MORE FORMALLY

- Given two points \((x_1, y_1)\) and \((x_2, y_2)\), we say \((x_1, y_1)\) dominates \((x_2, y_2)\) if \(x_1 > x_2\) and \(y_1 > y_2\).
- Input: a set \(S\) of \(n\) points with distinct \(x\) values.
- Output: all non-dominated points in \(S\), i.e., all points in \(S\) that are not dominated by any point in \(S\).

PROBLEM: NON-DOMINATED POINTS

- A point dominates everything to the southwest.

\[ \begin{align*}
   x' &= \max(x, x) \\
   y' &= \max(y, y)
\end{align*} \]

So, I am a non-dominated point.

No other point dominates me.

What's an easy (brute force) algorithm for this?

BRUTE FORCE ALGORITHM

```
non-dominated(S)
for p in S
    dominated[p] = false
    for q in S
        if q \neq p and x[p] > x[q] and y[p] > y[q]
            dominated[p] = true
    if not dominated[p]
        print p
```

Running time? \(O(n^2)\)

Let's come up with a better algorithm.

PROBLEM DECOMPOSITION

Suppose we pre-sort the points in \(S\) with respect to their \(x\)-coordinates. This takes time \(\Theta(n \log n)\).

Observe that the non-dominated points form a staircase and all the other points are "under" this staircase. The \(y\)-coordinates of the staircase are determined by the \(y\)-coordinates of the non-dominated points. The \(x\)-coordinates of the staircase descend from left to right.
**PROBLEM DECOMPOSITION**

**Divide:** Let the first $n/2$ points be denoted $S_1$ and let the last $n/2$ points be denoted $S_2$.

**Combine:** Given the non-dominated points in $S_1$ and the non-dominated points in $S_2$, how do we find the non-dominated points in $S$?

Observe that no point in $S_1$ dominates a point in $S_2$. Therefore we only need to eliminate the points in $S_1$ that are dominated by a point in $S_2$. It turns out that this can be done in time $O(n)$.

**COMBINING TO GET NON-DOMINATED POINTS**

- Let $Q_1, Q_2, ..., Q_k$ be the non-dominated points in $S_1$.
- Let $R_1, R_2, ..., R_m$ be the non-dominated points in $S_2$.

Just need to find rightmost $Q_i$ that is not dominated (that has $y$-coordinate $\geq R_i.y$).

Running time complexity?

Assume $n = 2^j$ for simplicity.

$$T(n) = T(\frac{n}{2}) + O(n)$$

Same as merge sort recurrence ($O(n \log n)$).

Total for sort + recursion is $O(n \log n)$.

Either way doesn't matter.

```
// Points.java
import java.util.*;

class Points {
    public static void main(String[] args) {
        // Example code...
    }
}
```
WHAT IF X VALUES ARE NOT DISTINCT?

- It might contain multiple points with the same x value but with different y values
- If there are points in Q with the same x as R[1], and a lower y, then the algorithm would say they are dominated by R[1]. Wrong!
- We can find all the points with the same x as R[1] in linear time
- If there are multiple such points, and some are in Q, then they are not dominated by R[1], but might be dominated by the next element R[j] of R that has a different x
- So, we compare them with R[j] y (in linear time) instead of R[1] y
- All of the other points in Q with x different from R[1] x are compared with R[1] y as usual (in linear time)

MULTIPRECISION MULTIPLICATION

- Input: two k-bit positive integers X and Y
- With binary representations:
  \[ X = [X[k-1], \ldots, X[0]] \]
  \[ Y = [Y[k-1], \ldots, Y[0]] \]
- Output: The 2k-bit positive integer Z = XY
- With binary representation: \[ Z = [Z[2k-1], \ldots, Z[0]] \]

Here, we are interested in the bit complexity of algorithms that solve Multiprecision Multiplication, which means that the complexity is expressed as a function of k (the size of the problem instance is 2k bits)

BRUTE FORCE ALGORITHM

- One row per digit of Y
- For each row, copy the k bits of X
- Add the k rows together
- \( O(k^2) \) binary additions of \( O(k) \) bit numbers
- Total runtime is \( O(k^3) \) bit operations

A DIVIDE-AND-CONQUER APPROACH

Let \( X_L \) be the integer formed by the k/2 high-order bits of X and let \( X_R \) be the integer formed by the k/2 low-order bits of X.

