Integer multiplication (Karatsuba)
• Given positive integers $y$, $z$, compute $x = y \times z$
• A naïve multiplication algorithm is below

```python
def naive_mul(y, z):
    x = 0
    while z > 0:
        if z % 2 == 1:
            x += y
        y *= 2
        z /= 2
    return x
```

Remark: these two operations can be implemented as $O(1)$ shifts
Integer multiplication

Addition takes $O(n)$ bit operations, where $n$ is the number of bits in $y$ and $z$. The naive multiplication algorithm takes $O(n)$ $n$-bit additions. Therefore, the naive multiplication algorithm takes $O(n^2)$ bit operations.

Can we multiply using fewer bit operations?
Integer multiplication

Suppose $n$ is a power of 2. Divide $y$ and $z$ into two halves, each with $n/2$ bits.

\[
\begin{array}{c|c|c}
\text{y} & a & b \\
\hline
\text{z} & c & d \\
\end{array}
\]
Integer multiplication

Then

\[ y = a2^{n/2} + b \]
\[ z = c2^{n/2} + d \]

and so

\[ yz = (a2^{n/2} + b)(c2^{n/2} + d) \]
\[ = ac2^n + (ad + bc)2^{n/2} + bd \]
Integer multiplication

This computes $yz$ with 4 multiplications of $n/2$ bit numbers, and some additions and shifts. Running time given by $T(1) = c$, $T(n) = 4T(n/2) + dn$, which has solution $O(n^2)$ by the General Theorem. No gain over naive algorithm!

Example 5.7: Consider the recurrence

$$T(n) = 4T(n/2) + n.$$

In this case, $n^{\log_2 4} = n^2 = n^2$. Thus, we are in Case 1, for $f(n)$ is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$. This means that $T(n)$ is $\Theta(n^2)$ by the master method.
Integer multiplication (Karatsuba algorithm)

• Consider the product
  \((a-b)(d-c) = (ad + bc) - (ac + bd)\)

• It contains two of the products we need \((ad\) and \(bc)\)

• Then
  \(yz = ac2^n + [(a-b)(d-c)+(ac+bd)]2^{n/2} + bd\)

• We need three multiplications of \(n/2\) bits and \(O(n)\) additional work
Integer multiplication (Karatsuba algorithm)

Therefore,

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  3T(n/2) + dn & \text{otherwise}
\end{cases}
\]

where \(c, d\) are constants.

Therefore, by our general theorem, the divide and conquer multiplication algorithm uses

\[
T(n) = O(n^{\log_3 3}) = O(n^{1.59})
\]

bit operations.
Karatsuba algorithm

```python
def multiply(y, z):
    l = max(len(y), len(z))
    if l == 1:
        return [y[0] * z[0]]
    y = [0 for i in range(len(y), l)] + y;
    z = [0 for i in range(len(z), l)] + z;
    m0 = (l + 1) / 2
    a = y[:m0]
    b = y[m0:]
    c = z[:m0]
    d = z[m0:]
```

Remark: pad y and z so that they have the same length
Karatsuba algorithm (continued)

\[ p_0 = \text{multiply}(a, c) \]
\[ p_1 = \text{multiply}((a + b), (c + d)) \]
\[ p_2 = \text{multiply}(b, d) \]

\[ z_0 = p_0 \]
\[ z_1 = \text{subtract}(p_1, (p_0 + p_2)) \]
\[ z_2 = p_2 \]

\[ z_{0\text{prod}} = z_0 + [0 \text{ for } i \text{ in range}(0, \ l)] \]
\[ z_{1\text{prod}} = z_1 + [0 \text{ for } i \text{ in range}(0, \ l/2)] \]

\[ \text{return add(add(z}_{0\text{prod}}, \ z_{1\text{prod}}), \ z_2) \]
Karatsuba vs. Naïve Algorithm

\[ O(n^{1.59}) \]

\[ O(n^2) \]
Matrix multiplication (Strassen)
**Problem**: Given two matrices $Y$ and $Z$ compute $X=Y*Z$.
Matrix multiplication

```python
def mult(Y, Z):
    X = zero(len(Y), len(Z[0]))
    for i in range(len(Y)):
        for j in range(len(Z[0])):
            for k in range(len(Z)):
                X[i][j] += Y[i][k] * Z[k][j]
    return X
```

Algorithm `mult(Y, Z)` is $O(n^3)$, can we do better?
Matrix multiplication

Divide $X, Y, Z$ each into four $(n/2) \times (n/2)$ matrices.

\[
X = \begin{bmatrix}
I & J \\
K & L
\end{bmatrix}
\]

\[
Y = \begin{bmatrix}
A & B \\
C & D
\end{bmatrix}
\]

\[
Z = \begin{bmatrix}
E & F \\
G & H
\end{bmatrix}
\]
Matrix multiplication

Then

\[ I = AE + BG \]
\[ J = AF + BH \]
\[ K = CE + DG \]
\[ L = CF + DH \]
Matrix multiplication

Let $T(n)$ be the time to multiply two $n \times n$ matrices.

