DIVIDE AND CONQUER ALGORITHMS
PROBLEM: NON-DOMINATED POINTS

• Definition: 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 \(y_1 > y_2\)

• The points dominated by \((x_1,y_1)\) are all the points in the “southwest quadrant” of \((x_1,y_1)\)

• Input: a set \(S\) of \(n\) points,

• Output: all non-dominated points in \(S\), i.e., all points in \(S\) that are not dominated by any point in \(S\)
EASY ALGORITHM

• For each point p
  • p_dominated = false
  • For each other point q
    • If q dominates p then p_dominated = true
  • If p_dominated = false then add p to the output

Running time? O(n^2) Let’s come up with a better algorithm
Staircases

Observe that the non-dominated points form a staircase and all the other points are “under” this staircase.

The treads of the staircase are determined by the $y$-co-ordinates of the non-dominated points. The risers of the staircase are determined by the $x$-co-ordinates of the non-dominated points. The staircase descends from left to right.
Suppose we **pre-sort** the points in $S$ with respect to their $x$-co-ordinates. This takes time $\Theta(n \log n)$.

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

**Conquer:** Recursively solve the subproblems defined by the two instances $S_1$ and $S_2$.

**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 $O(n)$. 
DELETING DOMINATED POINTS IN S1

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

Just need to find rightmost $Q_i$ that is not dominated (that has $y$-coordinate $\geq R_1.y$)
Algorithm: \textit{Non-dominated}(S_1, \ldots, S_n)

comment: these \( n \) points are in increasing order WRT their \( x \)-co-ordinates

if \( n = 1 \) then return \( \{S[1]\} \)

\[
\begin{cases}
(Q[1], \ldots, Q[\ell]) \leftarrow \text{Non-dominated}(S[1], \ldots, S[[n/2]]) \\
(R[1], \ldots, R[m]) \leftarrow \text{Non-dominated}(S[[n/2] + 1], \ldots, S[n])
\end{cases}
\]

else

\[
i \leftarrow 1
\]

\[
\begin{cases}
\text{while } i \leq \ell \text{ and } Q[i].y > R[1].y \\
\quad \text{do } i \leftarrow i + 1
\end{cases}
\]

return \( (Q[1], \ldots, Q[i - 1], R[1], \ldots, R[m]) \)

comment: these points are in increasing order WRT their \( x \)-co-ordinates

\begin{center}
Running time complexity?
\end{center}
MULTIPRECISION MULTIPLICATION

• Input: two k-bit positive integers X and Y
  • With binary representations:
    X=[X[k-1], ..., X[0]]
    Y=[Y[k-1], ..., Y[0]]
  • Output: The 2k-bit positive integer Z=XY
    • With binary representation: Z=[Z[2k-1], ..., 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).
GRADE SCHOOL MULTIPLICATION

• Of two k-bit numbers
• Multiply all $k^2$ pairs of digits (top row times least significant bottom digit, top row times next bottom digit, ...)
• In doing so, create $k$ intermediate numbers, each containing between $k$ and $2k$ bits
• Sum these $k$ numbers

**Total** cost:
• $k^2$ single digit (bit) multiplications: $O(k^2)$ bit operations
• $k$ sums of $\sim 2k$-bit numbers: $O(k) \times O(k) = O(k^2)$ operations

Summing two k-bit numbers can be done in $O(k)$ time.
A Divide-and-Conquer Approach

Assume $k$ is even.

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

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

Therefore, we have

$$XY = 2^k X_L Y_L + 2^{k/2} (X_L Y_R + X_R Y_L) + X_R Y_R.$$ 

Multiplication by a power of 2 is just a left shift.
Not-So-Fast D&C Multiprecision Multiplication

Algorithm: $\text{NotSoFastMultiply}(X, Y, k)$

if $k = 1$
then $Z \leftarrow X[0] \times Y[0]$

else
\[
\begin{align*}
Z_1 & \leftarrow \text{NotSoFastMultiply}(X_L, Y_L, k/2) \\
Z_2 & \leftarrow \text{NotSoFastMultiply}(X_R, Y_R, k/2) \\
Z_3 & \leftarrow \text{NotSoFastMultiply}(X_L, Y_R, k/2) \\
Z_4 & \leftarrow \text{NotSoFastMultiply}(X_R, Y_L, k/2)
\end{align*}
\]

$Z \leftarrow \text{LeftShift}(Z_1, k) + Z_2 + \text{LeftShift}(Z_3 + Z_4, k/2)$

return $(Z)$

What is the complexity of this algorithm?
An Improvement

Recall

$$XY = 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R.$$ 

Karatsuba's algorithm reduces the number of subproblems from 4 to 3.

Define

$$Z_1 = X_L Y_L$$
$$Z_2 = X_R Y_R$$
$$Z_3 = (X_L + X_R)(Y_L + Y_R).$$

Then

$$X_L Y_R + X_R Y_L = Z_3 - Z_1 - Z_2.$$
Karatsuba Multiplication

Algorithm:  \textit{Karatsuba}(X, Y, k)

\begin{align*}
\text{if } k &= 1 \\
\text{then } Z &\leftarrow X[0] \times Y[0] \\
X_T &\leftarrow X_L + X_R \\
Y_T &\leftarrow Y_L + Y_R
\end{align*}

\begin{align*}
\text{else} & \begin{cases} \\
Z_1 &\leftarrow \text{Karatsuba}(X_L, Y_L, k/2) \\
Z_2 &\leftarrow \text{Karatsuba}(X_R, Y_R, k/2) \\
Z_3 &\leftarrow \text{Karatsuba}(X_T, Y_T, k/2), \\
Z &\leftarrow \text{LeftShift}(Z_1, k) + Z_2 + \text{LeftShift}(Z_3 - Z_1 - Z_2, k/2)
\end{cases}
\end{align*}

\text{return } (Z)

What is the complexity of this algorithm?
Recurrence relation $T(k) = 3T(k/2) + \Theta(k)$, where $k$ is a power of 2. The complexity of $T(k)$ is $\Theta(k^{\log_2 3}) = \Theta(k^{1.59})$ by the Master Theorem.

