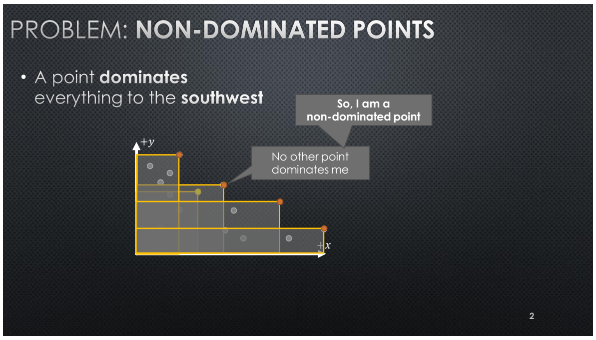
CS 341: ALGORITHMS

Lecture 3: divide & conquer II

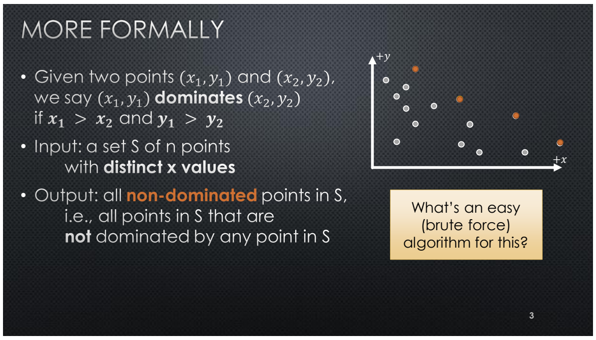
Readings: see website

Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca



PROBLEM: **NON-DOMINATED POINTS**

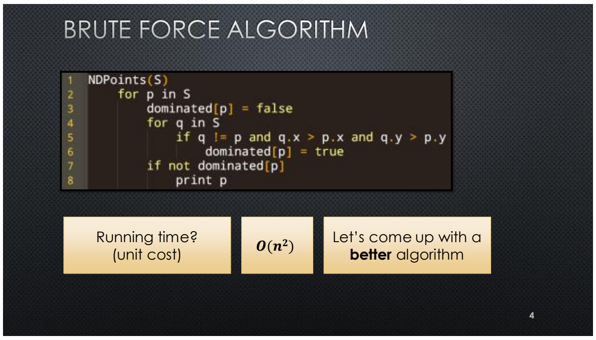
- A point **dominates** everything to the **southwest**



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 $\boldsymbol{x_1 > x_2}$ and $\boldsymbol{y_1 > y_2}$
- 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

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



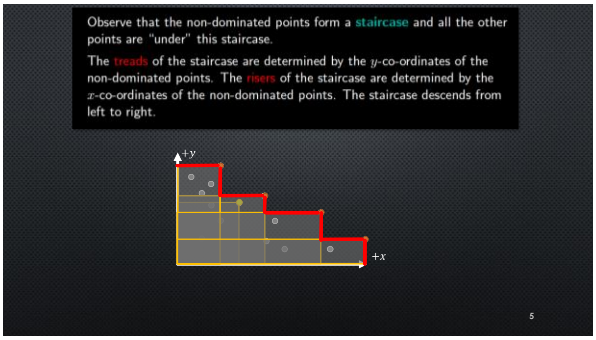
BRUTE FORCE ALGORITHM

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

Running time? (unit cost)

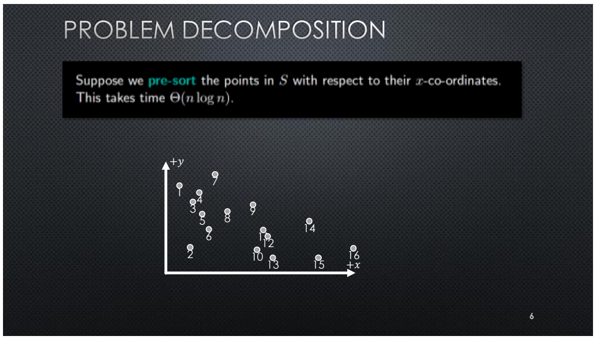
$\boldsymbol{O(n^2)}$

Let's come up with a **better** algorithm



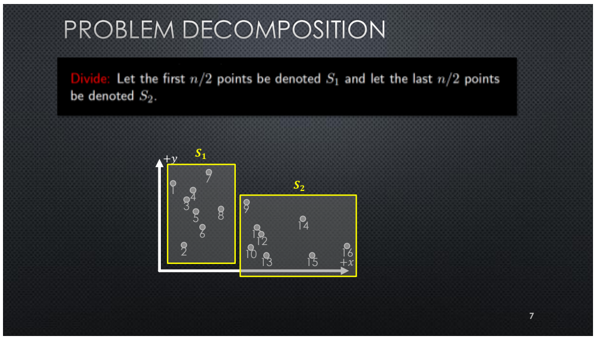
Observe that the non-dominated points form a staircase and all the other points are "under" this staircase.