$$T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  8T(n/2) + dn^2 & \text{otherwise}
\end{cases}$$

where $c, d$ are constants.
Matrix multiplication

Therefore,

\[
T(n) = 8T(n/2) + dn^2
\]

\[
= 8(8T(n/4) + d(n/2)^2) + dn^2
\]

\[
= 8^2T(n/4) + 2dn^2 + dn^2
\]

\[
= 8^3T(n/8) + 4dn^2 + 2dn^2 + dn^2
\]

\[
= 8^iT(n/2^i) + dn^2 \sum_{j=0}^{i-1} 2^j
\]

\[
= g^{\log n}T(1) + dn^2 \sum_{j=0}^{\log n-1} 2^j
\]

\[
= cn^3 + dn^2(n - 1)
\]

\[
= O(n^3)
\]
Matrix multiplication

• The naïve Divide and Conquer algorithm is no better than the straightforward algorithm
• However, it gives us an insight on the next algorithm
• Strassen’s algorithm uses only 7 multiplications instead of 8
Strassen algorithm

Compute

\[ M_1 := (A + C)(E + F) \]
\[ M_2 := (B + D)(G + H) \]
\[ M_3 := (A - D)(E + H) \]
\[ M_4 := A(F - H) \]
\[ M_5 := (C + D)E \]
\[ M_6 := (A + B)H \]
\[ M_7 := D(G - E) \]
Strassen algorithm

Then,

\[ I := M_2 + M_3 - M_6 - M_7 \]
\[ J := M_4 + M_6 \]
\[ K := M_5 + M_7 \]
\[ L := M_1 - M_3 - M_4 - M_5 \]
Strassen algorithm

\[ I := M_2 + M_3 - M_6 - M_7 \]
\[ = (B + D)(G + H) + (A - D)(E + H) \]
\[ - (A + B)H - D(G - E) \]
\[ = (BG + BH + DG + DH) \]
\[ + (AE + AH - DE - DH) \]
\[ + (-AH - BH) + (-DG + DE) \]
\[ = BG + AE \]
Strassen algorithm

\[ J := M_4 + M_6 \]
\[ = A(F - H) + (A + B)H \]
\[ = AF - AH + AH + BH \]
\[ = AF + BH \]
Strassen algorithm

\[ K := M_5 + M_7 \]
\[ = (C + D)E + D(G - E) \]
\[ = CE + DE + DG - DE \]
\[ = CE + DG \]
Strassen algorithm

\[ L := M_1 - M_3 - M_4 - M_5 \]
\[ = (A + C)(E + F) - (A - D)(E + H) \]
\[ - A(F - H) - (C + D)E \]
\[ = AE + AF + CE + CF - AE - AH \]
\[ + DE + DH - AF + AH - CE - DE \]
\[ = CF + DH \]
def strassen(Y, Z):
    if len(Y) <= 2:
        return mult(Y, Z)
    else:
        A, B, C, D = partition(Y)
        E, F, G, H = partition(Z)
        M1 = strassen(add(A, C), add(E, F))
        M2 = strassen(add(B, D), add(G, H))
        M3 = strassen(sub(A, D), add(E, H))
        M4 = strassen(A, sub(F, H))
        M5 = strassen(add(C, D), E)
        M6 = strassen(add(A, B), H)
        M7 = strassen(D, sub(G, E))
        I = sub(sub(add(M2, M3), M6), M7)
        J = add(M4, M6)
        K = add(M5, M7)
        L = sub(sub(sub(M1, M3), M4), M5)
        return recompose(I, J, K, L)
Analysis of Strassen algorithm

\[ T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  7T(n/2) + dn^2 & \text{otherwise}
\end{cases} \]

where \( c, d \) are constants.
Analysis of Strassen algorithm

\[
T(n) = 7T(n/2) + dn^2 \\
= 7(7T(n/4) + d(n/2)^2) + dn^2 \\
= 7^2T(n/4) + 7dn^2/4 + dn^2 \\
= 7^3T(n/8) + 7^2dn^2/4^2 + 7dn^2/4 + dn^2 \\
= 7^iT(n/2^i) + dn^2 \sum_{j=0}^{i-1} (7/4)^j \\
= 7^{\log n}T(1) + dn^2 \sum_{j=0}^{\log n-1} (7/4)^j \\
= cn^{\log 7} + dn^2 \left(\frac{(7/4)^{\log n} - 1}{7/4 - 1}\right) \\
= cn^{\log 7} + \frac{4}{3}dn^2 \left(\frac{n^{\log 7}}{n^2} - 1\right) \\
= O(n^{\log 7}) \\
\approx O(n^{2.8})
\]
Karatsuba vs. Naïve Algorithm

\[ O(n^2.8) \]

\[ O(n^3) \]
Discussion

- There is a large constant hidden which makes Strassen impractical, unless the matrices are large ($n > 45$) and dense
- For sparse matrices there are faster methods
- Strassen is not as numerically stable as the naïve
- Sub-matrices at each level consume space
- FYI: the current best algorithm for dense matrices runs in $O(n^{2.376})$
- Lower bound $\Omega(n^2)$ [for dense matrices]
Closest Pair
Closest Pair Problem

