis of running time

$T(n)$ = worst-case number of comparisons in Quick sort on an arrays of length n.

- Choosing pivot deterministically:

  the worst case happens when one sub-array is empty and the other is of size n-1, in this case :

  $T(n) = T(n-1) + T(0) + c\,n$

  $= ?$

Analysis of running time

$T(n)$ = worst-case number of comparisons in Quick sort on an arrays of length n.

- Choosing pivot deterministically:

  the worst case happens when one sub-array is empty and the other is of size n-1, in this case :

  $T(n)= T(n-1) + T(0) + c\,n$

  $\quad = O(n^2)$.

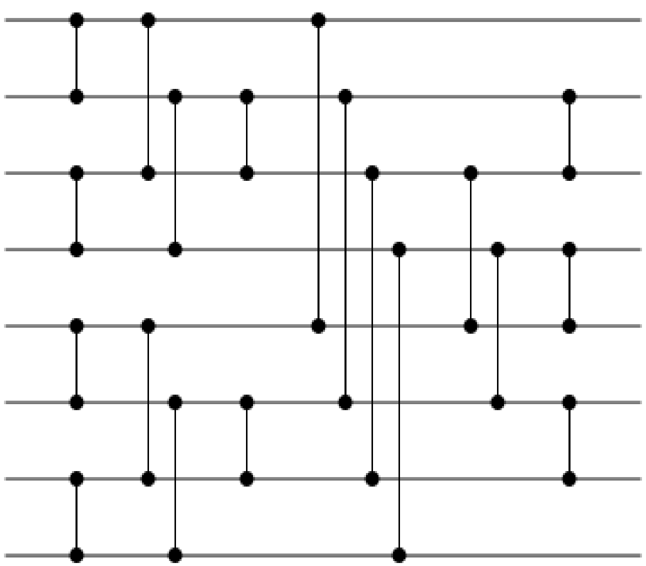
- Choosing pivot randomly we can guarantee

  $T(n) = O(n \log n)$ with high probability

Batcher's Odd-Even Mergesort

This is just like Merge sort except that the merge subroutine is replaced with a subroutine whose comparisons do not depend on the input.

Useful if you want to sort with a (non-programmable) piece of hardware



Batcher's Odd-Even Mergesort

This is just like Merge sort except that the merge subroutine is replaced with a subroutine whose comparisons do not depend on the input.

Assumption:

Size of the input sequence, n, is a power of 2.

```
Odd-even-Mergesort (a[1..n]) {
if n > 1 then
  odd-even-Mergesort(a[1.. n/2]);
  odd-even-Mergesort(a [n/2+1 .. n]);
  odd-even-merge(a[1..n]);
}
```

Same structure as Merge sort

But Odd-even-merge is more complicated, recursive

```
odd-even-merge(a[1..n]); {
  if n = 2 then compare-exchange(1,2);
  else {
    odd-even-merge(a[2,4 .. n]); //even subsequence

     odd-even-merge(a[1,3,5 .. n-1]); //odd subsequence

     for i ∈ {1,3,5, … n-1} do
         compare-exchange(i, i +1);
}
```