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.



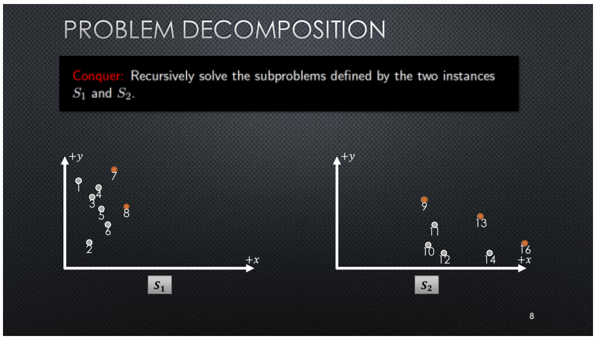
PROBLEM DECOMPOSITION

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



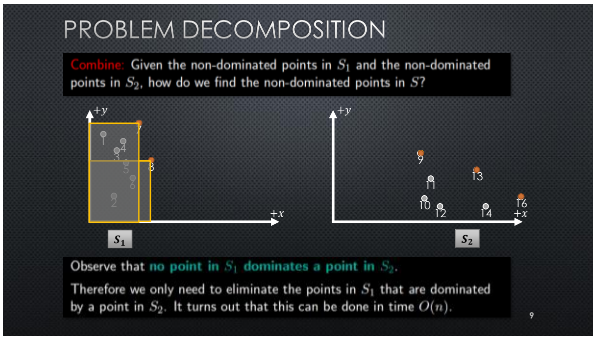
PROBLEM DECOMPOSITION

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



PROBLEM DECOMPOSITION

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

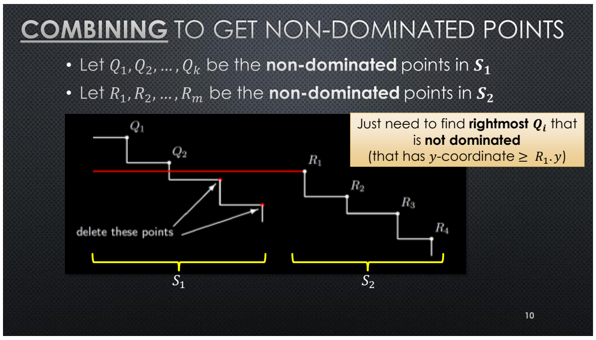


PROBLEM DECOMPOSITION

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.

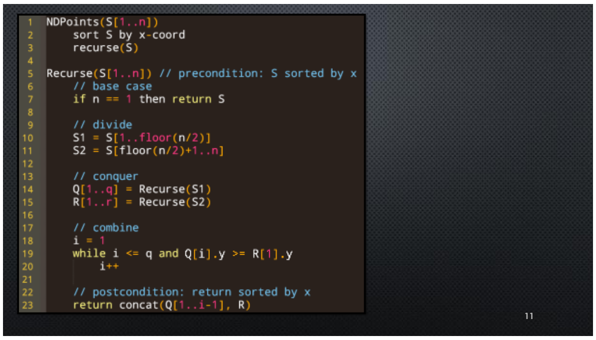
Therefore we only need to eliminate the points in S_1 that are dominated by a point in S_2. It turns out that this can be done in time $O(n)$.



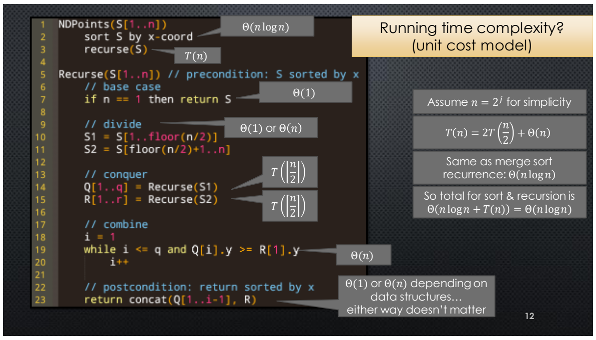
COMBINING TO GET NON-DOMINATED POINTS

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

Just need to find **rightmost $\boldsymbol{Q_i}$** that is **not dominated** (that has y-coordinate $\geq R_1.y$)



```
NDPoints(S[1..n])
    sort S by x-coord
    recurse(S)

Recurse(S[1..n]) // precondition: S sorted by x
    // base case
    if n == 1 then return S

    // divide
    S1 = S[1..floor(n/2)]
    S2 = S[floor(n/2)+1..n]

    // conquer
    Q[1..q] = Recurse(S1)
    R[1..r] = Recurse(S2)

    // combine
    i = 1
    while i <= q and Q[i].y >= R[1].y
        i++

    // postcondition: return sorted by x
    return concat(Q[1..i-1], R)
```



```
NDPoints(S[1..n])
    sort S by x-coord
    recurse(S)

Recurse(S[1..n]) // precondition: S sorted by x
    // base case
    if n == 1 then return S

    // divide
    S1 = S[1..floor(n/2)]
    S2 = S[floor(n/2)+1..n]

    // conquer
    Q[1..q] = Recurse(S1)
    R[1..r] = Recurse(S2)

    // combine
    i = 1
    while i <= q and Q[i].y >= R[1].y
        i++

    // postcondition: return sorted by x
    return concat(Q[1..i-1], R)
```

