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 0 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, ..., n in some arbitrary order provide the total number of inversions in the Array.

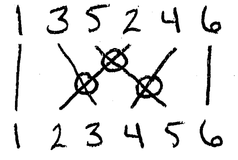
An inversion is a pair (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

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: A



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)

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 $i < j$:

$$\left.\begin{array}{l}\text{LEFT if } i, j \le n/2 \\ \text{RIGHT if } i, j > n/2\end{array}\right\} \text{DIVIDE}$$

$$\text{SPLIT if } i \le n/2 < j \;\} \text{ CONQUER}$$

PSEUDOCODE:

```
count_inversions(array A, length n):
base case -> if n ≤ 1: return 0
             else:
RECURSION {      x = count_inversions(1st half of A, n/2)
(DIVIDE)  {      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)

Counting Inversions (con't)

Merge Sort Solution (con't)

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 0
  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.

$$x = (\underset{\substack{\uparrow \\ i=1}}{1}, 3, 5) \qquad y = (\underset{\substack{\uparrow \\ j=1}}{2}, 4, 6)$$

merge_and_count_split_inversions(x, y, n)

1	2	3	4	5	6
$i=1$	$i=2$	$i=2$	$i=3$	$i=3$	$i=3$
$j=1$	$j=1$	$j=2$	$j=2$	$j=2$	$j=3$
	$x[i] > y[j]$ inv_count += len(x) - j inv_count = 0 + 2 inv_count = 2		$x[i] > y[j]$ inv_count += len(x) - j inv_count = 2 + 1 inv_count = 3		

(con't on next sheet)

COUNTING INVERSIONS (con't)

MERGE SORT SOL'N (con't)

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:

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

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

PYTHON IMPLEMENTATION OF COUNTING INVERSION

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

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

    num_inversions = sort_and_count_inversions(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

    # Divide with recursion
    list_lower_half = A[0:len(A)//2]
    list_upper_half = A[len(A)//2:]

    num_left_inversions = sort_and_count_inversions(list_lower_half)
    num_right_inversions = sort_and_count_inversions(list_upper_half)

    # Conquer with merge sort
    num_split_inversions = count_split_inversions(list_lower_half,
                                                  list_upper_half)

    return num_left_inversions + num_right_inversions
                               + num_split_inversions

def count_split_inversions(x, y):
    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.
            break
```

(con't on next sheet)

PYTHON IMPLEMENTATION OF COUNTING INVERSION (cont)

```
        elif j >= len(y):
            # Every element remaining in list x is greater than any
            # element in list y.
            while i < len(x):
                inversion_count += len(x) - i
                i += 1
            break

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

    return inversion_count

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

GO IMPLEMENTATION OF COUNTING INVERSIONS

```
package main

//An example program to demonstrate counting inversions between
// two slices in O(n lgn) 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 comparison 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 {
  // Base case: An empty slice or slice of length 1 is by default
  // sorted and contains no inversions.
  if len(xi) < 2:
    return 0

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

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

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

  return numLeftInversions+ numRightInversions + numSplitInversions
}

func countSplitInversions(xLower, xUpper []int) int {
  // Initialize counter and indices
  var countInversions, i, j int
```

(con't on next sheet)

GO IMPLEMENTATION OF COUNTING INVERSIONS (cont)

```
for k:=0; k < len(xLower)+len(xUpper); 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
    break
  } else if j >= len(xUpper) {
    // All values left in xLower are greater than any value in
    // xUpper and are inversions
    for i< len(xLower) {
      countInversions += len(xLower) - i
      i++
    }
    break
  }

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

STRASSEN'S SUBCUBIC MATRIX MULTIPLICATION ALGORITHM

REFRESH OF MATRIX MULTIPLICATION (DOT PRODUCT)

$$\text{ROW}_1 \begin{bmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{bmatrix} \cdot \begin{array}{c} \text{COLUMN} \\ \begin{bmatrix} Y_{1,1} & Y_{2,1} \\ Y_{2,1} & Y_{2,2} \end{bmatrix} \end{array} = \begin{bmatrix} Z_{1,1} & Z_{1,2} \\ Z_{2,1} & Z_{2,2} \end{bmatrix}$$

i = ROW
j = COLUMN

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

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 $Z_{i,j} = \underbrace{\sum_{k=1}^{n} x_{i,k}(y_{k,j})}_{}$ 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?