• Let \( P_1=(x_1,y_1), \ldots, P_n=(x_n,y_n) \) be a set \( S \) of \( n \) points in the plane

• Problem: Find the two closest points in \( S \)

• Assumptions:
  – \( n \) is a power of two
  – points are ordered by their \( x \) coordinate (if not, we can sort them in \( O(n \log n) \) time)
Closest-Pair Problem: Brute-force

• Compute the distance between every pair of distinct points
• Return the indexes of the points for which the distance is the smallest

Time complexity?
Closest-Pair: Divide and Conquer

**Step 1.** Divide the points in $S$ into two subsets $S_1$ and $S_2$ by a vertical line $x = c$ so that half the points lie to the left or on the line and half the points lie to the right or on the line ($c$ is the median of the $x$ coord)
Closest-Pair: Divide and Conquer

**Step 2.** Find recursively the closest pairs for the left and right subsets. Let $d_1, d_2$ be the distances of the two closest pairs.

Set $d = \min\{d_1, d_2\}$
Closest Pair: Divide and Conquer

**Step 3.** Consider the vertical strip $2d$-wide centered at $x = c$. Let $Y$ be the subset of points in this vertical strip of width $2d$. 
Closest Pair: Divide and Conquer

- **Observation 1**: if a pair of points $p_L, p_R$ has distance less than $d$, both points of the pair **must** be within $Y$
Closest Pair: Divide and Conquer

**Observation 2:** Since all the points within $S_i$ are at least $d$ units apart, at most 4 points can reside within the $d \times d$ square
Closest Pair: Divide and Conquer

**Proof:** Let’s suppose (for sake of contradiction) that five or more points are found in a square of size $d \times d$. Divide the square into four smaller squares of size $d/2 \times d/2$. At least one pair of points must fall within the same smaller square: these two points will be at a distance $d/\sqrt{2} < d$, which leads to a contradiction.
Closest Pair: Divide and Conquer

**Consequence:** At most 8 points can reside within the $d \times 2d$ rectangle, because on each side all points are at least $d$ unit apart.
Closest Pair: Divide and Conquer

**Step 4.** For each point $p$ in $Y$, try to find points in $Y$ that are within $d$ units of $p$. Only 7 points in $Y$ that follow $p$ need to be considered.

![Diagram showing the concept of closest pair with points within d units of p and coincident points marked.]
def closestPair(xP, yP):
    n = len(xP)
    if n <= 3:
        return bruteForceClosestPair(xP)

    Xl = xP[:n//2]
    Xr = xP[n//2:]
    Yl, Yr = [], []
    median = Xl[-1].x
    for p in yP:
        if p.x <= median:
            Yl.append(p)
        else:
            Yr.append(p)

Remark: \textbf{xP} and \textbf{yP} is the same of input points (x,y), but \textbf{xP} is sorted by x and \textbf{yP} is sorted by y

Remark: \textbf{Xl} is the first half of the points sorted by x, and \textbf{Xr} is the second half

Remark: \textbf{Yl} contains the points (sorted by y) which have a x coordinate smaller than the median
\[ \text{dl, pairl} = \text{closestPair}(X_1, Y_1) \]
\[ \text{dr, pairr} = \text{closestPair}(X_r, Y_r) \]
\[ \text{dm, pairm} = (\text{dl, pairl}) \text{ if } \text{dl} < \text{dr} \text{ else } (\text{dr, pairr}) \]

\[ \text{st} = [p \text{ for } p \text{ in } yP \text{ if } \text{abs}(p.x - \text{median}) < \text{dm}] \]
\[ \text{n_st} = \text{len(st)} \]
\[ \text{closest} = (\text{dm, pairm}) \]
\[ \text{if } \text{n_st} > 1: \]
\[ \text{for } i \text{ in range(n_st-1)}: \]
\[ \text{for } j \text{ in range}(i+1, \text{min}(i+8, \text{n_st})): \]
\[ \text{if } d(st[i], st[j]) < \text{closest}[0]: \]
\[ \text{closest} = (d(st[i], st[j]), (st[i], st[j])) \]
\[ \text{return closest} \]

Remark: variable \text{st} contains the points in the strip [median-\text{dm}, median+\text{dm}] sorted by y

Remark: \text{d(x,y)} returns the distance between x and y
Analysis of the Closest-Pair Algorithm

- We can keep the points in $Y$ stored in increasing order of their $y$ coordinates, which is maintained by merging during the execution of step 4
- We can process the points in $Y$ sequentially in linear time
- Running time is described by $T(n) = 2T(n/2) + O(n)$
- By the Master Theorem, $T(n)$ is $O(n \log n)$
Closest Pair vs. Naïve Algorithm
Extra Credit

You have 10 bags of gold coins, 10 coins per bag, 10 grams coin, but one bag of coins weigh only 9 grams per coin (because of low quality). How do you find out which bag contains low quality gold coins? You may use a scale only one time.

Up to 2%