$\Theta(n \log n)$

$T(n)$

$\Theta(1)$

$\Theta(1)$ or $\Theta(n)$

$T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)$

$T\left(\left\lceil\frac{n}{2}\right\rceil\right)$

$\Theta(n)$

$\Theta(1)$ or $\Theta(n)$ depending on data structures… either way doesn't matter

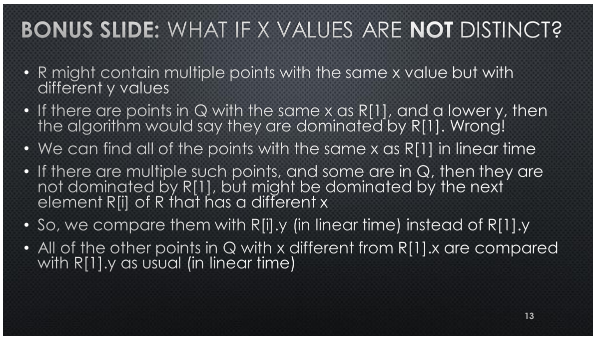
Running time complexity? (unit cost model)

Assume $n = 2^j$ for simplicity

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

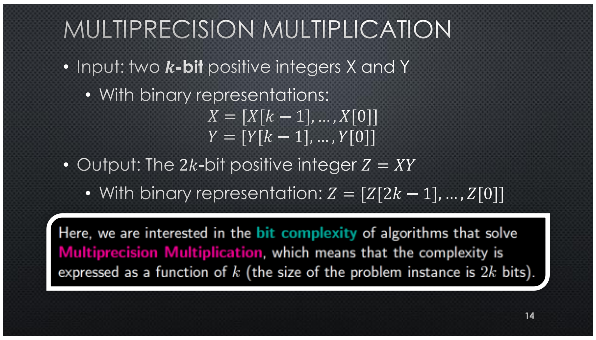
Same as merge sort recurrence: $\Theta(n \log n)$

So total for sort & recursion is $\Theta(n \log n + T(n)) = \Theta(n \log n)$



BONUS SLIDE: WHAT IF X VALUES ARE **NOT** DISTINCT?

- R 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 of 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[i] of R that has a different x
- So, we compare them with R[i].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)



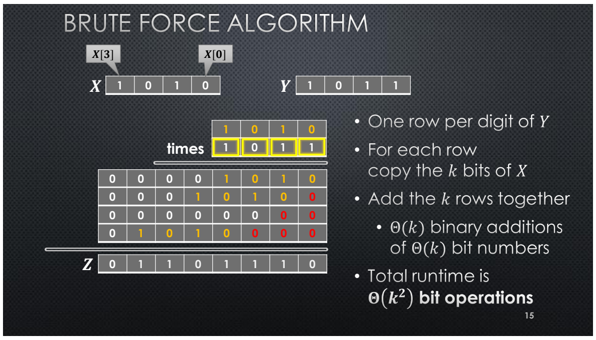
MULTIPRECISION MULTIPLICATION

- Input: two $\boldsymbol{k}$**-bit** positive integers X and Y
  - With binary representations:

$$X = [X[k-1], \dots, X[0]]$$
$$Y = [Y[k-1], \dots, Y[0]]$$

- Output: The $2k$-bit positive integer $Z = XY$
  - With binary representation: $Z = [Z[2k-1], \dots, Z[0]]$

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).



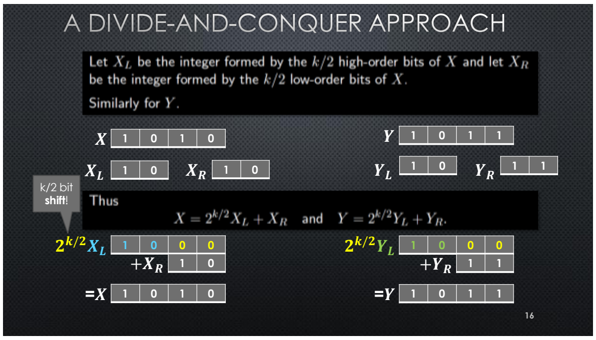
BRUTE FORCE ALGORITHM

X 1 0 1 0

Y 1 0 1 1

					1	0	1	0
times					1	0	1	1
	0	0	0	0	1	0	1	0
	0	0	0	1	0	1	0	0
	0	0	0	0	0	0	0	0
	0	1	0	1	0	0	0	0
Z	0	1	1	0	1	1	1	0

