3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

A refresh from our work in Merge Sort the divide and conquer model follows these steps:

1) Divide a problem into subproblems.

2) Conquer the subproblems by continually dividing the subproblems into even smaller subproblems (using recursion) until you reach a base case which you can solve. For example, with sorting numbers in merge sort the base case was if a subproblem has 1 or Ø elements by default the subproblem is sorted.

3) Combine solutions of subproblems into a solution that solves the original problem.

COUNTING INVERSIONS ALGORITHM EXAMPLE

THE PROBLEM: Given an Array, A, containing the numbers $1, 2, \ldots, n$ in some arbitrary order provide the total number of inversions in the Array.

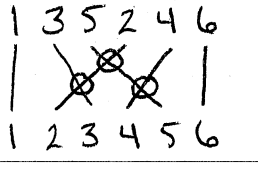
An inversion is a pair (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

WHY THIS PROBLEM MATTERS - One use of counting inversions provides a measure of numerical similarity between two ranked lists. This allows shopping suggestion algorithms to work. For example, you bought products x and y. Others who have bough x and y typically also by z, so the shopping algorithm would provide a suggestion or advertisement for z to you. COLLABORATIVE FILTERING

GIVEN: ARRAY $A = (1, 3, 5, 2, 4, 6)$

FIND: HOW MANY INVERSIONS THERE ARE WITH A SORTED LIST $(1, 2, 3, 4, 5, 6)$.

VISUALIZING THE PROBLEM:



INVERSIONS AT:
(3, 2)
(5, 2)
(5, 4)

There are three inversions in this small sample size. Scaling the problem to an arbitrarily large input array for A would be tedious for a human, but quick for a computer. Let's design an algorithm.

(con't on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

Counting Inversions (con't)

Brute Force Solution:

Run nested for loops and check every index for another position past the index where the value at the index is greater than the following values. When this happens increment an inversion counter variable to collect the number of inversions.

```
for i=1 to n-1:
  for j=1 to n:
    if A[i] > A[j]:
      invert_total ++
```

Since the outer loop runs n times and the inner loop runs n times for every iteration there are $n \cdot n$ operations. The brute force algorithm runs in n^2 time.

The Brute Force Algorithm can be improved somewhat by recognizing the inner loop only has to check values after the outer loops index value. There would be about n checks in the inner loop when i=1 and only 1 check on the inner loop when i = n-1. Therefore, instead of running n times per iteration of the outer loop the inner loop would run an average of $\frac{n}{2}$ times per iteration of the outer loop this leaves us with an algorithm that runs with $n \cdot \frac{n}{2}$ operations. However, with asymptotic analysis big O is still n^2.

```
for i=1 to n-1:
  for j=i+1 to n:
    if A[i] > A[j]
      invert_total ++
```

How can we do better?

Merge Sort Solution $O(n \lg n)$

Key Idea #1: Divide and Conquer

Let's call an inversion $[i, j]$ where $i < j$:

- LEFT if $i, j \le n/2$ } DIVIDE
- RIGHT if $i, j > n/2$ } DIVIDE
- SPLIT if $i \le n/2 < j$ } CONQUER

Pseudocode:

```
count_inversions(array A, length n):
  base case -> if n ≤ 1: return Ø
  else:
    RECURSION (DIVIDE) { x = count_inversions(1st half of A, n/2)
                       { y = count_inversions(2nd half of A, n/2)
    CONQUER -> z = count_split_inversions(A, n)
    return x+y+z
```

(con't on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

COUNTING INVERSIONS (cont)

MERGE SORT SOLUTION (cont)

From the pseudocode the divide portion will run in $\lg n$ time since the problem is split in 2 with every level of recursion. Since we want $O(n \lg n)$ that leaves the count_split_inversions function call linear time to run.

GOAL: Implement count_split_inversions in linear, $O(n)$, time.

KEY IDEA #2: Since the recursive calls run in $\lg n$ time have the recursive call count the inversions and sort.