Compare-exchange(x,y) compares a[x] and a[y] and swaps them if necessary

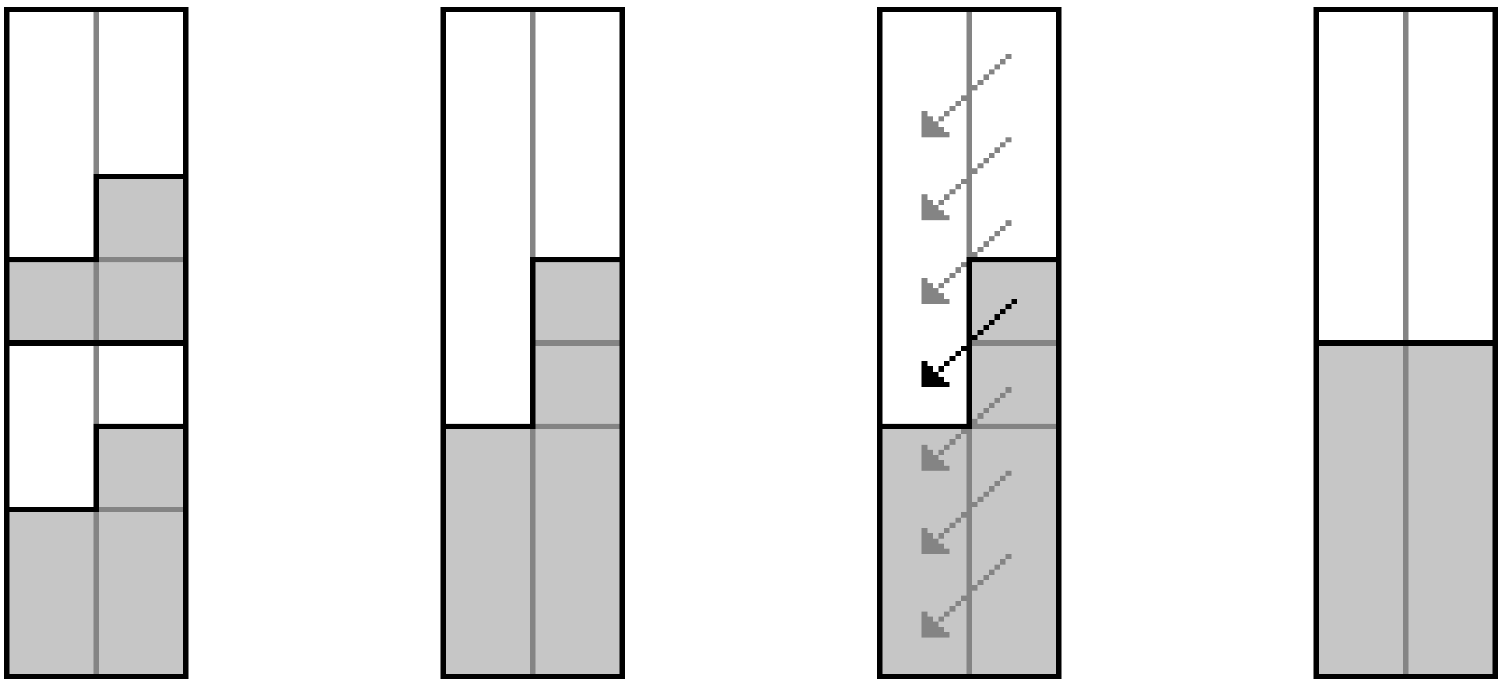
Merges correctly if a[1.. n/2] and a [n/2+1 .. n] are sorted

```
odd-even-merge(a[1..n]);
 if n = 2 then compare-exchange(1,2);
 else
   odd-even-merge(a[2,4 .. n]);
   odd-even-merge(a[1,3,5 .. n-1]);
   for i ∈ {1,3,5, … n-1} do
   compare-exchange(i, i +1);
```

0-1 principle: If we sort correctly all sequences of 0 and 1, then we sort correctly all sequences

```
odd-even-merge(a[1..n]);
  if n = 2 then compare-exchange(1,2);
  else
     odd-even-merge(a[2,4 .. n]);
     odd-even-merge(a[1,3,5 .. n-1]);
     for i ∈ {1,3,5, … n-1} do
     compare-exchange(i, i +1);
```

0	1
2	3
4	5
6	7
8	9
10	11
12	13
14	15



http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oem.htm

Analysis of running time

T(n) = number of comparisons.

$= 2T(n/2) + T'(n)$.

T'(n) = number of operations in odd-even-merge

$= 2T'(n/2) + c\, n = ?$

```
OE-Mergesort (a[1..n])
if n > 1 then
  OE-Mergesort(a[1.. n/2]);
  OE-Mergesort(a [n/2+1 .. n]);
  OE-merge(a[1..n]);
```

```
odd-even-merge(a[1..n]);
  if n = 2 then
    compare-exchange(1,2);
  else
    odd-even-merge(a[2,4 .. n]);
    odd-even-merge(a[1,3,5 .. n-1]);
    for i ∈ {1,3,5, … n-1} do
    compare-exchange(i, i +1);
```

Analysis of running time

T(n) = number of comparisons.

= 2T(n/2)+ T'(n)

= 2T(n/2)+ O(n log n).

= ?

T'(n) = number of operations in odd-even-merge

= 2T'(n/2)+c n = O(n logn).

```
OE-Mergesort (a[1..n])
if n > 1 then
  OE-Mergesort(a[1.. n/2]);
  OE-Mergesort(a [n/2+1 .. n]);
  OE-merge(a[1..n]);
```

```
odd-even-merge(a[1..n]);
  if n = 2 then
    compare-exchange(1,2);
  else
    odd-even-merge(a[2,4 .. n]);
    odd-even-merge(a[1,3,5 .. n-1]);
    for i ∈ {1,3,5, … n-1} do
    compare-exchange(i, i +1);
```

Analysis of running time

T(n) = number of comparisons.