- 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**



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 1 0 1 0

X_L 1 0 X_R 1 0

Y 1 0 1 1

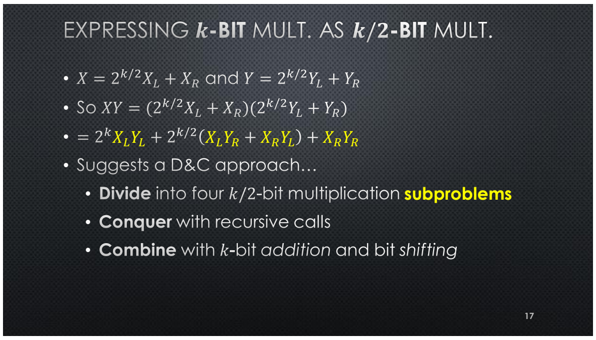
Y_L 1 0 Y_R 1 1

Thus

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

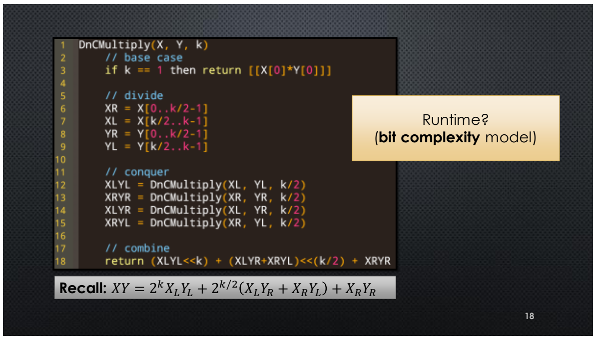
$2^{k/2}X_L$ 1 0 0 0
$+X_R$ 1 0
$=X$ 1 0 1 0

$2^{k/2}Y_L$ 1 0 0 0
$+Y_R$ 1 1
$=Y$ 1 0 1 1



EXPRESSING $\boldsymbol{k}$-BIT MULT. AS $\boldsymbol{k/2}$-BIT MULT.

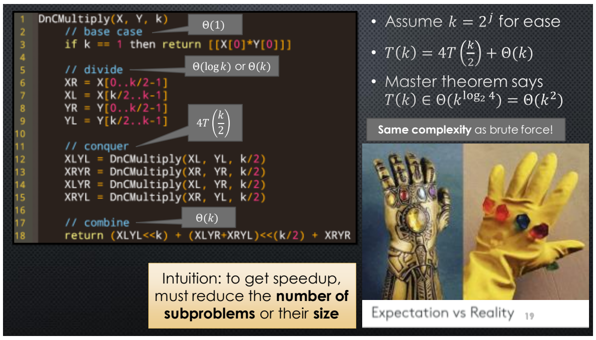
- $X = 2^{k/2}X_L + X_R$ and $Y = 2^{k/2}Y_L + Y_R$
- So $XY = (2^{k/2}X_L + X_R)(2^{k/2}Y_L + Y_R)$
- $= 2^k X_L Y_L + 2^{k/2}(X_L Y_R + X_R Y_L) + X_R Y_R$
- Suggests a D&C approach…
  - **Divide** into four $k/2$-bit multiplication **subproblems**
  - **Conquer** with recursive calls
  - **Combine**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



```
DnCMultiply(X, Y, k)
    // base case
    if k == 1 then return [[X[0]*Y[0]]]

    // divide
    XR = X[0..k/2-1]
    XL = X[k/2..k-1]
    YR = Y[0..k/2-1]
    YL = Y[k/2..k-1]

    // conquer
    XLYL = DnCMultiply(XL, YL, k/2)
    XRYR = DnCMultiply(XR, YR, k/2)
    XLYR = DnCMultiply(XL, YR, k/2)
    XRYL = DnCMultiply(XR, YL, k/2)

    // combine
    return (XLYL<<k) + (XLYR+XRYL)<<(k/2) + XRYR
```

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

Runtime?
(**bit complexity** model)



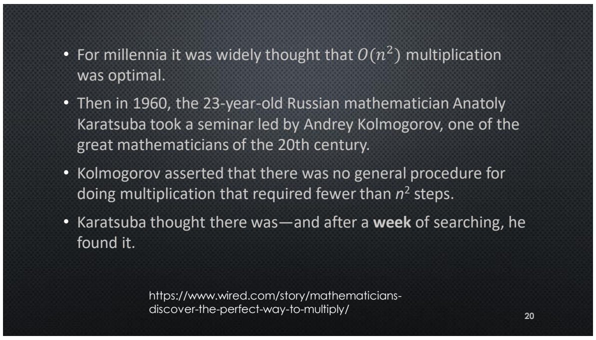
```
DnCMultiply(X, Y, k)
    // base case
    if k == 1 then return [[X[0]*Y[0]]]

    // divide
    XR = X[0..k/2-1]
    XL = X[k/2..k-1]
    YR = Y[0..k/2-1]
    YL = Y[k/2..k-1]

    // conquer
    XLYL = DnCMultiply(XL, YL, k/2)
    XRYR = DnCMultiply(XR, YR, k/2)
    XLYR = DnCMultiply(XL, YR, k/2)
    XRYL = DnCMultiply(XR, YL, k/2)

    // combine
    return (XLYL<<k) + (XLYR+XRYL)<<(k/2) + XRYR
```

- Assume $k = 2^j$ for ease
- $T(k) = 4T\left(\frac{k}{2}\right) + \Theta(k)$
- Master theorem says $T(k) \in \Theta(k^{\log_2 4}) = \Theta(k^2)$

Same complexity as brute force!