Similarly for Y.

Thus

\[ X = 2^{k/2} X_L + X_R \quad \text{and} \quad Y = 2^{k/2} Y_L + Y_R \]

\[ Z = 2^k X_L Y_L + X_R Y_L + X_R Y_R \]

Recall: \( Z = 2^{k/2} X_L Y_L + X_R Y_L + X_R Y_R \)

EXPRESSING k-BIT MULT. AS k/2-BIT MULT.

- \[ X = 2^{k/2} X_L + X_R \]
  \[ Y = 2^{k/2} Y_L + Y_R \]
- So \[ X \times Y = (2^{k/2} X_L + X_R)(2^{k/2} Y_L + Y_R) \]
  \[ = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R \]
- Suggests a D&C approach...
  - Divide into four \( k/2 \)-bit multiplication subproblems
  - Conquer with recursive calls
  - Combine with \( k \)-bit addition and bit shifting

Recall: \( X \times Y = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R \)
For millennia it was widely thought that $O(n^2)$ multiplication was optimal.

Then in 1960, the 23-year-old Russian mathematician Anatoly Karatsuba took a seminar led by Andrey Kolmogorov, one of the great mathematicians of the 20th century.

Kolmogorov asserted that there was no general procedure for doing multiplication that required fewer than $n^2$ steps.

Karatsuba thought there was—and after a week of searching, he found it.

For the first time in a thousand years, mathematics had shown a speedup to multiplication.

KARATSUBA’S ALGORITHM

Let’s optimize from four subproblems to three.

Recall: $X \times Y = 2X_1Y_1 + 2^{1/2}(X_2Y_2 + X_1Y_2 + X_1Y_2 + X_1Y_2)$

- Idea: compute $X_1Y_1 + X_2Y_2$ with only one multiplication
- Note $X_1 + X_2)(Y_1 + Y_2) = X_1Y_1 + X_2Y_2 + X_1Y_2 + X_2Y_1$
- Set $X_1 = X_1 + X_0$ and $Y_1 = Y_1 + Y_0$
- Then $X_1Y_1 + X_2Y_2 = X_1Y_1 + X_2Y_2 - X_0Y_0$
- And the other two terms $X_1Y_1 + X_2Y_2$ are already in $XY$

So $XY = X_0Y_0 + 2^{1/2}(X_2Y_2 + X_1Y_2 + X_1Y_2 + X_1Y_2)$

Recall:

$\begin{align*}
\text{KaratsubaMultiply}(X, Y, k) = \\
\text{KaratsubaMultiply}(X, Y, k/2) + \text{KaratsubaMultiply}(X, Y, k/2) + \text{KaratsubaMultiply}(X, Y, k/2)
\end{align*}$

Running time complexity?

- Assume $k = 2^j$ for ease
- Master theorem:
  - $a = 3, b = 2, y = 1$
  - $x = \log_b a = \log_2 3$
  - $T(k) \in \Theta(k^{\log_b a}) = \Theta(k^{1.585})$

Input size increase by 10x causes runtime to $38x$.

Compare to $O(k^2)$ algoc - 10x input causes 100x time.

Quoting Fürer, author of the $O(n \log n 2^{O(\log^* n)})$ algorithm:

"It was kind of a general consensus that multiplication is such an important basic operation that, just from an aesthetic point of view, such an important operation requires a nice complexity bound..."

From general experience the mathematics of basic things at the end always turns out to be elegant."
And Harvey and van der Hoeven achieved $O(n \log n)$ in November 2020! [https://hal.archives-ouvertes.fr/hal-02070778/document]

Their method is a refinement of the major work that came before them. It splits up digits, uses an improved version of the fast Fourier transform, and takes advantage of other advances made over the past 40 years.

Lower bound of $\Omega(n \log n)$ is conjectured.

A conditional proof is known… it holds if a central conjecture in the area of network coding turns out to be true. [https://arxiv.org/abs/1902.10935]

Unfortunately, simple complexity doesn’t always mean simple algorithm…

**MATRIX MULTIPLICATION**

- Input: A and B
- Output: their product $C = AB$
- Word-RAM model (e.g., 64-bit int)
- Naïve algorithm for $n \times n$ matrices:
  - For each output cell $C_{ij}$
    $$C_{ij} = \text{DotProd}((a_{i1},\ldots,a_{in}), (b_{1j},\ldots,b_{nj}))$$
    $$= \sum_{k=1}^{n} a_{ik}b_{kj}$$
  - Running time?