$= 2T(n/2) + T'(n)$

$= 2T(n/2) + O(n \log n)$

$= O(n \log^2 n)$.

```
OE-Mergesort (a[1..n])
if n > 1 then
  OE-Mergesort(a[1.. n/2]);
  OE-Mergesort(a [n/2+1 .. n]);
  OE-merge(a[1..n]);
```

```
odd-even-merge(a[1..n]);
  if n = 2 then
    compare-exchange(1,2);
  else
    odd-even-merge(a[2,4 .. n]);
    odd-even-merge(a[1,3,5 .. n-1]);
    for i ∈ {1,3,5, … n-1} do
    compare-exchange(i, i +1);
```

Sorting algorithm	Time	Space	Assumption/ Advantage
Bubble sort	$\Theta(n^2)$	O(1)	Easy to code
Counting sort	$\Theta(n+k)$	O(n+k)	Input range is [0..k]
Radix sort	$\Theta(d(n+k))$	O(n+k)	Inputs are d-digit integers in base k
Quick sort (deterministic)	$O(n^2)$	O(1)	
Quick sort (Randomized)	O(n log n)	O(1)	
Merge sort	O (n log n)	O(n)	
Odd-even merge sort	$O(n \log^2 n)$	O(1)	Comparisons are independent of input

Next

- View other divide-and-conquer algorithms
- Some related to sorting

Selecting h-th smallest element

- Input: A[1], …, A[n], and h
  Desired output: B[h] for B = sorted version of A

- Can do with sorting, would take O(n log n)

- Now we give O(n) algorithm

Selecting h-th smallest element

- Divide array in consecutive blocks of 5
- Find median of each
- Find median of medians, x
- Partition array according to x. Let x be k-th element
- If k = h return x, if k > h recurse on left, if k < h recurse on right

- Divide array in consecutive blocks of 5
- Find median of each
- Find median of medians, x
- Partition array according to x. Let x be k-th element
- If k = h return x, if k > h recurse on left, if k < h recurse on right

- Analysis: When partitioning according to x, half the medians will be ≥ x. Each contributes ≥ 3 elements from their 5. So we throw away ≥ ?

- Divide array in consecutive blocks of 5
- Find median of each
- Find median of medians, x
- Partition array according to x. Let x be k-th element
- If k = h return x, if k > h recurse on left, if k < h recurse on right

- Analysis: When partitioning according to x, half the medians will be ≥ x. Each contributes ≥ 3 elements from their 5. So we throw away ≥ 3n/10 elements

- $T(n) \leq$?

- Divide array in consecutive blocks of 5
- Find median of each
- Find median of medians, x
- Partition array according to x. Let x be k-th element
- If k = h return x, if k > h recurse on left, if k < h recurse on right

- Analysis: When partitioning according to x, half the medians will be ≥ x. Each contributes ≥ 3 elements from their 5. So we throw away ≥ 3n/10 elements

- $T(n) \leq T(n/5) + T(7n/10) + O(n)$
- $T(n)$ = ? (not immediate)

- Divide array in consecutive blocks of 5
- Find median of each
- Find median of medians, x
- Partition array according to x. Let x be k-th element
- If $k = h$ return x, if $k > h$ recurse on left, if $k < h$ recurse on right

- Analysis: When partitioning according to x, half the medians will be $\geq x$. Each contributes ≥ 3 elements from their 5. So we throw away $\geq 3n/10$ elements

- $T(n) \leq T(n/5) + T(7n/10) + O(n)$
- $T(n) = O(n)$ because $1/5 + 7/10 = 9/10 < 1$

Closest pair of points

Input:

Set P of n points in the plane

Output:

Two points x_1 and x_2 with the shortest (Euclidean) distance from each other.

Closest pair of points

Input:

Set P of n points in the plane

Output:

Two points x_1 and x_2 with the shortest (Euclidean) distance from each other.

- For the following algorithm we assume that we have two arrays X and Y, each containing all the points of P.
- X is sorted so that the x-coordinates are increasing
- Y is sorted so that y-coordinates are increasing.

Closest pair of points

Divide: find a vertical line L that bisects P into two sets

P_L:= { points in P that are on L or to the left of L}.

P_R:= { points in P that are to the right of L}.

Such that $|P_L|$= n/2 and P_R= n/2 (plus or minus 1)

Easy to do given that we have X that's sorted.

Next: Conquer

Closest pair of points

Divide: find a vertical line L that bisects P into two sets

P_L:= { points in P that are on L or to the left of L}.