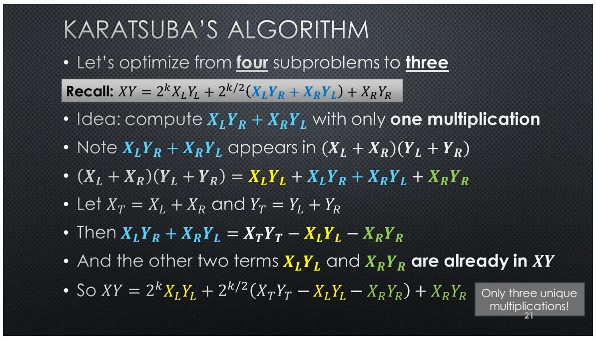
Intuition: to get speedup, must reduce the **number of subproblems** or their **size**

Expectation vs Reality



- 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.

https://www.wired.com/story/mathematicians-discover-the-perfect-way-to-multiply/



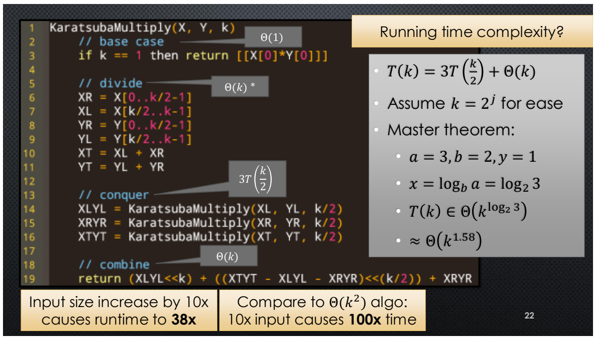
KARATSUBA'S ALGORITHM

- Let's optimize from <u>**four**</u> subproblems to <u>**three**</u>

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

- Idea: compute $\boldsymbol{X_L Y_R + X_R Y_L}$ with only **one multiplication**
- Note $\boldsymbol{X_L Y_R + X_R Y_L}$ appears in $\boldsymbol{(X_L + X_R)(Y_L + Y_R)}$
- $(X_L + X_R)(Y_L + Y_R) = \boldsymbol{X_L Y_L} + \boldsymbol{X_L Y_R} + \boldsymbol{X_R Y_L} + \boldsymbol{X_R Y_R}$
- Let $X_T = X_L + X_R$ and $Y_T = Y_L + Y_R$
- Then $\boldsymbol{X_L Y_R + X_R Y_L} = \boldsymbol{X_T Y_T} - \boldsymbol{X_L Y_L} - \boldsymbol{X_R Y_R}$
- And the other two terms $\boldsymbol{X_L Y_L}$ and $\boldsymbol{X_R Y_R}$ **are already in** $\boldsymbol{XY}$
- So $XY = 2^k \boldsymbol{X_L Y_L} + 2^{k/2}(X_T Y_T - \boldsymbol{X_L Y_L} - \boldsymbol{X_R Y_R}) + \boldsymbol{X_R Y_R}$

Only three unique multiplications!



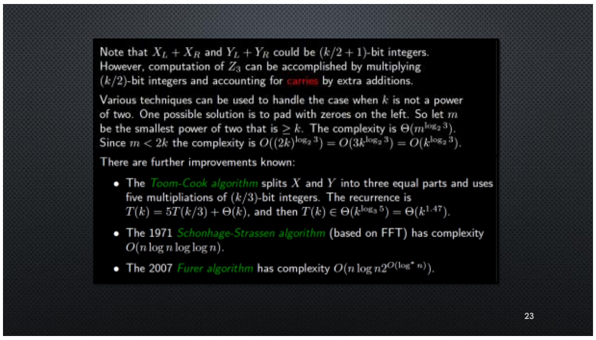
```
KaratsubaMultiply(X, Y, k)
    // base case
    if k == 1 then return [[X[0]*Y[0]]]

    // divide
    XR = X[0..k/2-1]
    XL = X[k/2..k-1]
    YR = Y[0..k/2-1]
    YL = Y[k/2..k-1]
    XT = XL + XR
    YT = YL + YR

    // conquer
    XLYL = KaratsubaMultiply(XL, YL, k/2)
    XRYR = KaratsubaMultiply(XR, YR, k/2)
    XTYT = KaratsubaMultiply(XT, YT, k/2)

    // combine
    return (XLYL<<k) + ((XTYT - XLYL - XRYR)<<(k/2)) + XRYR
```

Running time complexity?