(con't on next sheet)

STRASSEN'S SUBCUBIC MATRIX MULT. ALGO (CON'T)

* 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, 12x4 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

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

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

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

$P_4 = D(G-E)$

(con't on next sheet)

STRASSEN'S SUBCUBIC MATRIX MULT. ALGO (con't)

COMBINE:

$$X \cdot Y = \left[\begin{array}{c|c} AE+BG & AF+BH \\ \hline CE+DG & CF+DH \end{array}\right] = \left[\begin{array}{c|c} P_5+P_4-P_2+P_6 & P_1+P_2 \\ \hline P_3+P_4 & P_1+P_5-P_3-P_7 \end{array}\right]$$

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

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

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

$\underline{\underline{AE+BG}}$ 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 $n-1$ 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.

(Con't on next sheet)

PYTHON IMPLEMENTATION OF STRASSEN'S ALGORITHM

```
# strassens_algorithm.py

# A demonstration of Strassen's subcubic runtime matrix
# multiplication algorithm on square matrices using the divide
# and conquer model.

import numpy as np

def main():
  m1 = np.array([[1, 2], [3,4]])
  m2 = np.array([[5, 6], [7, 8]])

  print("The first square matrix is:")
  print (m1)
  print("The second square matrix is:")
  print(m2)

  output_matrix = strassens_algorithm(m1, m2)
  expected-matrix= np.matmul(m1, m2)

  print("When taking the dot product of m1 and m2 we \
          expected an output matrix of:")
  print(expected-matrix)
  print("Using Strassen's subcubic runtime matrix multiplication\
          algorithm we got:")
  print(output-matrix)

def strassens_algorithm(m1, m2):
  # Base case: A matrix of shape 1x1 is solved by default
  if np.shape(m1) == (1, 1):
    return np.asscalar(m1 * m2)

  # Pre-calculate the n/2 value. We'll be using it a lot and
  # it's the same for matrix 1 and matrix 2 since they have
  # equivalent square dimensions.
  n_over_2 = len(m1) // 2

  # Divide
  # Create submatrix blocks from quadrants of m1
  # |A B|
  # |C D|
```

(Con't on next sheet)

PYTHON IMPLEMENTATION OF STRASSEN'S ALGORITHM (cont)

```
    A = m1[0: n_over_2, 0: n_over_2]
    B = m1[0: n_over_2, n_over_2:]
    C = m1[n_over_2:, 0: n_over_2]
    D = m1[n_over_2:, n_over_2:]

    # Create submatrix blocks from quadrants of m2
    # | E F |
    # | G H |
    E = m2[0: n_over_2, 0: n_over_2]
    F = m2 [0: n_over_2, n_over_2:]
    G = m2 [n_over_2:, 0: n_over_2]
    H = m2[n_over_2:, n_over_2:]

    # Calculate the 7 products: (elements matrix 1) * (elements matrix 2)
    p1 = strassens_algorithm(A, F-H)        # p1 = A * (F-H)
    p2 = strassens_algorithm(A+B, H)        # p2 = (A+B) * H
    p3 = strassens_algorithm(C+D, E)        # p3 = (C+D) * E
    p4 = strassens_algorithm(D, G-E)        # p4 = D * (G-E)
    p5 = strassens_algorithm(A+D, E+H)      # p5 = (A+D) * (E+H)
    p6 = strassens_algorithm(B-D, G+H)      # p6 = (B-D) * (G+H)
    p7 = strassens_algorithm(A-C, E+F)      # p7 = (A-C) * (E+F)

    return np.array([[p5+p4-p2+p6, p1+p2],
                     [p3+p4, p1+p5-p3-p7]])

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

GO IMPLEMENTATION OF STRASSEN'S ALGORITHM

```
package main

// A demonstration of Strassen's subcubic runtime matrix
// multiplication algorithm on square matrices using the
// divide and conquer model.

import (
        "fmt"
        "gonum.org/v1/gonum/mat"
)

func main() {
  m1 := mat.NewDense(2, 2, []float64{
        1, 2,
        3, 4,
  })
  m2 := mat.NewDense(2, 2, []float64{
        5, 6,
        7, 8,
  })

