PROBLEM: **NON-DOMINATED POINTS**

- A point dominates everything to the southwest.

So, I am a non-dominated point.

No other point dominates me.

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

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

**BRUTE FORCE ALGORITHM**

```plaintext
NDPoints(S):
    for p in S
        dominated[p] = false
        for q in S
            if q = p and x[p] > x[q] and y[p] > y[q]
                dominated[p] = true
                if not dominated[p]
                    print p
```

Running time: \(\mathcal{O}(n^2)\)

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 \(\Theta(n \log n)\).
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 $S$?

Note 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 $O(n)$.

running time complexity?

$T(n) = 2T(n/2) + Θ(n)$

Assume $n = 2^j$ for simplicity

Same as merge sort recurrence (80 ms)

To total for split + recursion is $Θ(n log n)$ (80 ms)

Assume $n = 2^j$ for simplicity

To total for split + recursion is $Θ(n log n)$ (80 ms)

Assume $n = 2^j$ for simplicity

To total for split + recursion is $Θ(n log n)$ (80 ms)

Assume $n = 2^j$ for simplicity

To total for split + recursion is $Θ(n log n)$ (80 ms)
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).

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 $Y$.

$x = [X[k/2], ..., X[0]]$
$y = [Y[k/2], ..., Y[0]]$

Thus

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

$2^{k/2}X_L + X_R = x$ and $2^{k/2}Y_L + Y_R = y$

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

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

$= 2^kX_LY_L + 2^{k/2}(X_LY_R + X_RY_L) + X_RY_R$

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

BREUTE FORCE ALGORITHM

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

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

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

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

$= 2^kX_LY_L + 2^{k/2}(X_LY_R + X_RY_L) + X_RY_R$

Assume $k = 2^l$ for ease

$T(k) = 4T(k/2) + \Theta(k)$

Master Theorem says

$T(k) \in \Theta(2^{2l}) = \Theta(k^2)$

Same complexity as brute-force!
**KARATSUBA’S ALGORITHM**

- Let’s optimize from four subproblems to three.
  
  **Recall:** $XY = 2^3X_3Y_3 + 2^2(X_2Y_2 + X_2Y_3 + X_3Y_2) + X_1Y_1$

- Idea: compute $X_1Y_1, X_2Y_2, X_3Y_3$ with only one multiplication.
- Note $X_1Y_1 + X_2Y_2$ appears in $(X_1 + X_2)(Y_1 + Y_2)$.
  
  $(X_1 + X_2)(Y_1 + Y_2) = X_1Y_1 + X_1Y_2 + X_2Y_1 + X_2Y_2 + X_3Y_3$

- Let $X_1 = X_2 + X_3$ and $Y_1 = Y_2 + Y_3$.
  
  Then $X_1Y_1 + X_2Y_2 = X_2(Y_1 + Y_2) - X_2Y_3 - X_3Y_2$.

- And the other two terms $X_1Y_2, X_3Y_1$ are already in $XY$.

So $XY = 2^3X_3Y_3 + 2^2(X_2Y_2 - X_2Y_3 - X_3Y_2) + X_1Y_1$.

**Matrix Multiplication**

- Input: $A$ and $B$.

- Output: their product $C = AB$.

- Word-RAM model (e.g., 64-bit int).

- Naive algorithm for $n \times n$ matrices:
  - For each output cell $C_{ij}$
    
    $C_{ij} = \sum_{k=1}^n A_{ik}B_{kj}$

- Running time?

**Multiplying Partitioned Matrices**

Let $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and $B = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$.

Note $C = A B = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}$ where $a, b, ..., h$ are matrices.
Running time complexity?

Θ(1)

Θ(n²)

8Tn²

Θ(n²)

(recall A, B have n² entries)

Intuition: to get speedup, must reduce the number of subproblems or their size

Define

Each Pi requires one multiplication
Can combine these Pi terms with +/− to compute r, s, t, u!

Claim

Can combine c with multiplication of matrices

So we combine c with multiplication of matrices

Then A(c')c' can use the or square of P

Supporting a^k x k matrices

AB = C

Size of subproblems & subproblems to solve

Identifying subproblems
Running time complexity?

- $\mathcal{O}(n)$
- $\mathcal{O}(\log_2 7)$
- $\Theta(1)$
- $\Theta(n^2)$
- $7T(n)$
- $\Theta(n^2)$

How much better is $\Theta(n^{2.81})$ than $\Theta(n^3)$?

Let $n=10,000$

- $n^{2.81} \approx 174$ billion
- $n^3 = 1$ trillion (~6x more)

How much better is $\Theta(n^{2.376})$ than $\Theta(n^3)$?

Let $n=10,000$

- $n^{2.376} \approx 3.2$ billion
- $n^3 = 1$ trillion (~312x)

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