- $T(k) = 3T\left(\frac{k}{2}\right) + \Theta(k)$
- Assume $k = 2^j$ for ease
- Master theorem:
  - $a = 3, b = 2, y = 1$
  - $x = \log_b a = \log_2 3$
  - $T(k) \in \Theta\left(k^{\log_2 3}\right)$
  - $\approx \Theta\left(k^{1.58}\right)$

Input size increase by 10x causes runtime to **38x**

Compare to $\Theta(k^2)$ algo: 10x input causes **100x** time

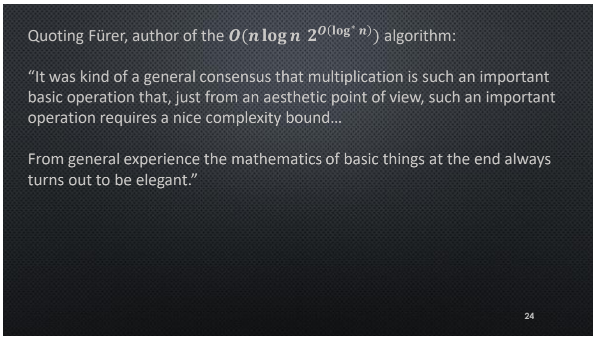


Note that $X_L + X_R$ and $Y_L + Y_R$ could be $(k/2+1)$-bit integers. However, computation of Z_3 can be accomplished by multiplying $(k/2)$-bit integers and accounting for carries by extra additions.

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 multipliations 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 $O(n \log n 2^{O(\log^* n)})$.



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

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

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

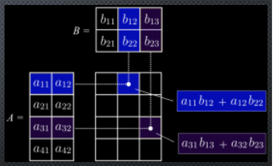
A **conditional** proof is known…
it holds if a central conjecture in the area of network coding turns out to be true. [https://arxiv.org/abs/1902.10935]

MATRIX MULTIPLICATION

- Input: **A** and **B**
- Output: their product **C=AB**
- Naïve algorithm for $n \times n$ matrices:
- For each output cell $\boldsymbol{C_{ij}}$

$$C_{ij} = DotProd(row_i(A), col_j(B)^T)$$
$$= \sum_{k=1}^{n} A_{ik} B_{kj}$$

- Running time (unit cost)?



ATTEMPTING A BETTER 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

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \left[\begin{array}{cc|cc} a_{11} & a_{12} & b_{11} & b_{12} \\ a_{21} & a_{22} & b_{21} & b_{22} \\ \hline c_{11} & c_{12} & d_{11} & d_{12} \\ c_{21} & c_{22} & d_{21} & d_{22} \end{array}\right]$$

MULTIPLYING PARTITIONED MATRICES

Let A = $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \left[\begin{array}{cc|cc} a_{11} & a_{12} & b_{11} & b_{12} \\ a_{21} & a_{22} & b_{21} & b_{22} \\ \hline c_{11} & c_{12} & d_{11} & d_{12} \\ c_{21} & c_{22} & d_{21} & d_{22} \end{array}\right]$

Let B = $\begin{bmatrix} e & f \\ g & h \end{bmatrix} = \left[\begin{array}{cc|cc} e_{11} & e_{12} & f_{11} & f_{12} \\ e_{21} & e_{22} & f_{21} & f_{22} \\ \hline g_{11} & g_{12} & h_{11} & h_{12} \\ g_{21} & g_{22} & h_{21} & h_{22} \end{array}\right]$

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

IDENTIFYING SUBPROBLEMS TO SOLVE

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

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

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

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

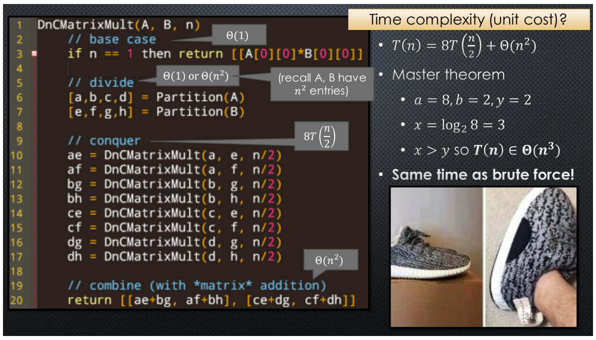
Recall $\boldsymbol{ae}$, $\boldsymbol{bg}$, etc., each represent **matrix multiplication!**