P_R:= { points in P that are to the right of L}.

Such that $|P_L|$= n/2 and P_R= n/2 (plus or minus 1)

Conquer: Make two recursive calls to find the closest pair of point in P_L and P_R.

Let the closest distances in P_L and P_R be δ_L and δ_R ,and let $\delta = \min(\delta_L , \delta_R)$.

Note computing X and Y for P_L and P_R is easy

Next: Combine

Closest pair of points

Divide: find a vertical line L that bisects P into two sets

P_L:= { points in P that are on L or to the left of L}.

P_R:= { points in P that are to the right of L}.

Such that $|P_L|= n/2$ and $P_R= n/2$ (plus or minus 1)

Conquer: Make two recursive calls to find the closest pair of point in P_L and P_R.

Let the closest distances in P_L and P_R be δ_L and δ_R ,and let $\delta = \min(\delta_L , \delta_R)$.

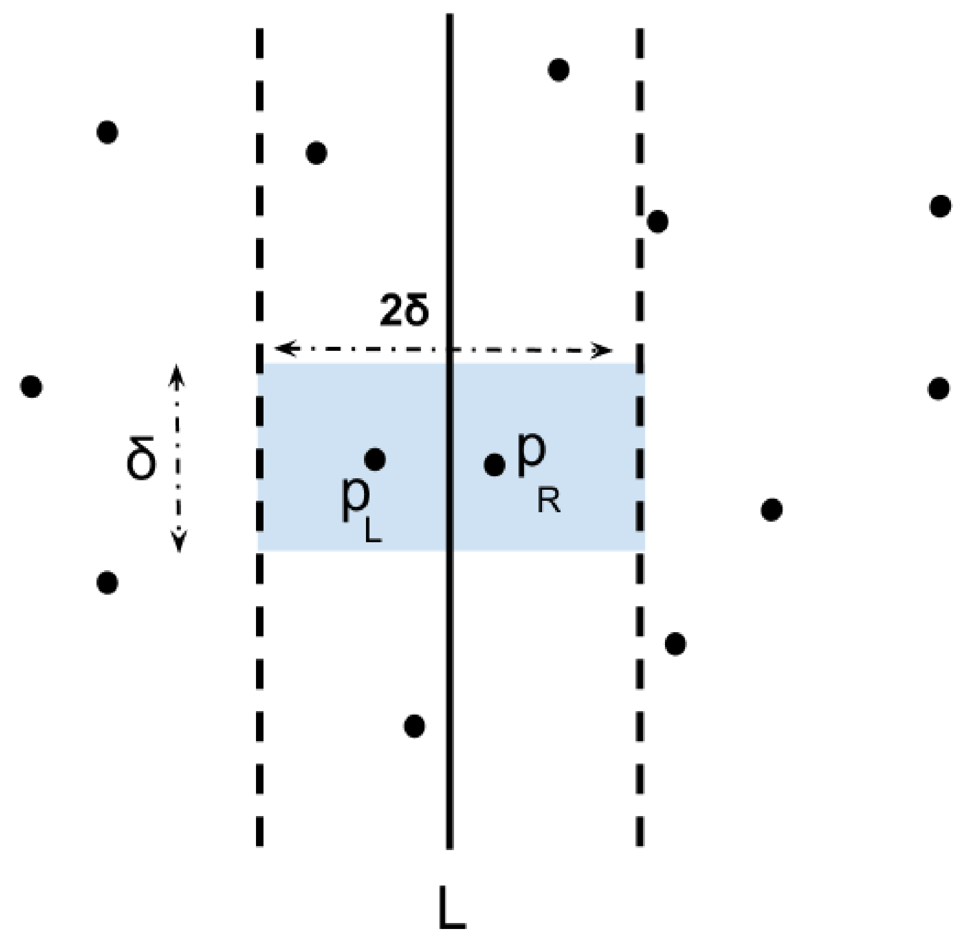
Combine: The closest pair is either the one with distance δ or it is a pair with one point in P_L and the other in P_R with distance less than δ. (No saving?)

Closest pair of points

Combine: The closest pair is either the one with distance δ or it is a pair with one point in P_L and the other in P_R with distance less than δ.