UPDATED PSEUDOCODE:

```
sort_and_count_inversions(array A, length n):
  if n ≤ 1:
    return ∅
  else:
    x = sort_and_count_inversions(1st half of A, n/2)
    y = sort_and_count_inversions(2nd half of A, n/2)
    z = merge_and_count_split_inversions(x, y, n)
  return x+y+z
```

From our original array, A, of (1, 3, 5, 2, 4, 6) suppose we've gone through the recursions and are recombining the first solved subproblems x = (1, 3, 5) and y = (2, 4, 6) with the function merge_and_count_split_inversions.

x = (1, 3, 5) y = (2, 4, 6)

i=1 j=1

merge_and_count_split_inversions(x, y, n)

1	2	3	4	5	6
i=1	i=2	i=2	i=3	i=3	i=3
j=1	j=1	j=2	j=2	j=2	j=3
	x[i] > y[j]		x[i] > y[j]		
	inv_count += len(x) - j		inv_count += len(x) - j		
	inv_count = ∅ + 2		inv_count = 2 + 1		
	inv_count = 2		inv_count = 3		

(cont on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

COUNTING INVERSIONS (cont)

MERGE SORT SOLN (cont)

CLAIM: The number of split inversions involving an element, e, of the second array, y, are precisely the number of elements left in the first array, x, when e is copied into the merged array.

PROOF: Let v be an element of the first array, x.

1) If v is copied to the merged array before e, then $v < e$ and there was no inversion involving v and e.

2) If e is copied to the merged array before v, then $v > e$ and there is a split inversion involving v and e.

PSEUDOCODE FOR merge_and_count_split_inversions (x, y, n)

ASSUME END CASES WHERE AN ARRAY IS EXHAUSTED ARE CONSIDERED

LET M BE THE MERGED (OUTPUT) ARRAY
k REPRESENT THE INDEX POSITION IN M.

```
FOR k = 1 TO k = n:
  IF x[i] < y[j]:
    M[k] = x[i]
    i ++
  ELSE  # x[i] > y[j] = inversion
    M[k] = y[j]
    inversion_count = inversion_count + (len(x) - i + 1)
    j ++
RETURN inversion_count
```

Very similar to merge step in merge sort

From the pseudocode above each element is touched at most once or n times, therefore the running time of this function meets our requirement to run in O(n) or linear time.

TOTAL NUMBER OF OPERATIONS FOR COUNTING INVERSION:

$$\text{TOTAL} = \underbrace{\#\text{ OF RECURSIONS}}_{\text{sort\_and\_count}} \underbrace{(\#\text{ OPS / RECURSION})}_{\text{merge\_and\_count}}$$