**MULTIPLYING PARTITIONED MATRICES**

Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$
$$= \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Let $B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$
$$= \begin{bmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{bmatrix} \begin{bmatrix} f_{11} & f_{12} \\ f_{21} & f_{22} \end{bmatrix}$$
$$= \begin{bmatrix} e_{11}f_{11} + e_{12}f_{21} & e_{11}f_{12} + e_{12}f_{22} \\ e_{21}f_{11} + e_{22}f_{21} & e_{21}f_{12} + e_{22}f_{22} \end{bmatrix}$$

Note $C = AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix}$, where $a, b, \ldots, h$ are matrices

**IDENTIFYING SUBPROBLEMS TO SOLVE**

$$C = AB = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

Recall $ae, bg, \text{etc.}$, each represent matrix multiplication!

Can compute $C$ using 8 matrix multiplications.

**SIZE OF SUBPROBLEMS & SUBSOLUTIONS**

$$AB = \begin{bmatrix} a & b \\ c & d \end{bmatrix}\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix} = C = \begin{bmatrix} r & s \\ t & u \end{bmatrix}$$

- Suppose $A, B$ are $n \times n$ matrices
- For simplicity assume $n$ is a power of 2
- Then $a, b, c, d, e, f, g, h, r, s, t, u$ are $\frac{n}{2} \times \frac{n}{2}$ matrices
- So we compute $C$ with 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices
Running time complexity?

- \( T(n) = \Theta(n^2) + O(n^2) \)
- Master theorem
- \( a = 8, b = 2, y = 2 \)
- \( x = \log_2 3 \)
- \( x > y \) so \( T(n) \in \Theta(n^2) \)
- Same time as brute force!

**STRAßEN** FAST MATRIX MULTIPLICATION ALGORITHM

\[
AB = \begin{bmatrix}
a & b \\
c & d
\end{bmatrix}
\begin{bmatrix}
e & f \\
g & h
\end{bmatrix}
= \begin{bmatrix}
ae+bg & af+bh \\
cf+dh
\end{bmatrix}
= C = \begin{bmatrix}
i & j \\
k & l
\end{bmatrix}
\]

**Define**

- \( P_1 = a(f-h) \)
- \( P_2 = (a+b)h \)
- \( P_3 = (c+d)e \)
- \( P_4 = d(g-e) \)
- \( P_5 = (a+d)(c+h) \)
- \( P_6 = (b-d)(g+h) \)
- \( P_7 = (a-c)(e+f) \)

**Claim**

- As an example, according to Strassen, \( t = P_3 + P_4 \)
- Plugging in \( P_3, P_4 \), we get \( t = (c+d)e + d(g-e) \)
- This simplifies to \( t = ce + de + dg - de = ce + dg \)

\[
A = \begin{bmatrix}
a & b \\
c & d
\end{bmatrix}
B = \begin{bmatrix}
e & f \\
g & h
\end{bmatrix}
C = \begin{bmatrix}
i & j \\
k & l
\end{bmatrix}
\]

**STRAßEN** Matrix Multiplication

- \( T(n) = \Theta(n^2) + O(n^2) \)
- Master theorem
- \( a = 7, b = 2, y = 2 \)
- \( x = \log_2 7 \)
- \( x > y \) so \( T(n) \in \Theta(n^2) \)
- \( T(n) \in \Theta(n^{\log_2 7}) = \Theta(n^{2.8075}) \)

Running time complexity?
Strassen’s algorithm was improved in 1990 by Coppersmith-Winograd. Their algorithm has complexity $O(n^{2.376})$. Some tight improvements have been found more recently.

<table>
<thead>
<tr>
<th>How much better is $O(n^{2.81})$ than $O(n^{3})$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $n = 10,000$</td>
</tr>
<tr>
<td>$n^{2.81} = 174$ billion</td>
</tr>
<tr>
<td>$n^{3} = 1$ trillion ($41$ billion)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How much better is $O(n^{2.376})$ than $O(n^{3})$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $n = 10,000$</td>
</tr>
<tr>
<td>$n^{2.376} = 3.2$ billion</td>
</tr>
<tr>
<td>$n^{3} = 1$ trillion ($3.1$ billion)</td>
</tr>
</tbody>
</table>