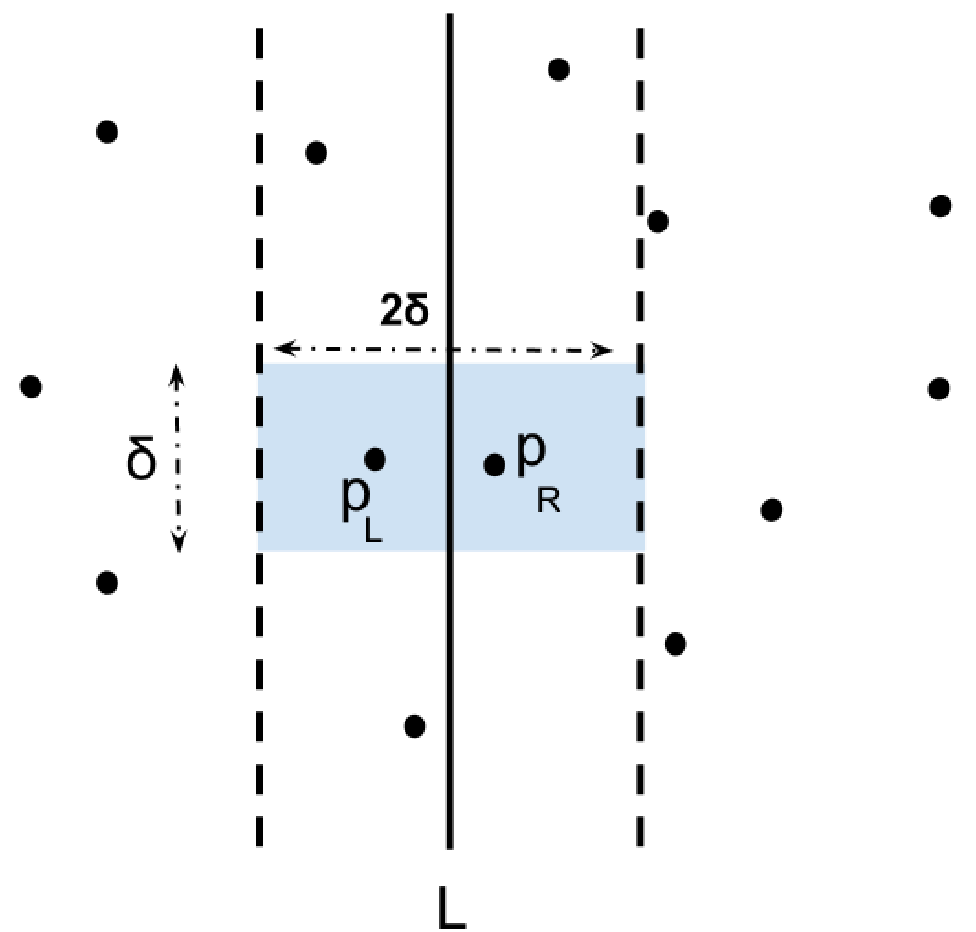
If such a pair exists it must be in a δ x 2δ box straddling L.

How do we find it?



We can find such pairs if any exist by:

- Create Y' by removing from Y points that are not in 2δ-wide vertical strip.
- For each point $p \in Y'$, Check the distance between p and the seven following points (why 7?) If any of them are closer than δ, update the closest pair and the shortest distance δ.
- Return δ and the closest pair.



- (Here p would be somewhere on top edge of box.)

Why 7?

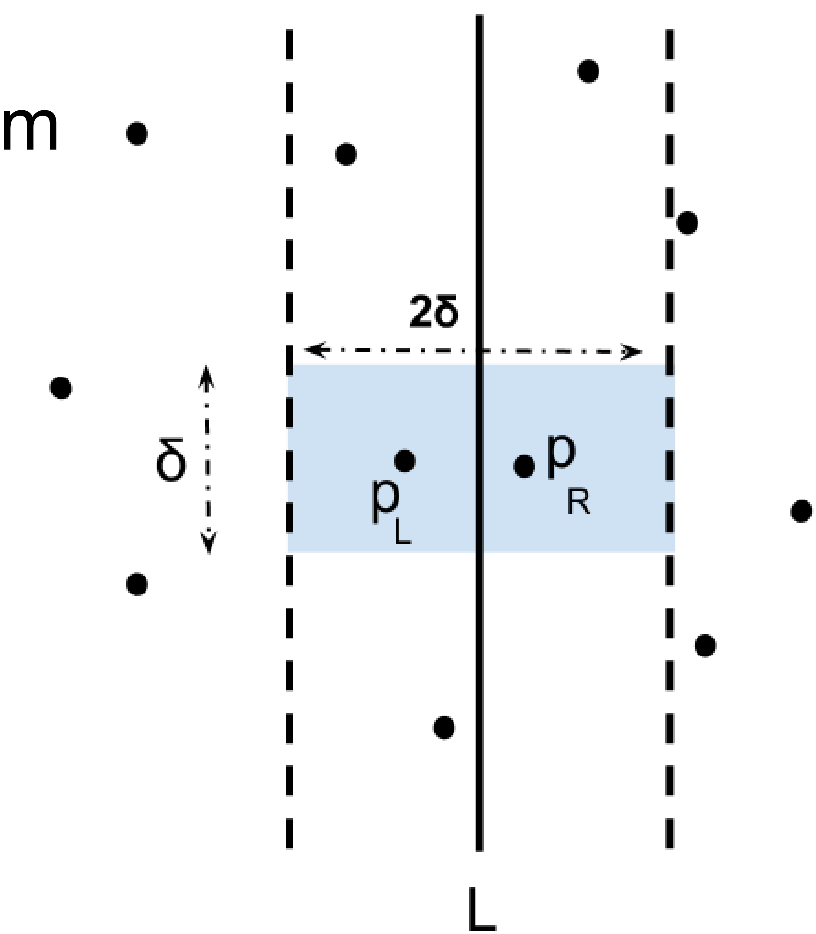
We know all pairs of points in P_L have distance ≥ δ