$$O(k) = \lg n\,(n) = \underline{\underline{n \lg n}}$$ BETTER THAN BRUTE FORCE $O(n^2)$

```
# counting_inversion.py
# A program implemented in O(n lg n) time to demonstrate counting inversions.

def main():
    A = [1, 5, 3, 6, 4, 2]
    print("The input list is", A)

    num_inversions, A = sort_and_count_inversions(A)

    print("The comparison list is", A)

    print("The number of inversions between the input list and the target list is
{}.".format(num_inversions))

# Divide and conquer!
def sort_and_count_inversions(A):
    # Base case: An empty or one element length list is by default sorted and
    # cannot have an inversion.
    if len(A) < 2:
        return 0, A

    # Divide with recursion
    list_lower_half = A[0:len(A)//2]
    list_upper_half = A[len(A)//2:]
    num_left_inversions, list_lower_half =
sort_and_count_inversions(list_lower_half)
    num_right_inversions, list_upper_half =
sort_and_count_inversions(list_upper_half)
    # Conquer with merge sort
    num_split_inversions, sorted_list = count_split_inversions(list_lower_half,
                                                    list_upper_half)

    return num_left_inversions + num_right_inversions + num_split_inversions,
   sorted_list

def count_split_inversions(x, y):
    sorted_list = []
    inversion_count = i = j = 0

    for _ in range(len(x) + len(y)):
        # Check if we've reached the end of a list
        if i >= len(x):
            # There are no inversions remaining as every element left in list y
            # is greater than anything that was in list x.
            while j < len(y):
                sorted_list.append(y[j])
                j += 1
            break
```

```
        elif j >= len(y):
            # Every element remaining in list x is greater than any element in
            # list y and has been accounted for below.
            while i < len(x):
                sorted_list.append(x[i])
                i += 1
            break

        # Not a split inversion
        if x[i] < y[j]:
            sorted_list.append(x[i])
            i += 1
        else:
            sorted_list.append(y[j])
            inversion_count += len(x) - i
            j += 1

    return inversion_count, sorted_list

if __name__ == "__main__":
    main()
```

```
package main

// An example program to demonstrate counting inversions between two slices
// in O(n lg n) time using principles of merge sort.

import "fmt"

func main() {
        xi := []int{1, 5, 3, 6, 4, 2}
        fmt.Println("The input slice is", xi)
        fmt.Println("The comparision slice is [1 2 3 4 5 6]")

        numInversions, _ := sortAndCountInversions(xi)

        fmt.Printf("\nThere are %d inversions between the two lists.",
numInversions)
}

func sortAndCountInversions(xi []int) (int, []int) {
        // Base case: An empty slice or slice of length 1 is by default sorted and
        // contains no inversions.
        if len(xi) < 2 {
                return 0, xi
        }

        // Divide with recursion
        sliceLowerHalf := xi[0 : len(xi)/2]
        sliceUpperHalf := xi[len(xi)/2:]

        numLeftInversions, sliceLowerHalf := sortAndCountInversions(sliceLowerHalf)
        numRightInversions, sliceUpperHalf := sortAndCountInversions(sliceUpperHalf)

        // Conquer with merge sort principles
        numSplitInversions, sortedList := countSplitInversions(sliceLowerHalf,
sliceUpperHalf)

        return numLeftInversions + numRightInversions + numSplitInversions,
sortedList
}

func countSplitInversions(xLower, xUpper []int) (int, []int) {
        // Initialize counter and indices
        var countInversions, i, j int
        sortedList := make([]int, len(xLower)+len(xUpper))

        for k := 0; k < len(sortedList); k++ {
                // Check if we've exhausted all values in one of the slices
                if i >= len(xLower) {
```

```
                        // All values left in xUpper were greater than any value in
xLower
                        for j < len(xUpper) {
                                sortedList[k] = xUpper[j]
                                j++
                                k++
                        }
                        break
                } else if j >= len(xUpper) {
                        // All values left in xLower are greater than any value in
xUpper
                        // and are inversions and have been accounted for below.
                        for i < len(xLower) {
                                sortedList[k] = xLower[i]
                                i++
                                k++
                        }
                        break
                }

                // Not an inversion
                if xLower[i] < xUpper[j] {
                        sortedList[k] = xLower[i]
                        i++
                } else {
                        sortedList[k] = xUpper[j]
                        countInversions += len(xLower) - i
                        j++
                }
        }
        return countInversions, sortedList
}
```

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

STRASSEN'S SUBCUBIC MATRIX MULTIPLICATION ALGORITHM

REFRESH OF MATRIX MULTIPLICATION (DOT PRODUCT)

$$\text{ROW}_i^T \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} \cdot \overset{\text{COLUMN}_j}{\begin{bmatrix} Y_{1,1} & Y_{2,1} \\ Y_{2,1} & Y_{2,2} \end{bmatrix}} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} & Z_{2,2} \end{bmatrix}$$

i = ROW
j = COLUMN

WHERE $Z_{i,j} = (i^{th} \text{ row of } x) \circ (j^{th} \text{ column of } y)$

SO $Z_{2,1} = X_{2,1}(Y_{1,1}) + X_{2,2}(Y_{2,1})$

OR $Z_{i,j} = \sum_{k=1}^{n} X_{i,k}(Y_{k,j})$

Given square matrices with arbitrary dimensions of $n \times n$, there are $2n^2$ inputs and n^2 outputs giving a theoretical lower limit of $O(n^2)$ runtime. This does not happen in reality.