Can compute C using 8 matrix multiplications

SIZE OF SUBPROBLEMS & SUBSOLUTIONS

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

- 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
  - (and 4 additions of such matrices)



```
DnCMatrixMult(A, B, n)
    // base case
    if n == 1 then return [[A[0][0]*B[0][0]]

    // divide
    [a,b,c,d] = Partition(A)
    [e,f,g,h] = Partition(B)

    // conquer
    ae = DnCMatrixMult(a, e, n/2)
    af = DnCMatrixMult(a, f, n/2)
    bg = DnCMatrixMult(b, g, n/2)
    bh = DnCMatrixMult(b, h, n/2)
    ce = DnCMatrixMult(c, e, n/2)
    cf = DnCMatrixMult(c, f, n/2)
    dg = DnCMatrixMult(d, g, n/2)
    dh = DnCMatrixMult(d, h, n/2)

    // combine (with *matrix* addition)
    return [[ae+bg, af+bh], [ce+dg, cf+dh]]
```

$\Theta(1)$

$\Theta(1)$ or $\Theta(n^2)$

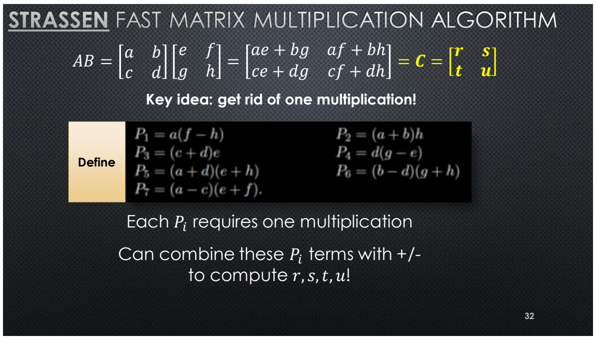
(recall A, B have n^2 entries)

$8T\left(\frac{n}{2}\right)$

$\Theta(n^2)$

Time complexity (unit cost)?

- $T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$
- Master theorem
  - $a = 8, b = 2, y = 2$
  - $x = \log_2 8 = 3$
  - $x > y$ so $\boldsymbol{T(n) \in \Theta(n^3)}$
- **Same time as brute force!**



STRASSEN FAST MATRIX MULTIPLICATION ALGORITHM

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

Key idea: get rid of one multiplication!

Define

$P_1 = a(f-h)$ $\quad P_2 = (a+b)h$

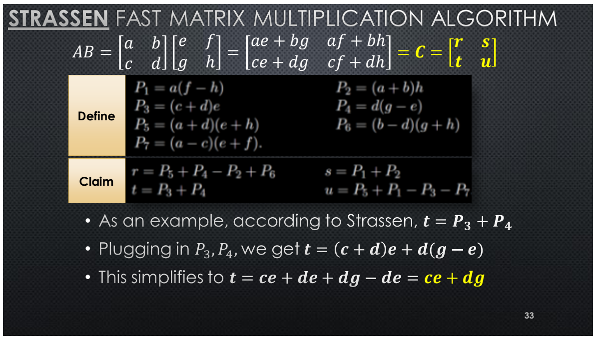
$P_3 = (c+d)e$ $\quad P_4 = d(g-e)$

$P_5 = (a+d)(e+h)$ $\quad P_6 = (b-d)(g+h)$

$P_7 = (a-c)(e+f).$

Each P_i requires one multiplication

Can combine these P_i terms with +/-
to compute r, s, t, u!



STRASSEN FAST MATRIX MULTIPLICATION ALGORITHM

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

Define

$P_1 = a(f-h)$ $\quad P_2 = (a+b)h$

$P_3 = (c+d)e$ $\quad P_4 = d(g-e)$

$P_5 = (a+d)(e+h)$ $\quad P_6 = (b-d)(g+h)$

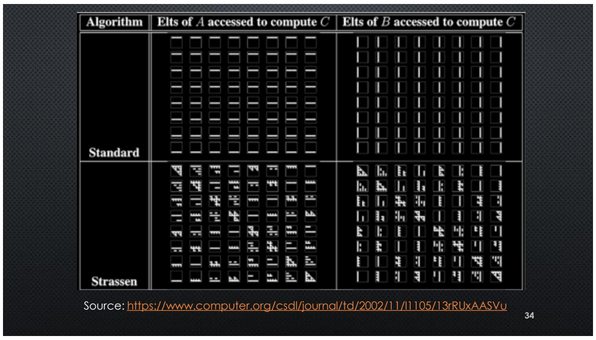
$P_7 = (a-c)(e+f).$

Claim

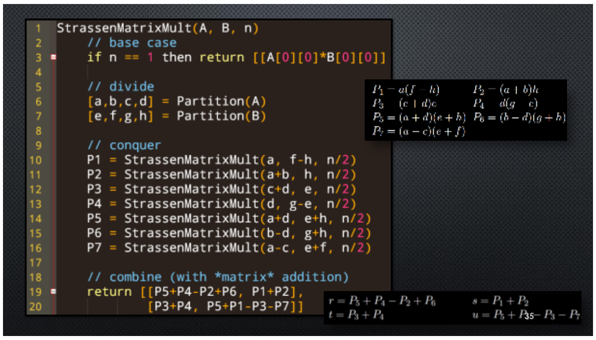
$r = P_5 + P_4 - P_2 + P_6$ $\quad s = P_1 + P_2$