so at most 4 points in P_L can be in a δxδ square left of L.

Similarly to the right.

This gives 8 points, and one of them

is your current p



Analysis of running time

Similar to Merge sort:

$T(n)$ = number of operations

$$T(n) = 2\,T(n/2) + c\,n$$

$$= O(n \log n).$$

Exercise: What is the space requirement?

Addition

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer c=a + b.

Operations allowed: only on digits

The simple way to add takes ?

Addition

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer c=a + b.

Operations allowed: only on digits

The simple way to add takes O(n)

optimal?

Addition

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer c=a + b.

Operations allowed: only on digits

The simple way to add takes O(n)

This is optimal, since we need at least to write c

Multiplication

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer $c=a\cdot b$.

Operations allowed: only on digits

Simple way takes ?

```
   23958233
       5830 ×
------------
   00000000 ( =      23,958,233 ×     0)
  71874699  ( =      23,958,233 ×    30)
 191665864   ( =      23,958,233 ×   800)
119791165    ( =      23,958,233 × 5,000)
------------
139676498390 ( = 139,676,498,390        )
```

Multiplication

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer $c = a \cdot b$.

Operations allowed: only on digits

The simple way to multiply takes $\Omega(n^2)$

Can we do this any faster?

Multiplication

Example:

2-digit numbers N_1 and N_2 in base w.

$N_1 = a_0+a_1w.$

$N_2 = b_0+b_1w.$

For this example, think w very large, like $w = 2^{32}$

Multiplication

Example:

2-digit numbers N_1 and N_2 in base w.

$N_1 = a_0+a_1w.$

$N_2 = b_0+b_1w.$

$P = N_1N_2$

$= a_0b_0+(a_0b_1+a_1b_0)w+a_1b_1w^2$

$= p_0 + p_1w + p_2w^2.$

This can be done with ? multiplications

Multiplication

Example:

2-digit numbers N_1 and N_2 in base w.

$N_1 = a_0+a_1w.$

$N_2 = b_0+b_1w.$

$$P = N_1N_2$$

$$= a_0b_0+(a_0b_1+a_1b_0)w+a_1b_1w^2$$

$$= p_0 + p_1w + p_2w^2.$$

This can be done with 4 multiplications

Can we save multiplications, possibly increasing additions?

$$P = a_0b_0+(a_0b_1+a_1b_0)w+a_1b_1w^2$$
$$= p_0 + p_1w + p_2w^2.$$

Compute

$q_0=a_0b_0.$

$q_1=(a_0+a_1)(b_1+b_0).$

$q_2=a_1b_1.$

Note:

$q_0=p_0.$		$p_0=q_0.$
$q_1=p_1+p_0+p_2.$	$\Rightarrow$	$p_1=q_1-q_0-q_2.$
$q_2=p_2.$		$p_2=q_2.$

So the three digits of P are evaluated using 3 multiplications rather than 4.

What to do for larger numbers?

The Karatsuba algorithm

Input: two n-digit integers a, b in base w.

Output: One integer $c = a \cdot b$.

Divide:

How?

The Karatsuba algorithm

Input: two n-digit integers a, b in base w.

Output: One integer $c = a \cdot b$.