STANDARD APPROACH ALGORITHM ANALYSIS

GIVEN $n=2$ square matrices with arbitrary elements

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+ch \end{bmatrix}$$

From above's refresh $Z_{i,j} = \sum_{k=1}^{n} X_{i,k}(Y_{k,j})$ ← RUNTIME = $O(n)$

For each position in the output matrix there are n operations. Since there are n^2 positions there must be $n^2 \cdot n$ operations to completely compute the output matrix, or $O(n^3)$. We are an order higher runtime than the theoretical minimum. Can we do better?

(cont on next sheet)

COMET 3-0235 — 50 SHEETS — 5 SQUARES 3-0236 — 100 SHEETS — 5 SQUARES 3-0237 — 200 SHEETS — 5 SQUARES 3-0137 — 200 SHEETS — FILLER

STRASSEN'S SUBCUBIC MATRIX MULT. ALGO (cont)

* This algorithm only works on square matrices! Be cognizant to overcome this deficiency you can either pad the original matrices or break them up into smaller square matrices, 12 x 4 becomes 3 - 4x4 matrices.

Strassen's algorithm uses the principles of divide and conquer.

1) Divide a problem into subproblems
2) Conquer subproblems by dividing them further until you reach an easily solvable base case.
3) Combine solutions of subproblems into a solution that solves the original problem.

DIVIDE

Break the original matrices into blocks with dimensions $n/2 \times n/2$

$$X = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right], \quad Y = \left[\begin{array}{c|c} E & F \\ \hline G & H \end{array}\right]$$

where A through H are all $n/2 \times n/2$ submatrices (blocks).

$$\therefore X \cdot Y = \left[\begin{array}{c|c} AE + BG & AF + BH \\ \hline CE + DG & CF + DH \end{array}\right]$$

similar to standard approach so far, but with ability to continue recursively

Here is where Strassen's algorithm gets its savings. Strassen's algorithm only makes 7 recursive calls instead of 8 using the standard algorithm. This becomes more beneficial as matrices get larger and the number of recursions needed to get to the base case increases.

CONQUER: The seven recursive calls are

$P_1 = A(F - H)$ $\quad P_5 = (A + D)(E + H)$

$P_2 = (A + B)H$ $\quad P_6 = (B - D)(G + H)$

$P_3 = (C + D)E$ $\quad P_7 = (A - C)(E + F)$

$P_4 = D(G - E)$

(cont on next sheet)

STRASSEN'S SUBCUBIC MATRIX MULT. ALGO (cont'd)

COMBINE:

$$X \cdot Y = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix} = \begin{bmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{bmatrix}$$

PROOF: $AE+BG = P_5+P_4-P_2+P_6$

$$\overbrace{[(A+D)(E+H)]}^{P_5} + \overbrace{[D(G-E)]}^{P_4} - \overbrace{[(A+B)H]}^{P_2} + \overbrace{[(B-D)(G+H)]}^{P_6} \rightarrow$$

$$AE + \cancel{AH} + \cancel{DE} + DH + DG - DE - \cancel{HA} - HB + BG + \cancel{BH} - \cancel{DG} - DH \rightarrow$$

$\underline{AE+BG}$ This holds for blocks.

ANALYSIS: There are still n^2 inputs, but instead of n calculations to completely compute the output matrix due to recursion there are $8n-1$ recursive calls to calculate the products. So, the algorithm is no longer n^3, but also not n^2, however, this is as good as we can do using this algorithm.