$t = P_3 + P_4$ $\quad u = P_5 + P_1 - P_3 - P_7$

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



Source: https://www.computer.org/csdl/journal/td/2002/11/l1105/13rRUxAASVu



```
StrassenMatrixMult(A, B, n)
    // base case
    if n == 1 then return [[A[0][0]*B[0][0]]

    // divide
    [a,b,c,d] = Partition(A)
    [e,f,g,h] = Partition(B)

    // conquer
    P1 = StrassenMatrixMult(a, f-h, n/2)
    P2 = StrassenMatrixMult(a+b, h, n/2)
    P3 = StrassenMatrixMult(c+d, e, n/2)
    P4 = StrassenMatrixMult(d, g-e, n/2)
    P5 = StrassenMatrixMult(a+d, e+h, n/2)
    P6 = StrassenMatrixMult(b-d, g+h, n/2)
    P7 = StrassenMatrixMult(a-c, e+f, n/2)

    // combine (with *matrix* addition)
    return [[P5+P4-P2+P6, P1+P2],
            [P3+P4, P5+P1-P3-P7]]
```

$P_1 = a(f-h)$ $\quad P_2 = (a+b)h$

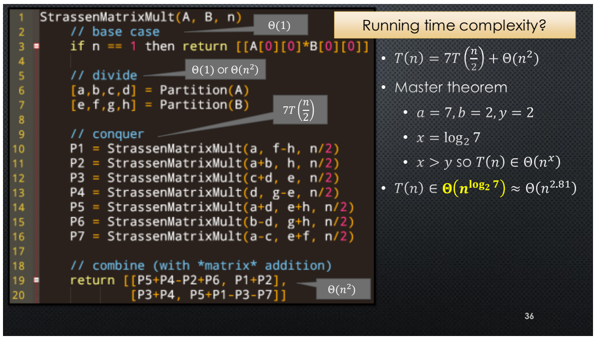
$P_3 = (c+d)e$ $\quad P_4 = d(g-e)$

$P_5 = (a+d)(e+h)$ $\quad P_6 = (b-d)(g+h)$

$P_7 = (a-c)(e+f)$

$r = P_5 + P_4 - P_2 + P_6$ $\quad s = P_1 + P_2$

$t = P_3 + P_4$ $\quad u = P_5 + P_1 - P_3 - P_7$



```
StrassenMatrixMult(A, B, n)
    // base case
    if n == 1 then return [[A[0][0]*B[0][0]]

    // divide
    [a,b,c,d] = Partition(A)
    [e,f,g,h] = Partition(B)

    // conquer
    P1 = StrassenMatrixMult(a, f-h, n/2)
    P2 = StrassenMatrixMult(a+b, h, n/2)
    P3 = StrassenMatrixMult(c+d, e, n/2)
    P4 = StrassenMatrixMult(d, g-e, n/2)
    P5 = StrassenMatrixMult(a+d, e+h, n/2)
    P6 = StrassenMatrixMult(b-d, g+h, n/2)
    P7 = StrassenMatrixMult(a-c, e+f, n/2)

    // combine (with *matrix* addition)
    return [[P5+P4-P2+P6, P1+P2],
            [P3+P4, P5+P1-P3-P7]]
```

$\Theta(1)$

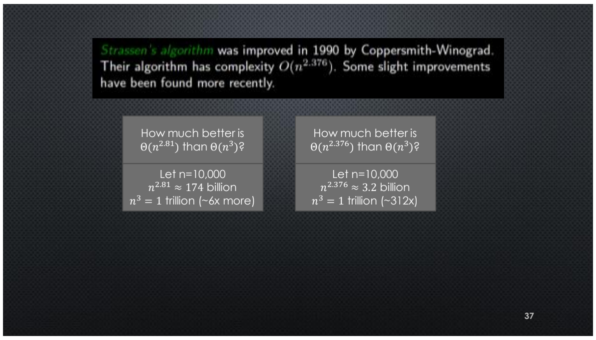
$\Theta(1)$ or $\Theta(n^2)$

$7T\left(\frac{n}{2}\right)$

$\Theta(n^2)$

Running time complexity?

- $T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$
- Master theorem
  - $a = 7, b = 2, y = 2$
  - $x = \log_2 7$
  - $x > y$ so $T(n) \in \Theta(n^x)$
- $T(n) \in \boldsymbol{\Theta\left(n^{\log_2 7}\right)} \approx \Theta(n^{2.81})$



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.

How much better is $\Theta(n^{2.81})$ than $\Theta(n^3)$?	How much better is $\Theta(n^{2.376})$ than $\Theta(n^3)$?
Let n=10,000 $n^{2.81} \approx$ 174 billion $n^3 = 1$ trillion (~6x more)	Let n=10,000 $n^{2.376} \approx$ 3.2 billion $n^3 = 1$ trillion (~312x)