Note that $X_L + X_R$ and $Y_L + Y_R$ could be $(k/2 + 1)$-bit integers. However, computation of $Z_3$ can be accomplished by multiplying $(k/2)$-bit integers and accounting for carries by extra additions.

Various techniques can be used to handle the case when $k$ is not a power of two. One possible solution is to pad with zeroes on the left. So let $m$ be the smallest power of two that is $\geq k$. The complexity is $\Theta(m^{\log_2 3})$. Since $m < 2k$ the complexity is $O((2k)^{\log_2 3}) = O(3k^{\log_2 3}) = O(k^{\log_2 3})$.

There are further improvements known:

- The **Toom-Cook algorithm** splits $X$ and $Y$ into three equal parts and uses five multiplications of $(k/3)$-bit integers. The recurrence is $T(k) = 5T(k/3) + \Theta(k)$, and then $T(k) \in \Theta(k^{\log_3 5}) = \Theta(k^{1.47})$.

- The 1971 **Schonhage-Strassen algorithm** (based on FFT) has complexity $O(n \log n \log \log n)$.

- The 2007 **Furer algorithm** has complexity $O(n \log n 2^{O(\log^* n)})$. 
Matrix Multiplication

- Input: \( n \) by \( n \) matrices \( A \) and \( B \)
- Output: their product \( C = AB \)
- Unit cost model (64-bit integers)
- Naïve algorithm:
  - For each cell \( C_{ij} \) of \( C \)
  - \( C_{ij} = \text{DotProd} (\text{row}_i(A), \text{col}_j(B)^T) \)
    - \( = \sum_{k=1}^{n} A_{ik} B_{kj} \)
  - Running time?
ATTEMPTING ANOTHER SOLUTION

• What if we first **partition** the matrix into **sub-matrices**
• Then **divide and conquer** on the **sub-matrices**
• Example of partitioning: 4x4 matrix into four 2x2 matrices

\[
\begin{bmatrix}
M_1 & M_2 \\
M_3 & M_4
\end{bmatrix} =
\begin{bmatrix}
a_1 & b_1 & a_2 & b_2 \\
c_1 & d_1 & c_2 & d_2 \\
a_3 & b_3 & a_4 & b_4 \\
c_3 & d_3 & c_4 & d_4
\end{bmatrix}
\]
D&C Matrix Multiplication: Problem Decomposition

Let

\[
A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix}
\]

If \( A, B \) are \( n \) by \( n \) matrices, then \( a, b, ..., h, r, s, t, u \) are \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices, where

\[
\begin{align*}
  r &= ae + bg \\
  s &= af + bh \\
  t &= ce + dg \\
  u &= cf + dh
\end{align*}
\]

We require 8 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).

What is the complexity of the resulting divide-and-conquer algorithm?
DERIVING A RECURRENCE

• Each recursive call performs:
  • Four additions of n/2 by n/2 matrices
    • Steps: \( \frac{n}{2} \times \frac{n}{2} \in \Theta(n^2) \)
  • Eight **recursive calls** to perform multiplications of n/2 by n/2 matrices \( \rightarrow \) on problem size n/2

\[
T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)
\]

The work done by a call to `D&CMatrixMultiply`

\[
\begin{align*}
  r &= ae + bg \\
  t &= ce + dg \\
  s &= af + bh \\
  u &= cf + dh
\end{align*}
\]
SOLVE USING THE MASTER THEOREM

Suppose that \( a \geq 1 \) and \( b > 1 \). Consider the recurrence

\[
T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^y), \text{ where } n \text{ is a power of } b.
\]

Denote \( x = \log_b a \). Then

\[
T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } y < x \\
\Theta(n^x \log n) & \text{if } y = x \\
\Theta(n^y) & \text{if } y > x.
\end{cases}
\]

Recurrence: \( T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) \)

\[
a = 8 \quad b = 2 \quad y = 2 \quad x = \log_2 8 = 3
\]

\( x > y \) so \( T(n) \in \Theta(n^x) = \Theta(n^3) \)

No better than the naïve algorithm!
Define \( A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix} \)

\[ P_1 = a(f - h) \]
\[ P_3 = (c + d)e \]
\[ P_5 = (a + d)(e + h) \]
\[ P_7 = (a - c)(e + f). \]

\[ P_2 = (a + b)h \]
\[ P_4 = d(g - e) \]
\[ P_6 = (b - d)(g + h) \]

Then, compute

\[ r = P_5 + P_4 - P_2 + P_6 \]
\[ s = P_1 + P_2 \]
\[ t = P_3 + P_4 \]
\[ u = P_5 + P_1 - P_3 - P_7. \]

We now require only 7 multiplications of \( \frac{n}{2} \) by \( \frac{n}{2} \) matrices in order to compute \( C = AB \).