Divide:

$m = n/2.$

$a = a_0 + a_1 w^m.$

$b = b_0 + b_1 w^m.$

$$a \cdot b = a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m}$$
$$= p_0 + p_1 w^m + p_2 w^{2m}$$

The Karatsuba algorithm

Input: two n-digit integers a, b in base w.

Output: One integer $c = a \cdot b$.

Divide:

$m = n/2.$

$a = a_0 + a_1 w^m.$

$b = b_0 + b_1 w^m.$

$$a \cdot b = a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m}$$
$$= p_0 + p_1 w^m + p_2 w^{2m}$$

Conquer:

$q_0 = a_0 \times b_0.$

$q_1 = (a_0 + a_1) \times (b_1 + b_0).$

$q_2 = a_1 \times b_1.$

Each $\times$ is a recursive call

The Karatsuba algorithm

Input: two n-digit integers a, b in base w.

Output: One integer $c = a \cdot b$.

Divide:

$m = n/2.$

$a = a_0 + a_1 w^m.$

$b = b_0 + b_1 w^m.$

$$\begin{aligned} a \cdot b &= a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m} \\ &= p_0 + p_1 w^m + p_2 w^{2m} \end{aligned}$$

Conquer:

$q_0 = a_0 \times b_0.$

$q_1 = (a_0 + a_1) \times (b_1 + b_0).$

$q_2 = a_1 \times b_1.$

Each $\times$ is a recursive call

Combine:

$p_0 = q_0.$

$p_1 = q_1 - q_0 - q_2.$

$p_2 = q_2.$

Analysis of running time

$T(n)$ = number of operations.

$T(n) = 3\,T(n/2) + O(n)$

$= ?$

Can someone do this on the board?

Analysis of running time

$T(n)$ = number of operations.

$T(n) = 3\,T(n/2) + O(n)$

$= \Theta(n^{\log 3})$ (log in base 2)

$= O(n^{1.59})$.

Karatsuba may be used in your computers to reduce, say, multiplication of 128-bit integers to 64-bit integers.

Algorithms taking essentially $O(n \log n)$ are known.

We will see them later. Still based on divide and conquer!

Matrix Multiplication

n x n matrixes. Note input length is n^2

$$n=4 \quad \overset{A}{\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}} \bullet \overset{B}{\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}} = \begin{bmatrix} \ & \ & \ & \ \\ \ & \ & 1 & \ \\ \ & \ & \ & \ \\ \ & \ & \ & \ \end{bmatrix}$$

Just to write down output need time $\Omega(n^2)$

The simple way to do matrix multiplication takes ?

Matrix Multiplication

n x n matrixes. Note input length is n^2

$$n=4 \quad \left\{ \overset{A}{\begin{array}{|c|c|c|c|}\hline 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 \\ \hline\end{array}} \right. \bullet \overset{B}{\begin{array}{|c|c|c|c|}\hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 \\ \hline\end{array}} = \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & & 1 & \\ \hline & & & \\ \hline & & & \\ \hline\end{array}$$

Just to write down output need time $\Omega(n^2)$

The simple way to do matrix multiplication takes $O(n^3)$.

Strassen's Matrix Multiplication

Input: two n×n matrices A, B.

Output: One n×n matix C=A·B.

Strassen's Matrix Multiplication

Divide:

Divide each of the input matrices A and B into 4 matrices of size n/2xn/2, a follow:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$A.B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Strassen's Matrix Multiplication

Conquer:

Compute the following 7 products:

$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$.

$M_2 = (A_{21} + A_{22})B_{11}$.

$M_3 = A_{11}(B_{12} - B_{22})$.

$M_4 = A_{22}(B_{21} - B_{11})$.

$M_5 = (A_{11} + A_{12})B_{22}$.

$M_6 = (A_{21} - A_{11})(B_{11} - B_{12})$.

$M_7 = (A_{12} - A_{22})(B_{21} - B_{22})$.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Strassen's Matrix Multiplication

Combine:

$C_{11} = M_1 + M_4 - M_5 + M_7.$

$C_{12} = M_3 + M_5 .$

$C_{21} = M_2 + M_4 .$

$C_{22} = M_1 - M_2 + M_3 + M_6 .$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Analysis of running time

$T(n)$ = number of operations

$T(n) = 7\ T(n/2) + 18$ {Time to do matrix addition}

$= 7\ T(n/2) + \Theta(n^2)$

$=$?