(cont'd on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

Python Implementation of Strassen's Algorithm

```
1  # strassens_algorithm.py
2
3  # A demonstration of Strassen's subcubic runtime matrix
4  # multiplication algorithm on square matrices using the divide
5  # and conquer model.
6
7  import numpy as np
8
9  def main():
10   m1 = np.array([[1, 2], [3, 4]])
11   m2 = np.array([[5, 6], [7, 8]])
12
13   print("The first square matrix is:")
14   print(m1)
15   print("The second square matrix is:")
16   print(m2)
17
18   output_matrix = strassens_algorithm(m1, m2)
19   expected_matrix = np.matmul(m1, m2)
20
21   print("When taking the dot product of m1 and m2 we \
22         expected an output matrix of:")
23   print(expected_matrix)
24   print("Using Strassen's subcubic runtime matrix multiplication \
25         algorithm we got:")
26   print(output_matrix)
27
28 def strassens_algorithm(m1, m2):
29   # Base case: A matrix of shape 1x1 is solved by default
30   if np.shape(m1) == (1, 1):
31     return np.asscalar(m1 * m2)
32
33   # Pre-calculate the n/2 value. We'll be using it a lot and
34   # it's the same for matrix 1 and matrix 2 since they have
35   # equivalent square dimensions.
36   n_over_2 = len(m1) // 2
37
38   # Divide
39   # Create submatrix blocks from quadrants of m1
40   # | A B |
41   # | C D |
```

(Cont on next sheet)

PYTHON IMPLEMENTATION OF STRASSEN'S ALGORITHM (cont)

```
42  A = m1[Ø: n_over_2, Ø: n_over_2]
43  B = m1[Ø: n_over_2, n_over_2:]
44  C = m1[n_over_2:, Ø: n_over_2]
45  D = m1[n_over_2:, n_over_2:]
46
47  # Create submatrix blocks from quadrants of m2
48  # | E  F |
49  # | G  H |
50  E = m2[Ø: n_over_2, Ø: n_over_2]
51  F = m2[Ø: n_over_2, n_over_2:]
52  G = m2[n_over_2:, Ø: n_over_2]
53  H = m2[n_over_2:, n_over_2:]
54
55  # Calculate the 7 products: (elements matrix 1) * (elements matrix 2)
56  p1 = strassens_algorithm(A, F-H)      # p1 = A * (F-H)
57  p2 = strassens_algorithm(A+B, H)      # p2 = (A+B)* H
58  p3 = strassens_algorithm(C+D, E)      # p3 = (C+D) * E
59  p4 = strassens_algorithm(D, G-E)      # p4 = D * (G-E)
60  p5 = strassens_algorithm(A+D, E+H)    # p5 = (A+D) * (E+H)
61  p6 = strassens_algorithm(B-D, G+H)    # p6 = (B-D) * (G+H)
62  p7 = strassens_algorithm(A-C, E+F)    # p7 = (A-C) * (E+F)
63
64  return np.array([[p5+p4-p2+p6, p1+p2],
65                   [p3+p4, p1+p5-p3-p7]])
66
67 if __name__ == "__main__":
68  main()
```

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

Go Implementation of Strassen's Algorithm

```
1  package main
2
3  // A demonstration of Strassen's subcubic runtime matrix
4  // multiplication algorithm on square matrices using the
5  // divide and conquer model.
6
7  import (
8          "fmt"
9          "gonum.org/v1/gonum/mat"
10 )
11
12 func main() {
13   m1 := mat.NewDense(2, 2, []float64{
14         1, 2,
15         3, 4,
16   })
17   m2 := mat.NewDense(2, 2, []float64{
18         5, 6,
19         7, 8,
20   })
21
22   // Expected output
23   var o mat.Dense
24   o.Mul(m1, m2)
25
26   // Prefix number of spaces = 5, the number "m1 =" takes up.
27   fmt.Println("m1 =", mat.Formatted(m1, mat.Prefix("     "),
28               mat.Squeeze()))
29   fmt.Println("m2 =", mat.Formatted(m2, mat.Prefix("     "),
30               mat.Squeeze()))
31
32   fmt.Println("Expected output:")
33   fmt.Println(mat.Formatted(&o, mat.Squeeze()))
34
35   fmt.Println("Strassen's algorithm output:")
36   outputMatrix := strassensAlgorithm(m1, m2)
37   fmt.Println(mat.Formatted(outputMatrix, mat.Squeeze()))
38 }
```