  // Expected output
  var o mat.dense
  o.Mul(m1, m2)

  // Prefix number of spaces = 5, the number "m1 =" takes up.
  fmt.Println("m1 =", mat.Formatted(m1, mat.Prefix("     "),
              mat.Squeeze()))
  fmt.Println("m2 =", mat.Formatted(m2, mat.Prefix("     "),
              mat.Squeeze()))

  fmt.Println("Expected output:")
  fmt.Println(mat.Formatted(&o, mat.Squeeze()))

  fmt.Println("Strassen's algorithm output:")
  outputMatrix := strassensAlgorithm(m1, m2)
  fmt.Println(mat.Formatted(outputMatrix, mat.Squeeze()))
}
```

(con't on next sheet)

GO IMPLEMENTATION OF STRASSEN'S ALGORITHM (cont)

```
func strassensAlgorithm(m1, m2 *mat.Dense) *mat.Dense {
  var productMatrix mat.Dense

  rows, _ := m1.Dims()
  // Base case: A square matrix of dimension 1x1 is solved
  if rows == 1 {
    productMatrix.Mul(m1, m2)
    return &productMatrix
  }

  // Break up m1 into submatrices from quadrant blocks.
  //|a b|
  //|c d|
  a := m1.Slice(0, rows/2, 0, rows/2)
  b := m1.Slice(0, rows/2, rows/2, rows)
  c := m1.Slice(rows/2, rows, 0, rows/2)
  d := m1.Slice(rows/2, rows, rows/2, rows)

  // Break up m2 into submatrices from quadrant blocks.
  //|e f|
  //|g h|
  e := m2.Slice(0, rows/2, 0, rows/2)
  f := m2.Slice(0, rows/2, rows/2, rows)
  g := m2.Slice(rows/2, rows, 0, rows/2)
  h := m2.Slice(rows/2, rows, rows/2, rows)

  // Calculate the 7 products: (elements matrix 1) * (elements m2)
  // Resultants are used for intermediate step calcs, add/subtract.
  var resultant1 mat.Dense
  var resultant2 mat.Dense

  // p1 = a * (f-h)
  resultant1.Sub(f, h)
  p1 := strassensAlgorithm(d.(*mat.Dense), &resultant1)

  // p2 = (a+b) * h
  resultant1.Add(a, b)
  p2 := strassensAlgorithm(&resultant1, h.(*mat.Dense))

  //p3 = (c+d) * e
  resultant1.Add(c, d)
  p3 := strassensAlgorithm(&resultant1, e.(*mat.Dense())
```

(con't on next sheet)

GO IMPLEMENTATION OF STRASSEN'S ALGORITHM (cont)

```
//p4 = d * (g-e)
resultant1.Sub(g, e)
p4 := strassensAlgorithm(d.(*mat.Dense), &resultant1)

//p5 = (a+d) * (e+h)
resultant1.Add(a, d)
resultant2.Add(e, h)
p5 := strassensAlgorithm(&resultant1, &resultant2)

//p6 = (b-d) * (g+h)
resultant1.Sub(b, d)
resultant2.Add(g, h)
p6 := strassensAlgorithm(&resultant1, &resultant2)

//p7 = (a-c) * (e+f)
resultant1.Sub(a, c)
resultant2.Add(e, f)
p7 := strassensAlgorithm(&resultant1, &resultant2)

// Product matrix quadrants
//| i j |
//| k l |
var i, j, k, l mat.Dense

// i = p5 + p4 - p2 + p6
resultant1.Add(p5, p4)
resultant2.Sub(&resultant1, p2)
i.Add(&resultant2, p6)

// j = p1 + p2
j.Add(p1, p2)

// k = p3 + p4
k.Add(p3, p4)

// l = p1 + p5 - p3 - p7
resultant1.Add(p1, p5)
resultant2.Sub(&resultant1, p3)
l.Sub(&resultant2, p7)
```

GO IMPLEMENTATION OF STRASSEN'S ALGORITHM (CON'T)

```
    // Combine and build the product matrix
    var topHalf, bottomHalf mat.Dense
    topHalf.Augment(&i, &j)
    bottomHalf.Augment(&k, &l)
    productMatrix.Stack(&topHalf, &bottomHalf)

    return &productMatrix
}
```