Analysis of running time

T(n) = number of operations

$T(n) = 7\ T(n/2) + 18$ {Time to do matrix addition}

$$= 7\ T(n/2) + \Theta(n^2)$$

$$= \Theta(n^{\log 7})$$

$$= O(n^{2.81}).$$

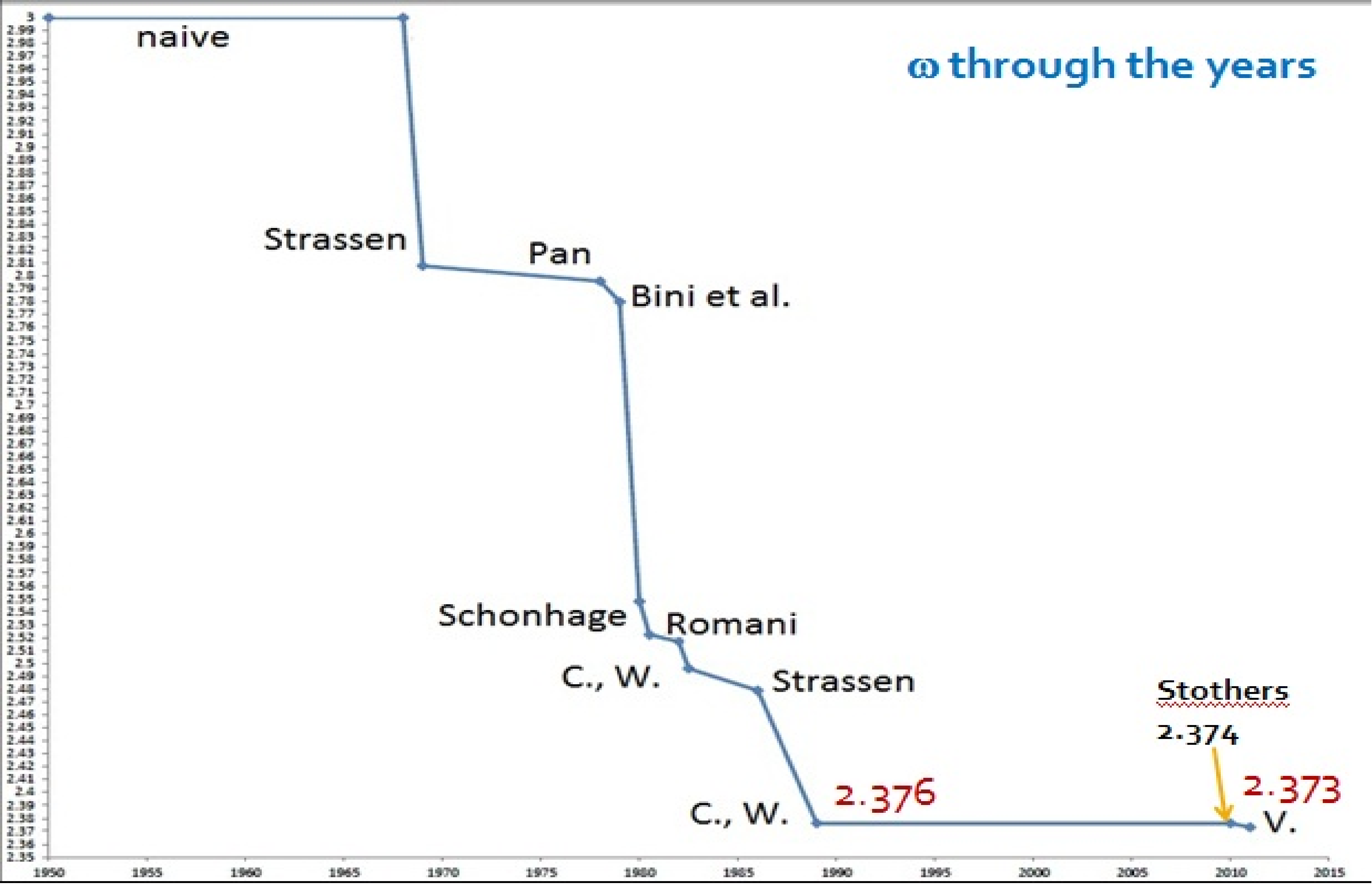
Definition: ω is the smallest number such that multiplication of n x n matrices can be computed in time $n^{\omega+\varepsilon}$ for every $\varepsilon > 0$

Meaning: time n^{ω} up to lower-order factors

$\omega \geq 2$ because you need to write the output

$\omega \leq 2.81$ Strassen, just seen

Determining ω is one of the most important problems



- Picture by Virginia V. = Vassilevska

From http://rjlipton.files.wordpress.com/2012/02/history.jpg