(cont on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

GO IMPLEMENTATION OF STRASSEN'S ALGORITHM (cont)

```
39 func strassensAlgorithm(m1, m2 *mat.Dense) *mat.Dense {
40   var productMatrix mat.Dense
41
42   rows, _ := m1.Dims()
43   // Base case: A square matrix of dimension 1x1 is solved
44   if rows == 1 {
45     productMatrix.Mul(m1, m2)
46     return &productMatrix
47   }
48
49   // Break up m1 into submatrices from quadrant blocks.
50   //|a b|
51   //|c d|
52   a := m1.Slice(0, rows/2, 0, rows/2)
53   b := m1.Slice(0, rows/2, rows/2, rows)
54   c := m1.Slice(rows/2, rows, 0, rows/2)
55   d := m1.Slice(rows/2, rows, rows/2, rows)
56
57   // Break up m2 into submatrices from quadrant blocks.
58   //|e f|
59   //|g h|
60   e := m2.Slice(0, rows/2, 0, rows/2)
61   f := m2.Slice(0, rows/2, rows/2, rows)
62   g := m2.Slice(rows/2, rows, 0, rows/2)
63   h := m2.Slice(rows/2, rows, rows/2, rows)
64
65   // Calculate the 7 products: (elements matrix 1) * (elements m2)
66   // Resultants are used for intermediate step calcs, add/subtract.
67   var resultant1  mat.Dense
68   var resultant2  mat.Dense
69
70   // p1 = a * (f - h)
71   resultant1.Sub(f, h)
72   p1 := strassensAlgorithm(d.(*mat.Dense), &resultant1)
73
74   // p2 = (a+b) * h
75   resultant1.Add(a, b)
76   p2 := strassensAlgorithm(&resultant1, h.(*mat.Dense))
77
78   // p3 = (c+d) * e
79   resultant1.Add(c,d)
80   p3 := strassensAlgorithm(&resultant1, e.(*mat.Dense))
```

(cont on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

Go Implementation of Strassen's Algorithm (cont)

```
81  // p4 = d * (g-e)
82  resultant1.Sub(g, e)
83  p4 := strassensAlgorithm(d.(*mat.Dense), &resultant1)
84
85  // p5 = (a+d) * (e+h)
86  resultant1.Add(a, d)
87  resultant2.Add(e, h)
88  p5 := strassensAlgorithm(&resultant1, &resultant2)
89
90  // p6 = (b-d) * (g+h)
91  resultant1.Sub(b, d)
92  resultant2.Add(g, h)
93  p6 := strassensAlgorithm(&resultant1, &resultant2)
94
95  // p7 = (a-c) * (e+f)
96  resultant1.Sub(a, c)
97  resultant2.Add(e, f)
98  p7 := strassensAlgorithm(&resultant1, &resultant2)
99
100 // Product matrix quadrants
101 // | i  j |
102 // | k  l |
103 var i, j, k, l mat.Dense
104
105 // i = p5 + p4 - p2 + p6
106 resultant1.Add(p5, p4)
107 resultant2.Sub(&resultant1, p2)
108 i.Add(&resultant2, p6)
109
110 // j = p1 + p2
111 j.Add(p1, p2)
112
113 // k = p3 + p4
114 k.Add(p3, p4)
115
116 // l = p1 + p5 - p3 - p7
117 resultant1.Add(p1, p5)
118 resultant2.Sub(&resultant1, p3)
119 l.Sub(&resultant2, p7)
```

Go Implementation of Strassen's Algorithm (cont)

```
120  // Combine and build the product matrix
121  var topHalf, bottomHalf mat.Dense
122  topHalf.Augment(&i, &j)
123  bottomHalf.Augment(&k, &l)
124  productMatrix.Stack(&topHalf, &bottomHalf)
125
126  return &productMatrix
127 }
```

COMET 3-0235 — 50 SHEETS — 5 SQUARES 3-0236 — 100 SHEETS — 5 SQUARES 3-0237 — 200 SHEETS — 5 SQUARES 3-0137 — 200 SHEETS — FILLER