# 1. Counting Inversions

## 1.1. Brute Force Algorithm

Here is the algorithm I designed for counting inversions

```
ALGORITHM InversionsBruteForce(A[0..n − 1])
    // c keeps track of the inversions
    // A is an array of integers, length 0 to n–1
    c ← 0
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            if A[i] > A[j]:
                c ← c + 1
    return c
```

The common operation is comparison, it occurs once every inner loop, therefore I can set up a sum to count it.

$\Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}(1)$

$= \Sigma_{i=0}^{n-2}((n - 1) - (i + 1) + 1)$ (upper – lower + 1)

$= \Sigma_{i=0}^{n-2}(n - 1 - i - 1 + 1)$

$= \Sigma_{i=0}^{n-2}(n - i - 1)$

$= \Sigma_{i=0}^{n-2}(n) - \Sigma_{i=0}^{n-2}(i) - \Sigma_{i=0}^{n-2}(1)$

$= (n - 1)(n) - (\frac{(n-2)(n-1)}{2}) - (n - 1)$

$= n^2 - n - (\frac{n^2-3n+2}{2}) - n + 1$

$= n^2 - 2n - \frac{1}{2}n^2 + \frac{3}{2}n - 1 + 1$

$= \frac{1}{2}n^2 - \frac{1}{2}n$

$\Theta(n^2)$

There is no best/worst case, only the average case.

**Therefore the algorithm is in** $\Theta(n^2)$

## 1.2. Divide & Conquer

* Modified merge sort algorithm discussed in class

```
ALGORITHM InversionsDivdeConquer(D[0..n − 1])
    if n = 1 then
        return 0

    // c keeps track of the inversions
    // m is the middle index of the array
    c ← 0
```

```
m ← floor(n/2)
// partition D into 2 sub arrays, L & R
copy D[0..m − 1] to L[0..p]
copy D[m..n − 1] to R[m..q]

// Add inversions from partitions
c ← c + InversionsDivideConquer(L[0..p])
c ← c + InversionsDivideConquer(R[m..q])

// Add theoretical # of inversions
c ← c + (p × q)

while i < p and j < q do
        if L[i] < R[j] then
                D[k] ← L[i]
                k ← k + 1
                i ← i + 1
                // Remove false inversions
                c ← c − (q − j)
        else
                D[k] ← R[j]
                k ← k + 1
                j ← j + 1
// copy the rest of either R or L
if j < q then
        copy R[j..q] to D[k..n − 1]
else
        copy L[i..p] to D[k..n − 1]
return c
```

**Setup Recurrence:**

*** Basic operation is comparison, best case has $\frac{n}{2}$.**

$$C(n) = C(\tfrac{n}{2}) + C(\tfrac{n}{2}) + \tfrac{n}{2}$$

$$= 2C(\tfrac{n}{2}) + \tfrac{n}{2}$$

with base case $C(1) = 0$ or $n = 2^k$ with $k = 0$

$$C(2^k) = 2C(2^{k-1}) + 2^{k-1}$$

$$C(2^k) = 2[2C(2^{k-2}) + 2^{k-2}] + 2^{k-1} = 2^2C(2^{k-2}) + 2^{k-2} + 2^{k-1}$$

$$C(2^k) = 2^2[2C(2^{k-3}) + 2^{k-3}] + 2^{k-2} + 2^{k-1} = 2^3C(2^{k-3}) + 2^{k-3} + 2^{k-2} + 2^{k-1}$$

Clear pattern emerges:

$$C(2^k) = 2^iC(2^{k-i}) + \sum_{j=0}^{i}(2^{k-j})$$

$C(2^k) = 2^i C(2^{k-i}) + \sum_{j=0}^{i} (2^k 2^{-j})$

$C(2^k) = 2^k C(2^{k-k}) + 2^k \sum_{j=0}^{k} (2^{-j})$

$C(2^k) = 2^k C(2^0) + 2^k (2^{-1})(2^{-k-1} - 1)$

$C(2^k) = 2^k C(2^0) + 2^k (2^{-1})(2^{-k-1} - 1)$

$C(2^k) = 2^k [0] + 2^{-2} - 2^k$

$C(2^k) = 2^{-2} - 2^k$

$C(n) = \frac{1}{4} - \log_2 n$

Using master theorem:

$C(n) = 2C(\frac{n}{2}) + \frac{n}{2}$

$a = 2, b = 2, d = 1$

Since $a = b^2$ $(2 = 2^1)$ then $C(n) \in \Theta(n \log n)$

Although mine did not match the master theorem (calculation error), I believe the theorem and the algorithm to be in $\Theta(n \log n)$

## 1.3 Implementation

| | |
|---|---|
| Brute Force average: | 10,505,521ms |
| Divide & Conquer average: | 18,604ms |

# 2. Convex Hull

## 2.1. Convex Hull Brute force

This is the algorithm I was able to come up with.

```
ALGORITHM ConvexHullBruteforce(P[0..n − 1])
    // l counts the number of points to the left of the line
    // r counts the number of points to the right of the line
    for i ← 0 to n − 2 do
        for j ← i + 1 to n − 1 do
            l ← 0
            r ← 0
            L ← line from P[i] to P[j]

            for k ← 0 to n − 1 do
                if k = i or k = j then
                    continue
                else if P[k] left of L then
                    l ← l + 1
```

```
                    else if  P[k] right of L  then
                            r ← r + 1

                    if l > 0 and r > 0 then
                            break
            // xor because we want exclusivity
            if  l = 0  xor  r = 0  then
                    add P[i] to S
                    add P[j] to S

    return S
```

I found that I can set up a sum to evaluate the efficiency class:

$\Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}\Sigma_{k=0}^{n-1}(2)$

$= \Sigma_{i=0}^{n-2}\Sigma_{j=i+1}^{n-1}(2n)$

$= \Sigma_{i=0}^{n-2}(2n \cdot \Sigma_{j=i+1}^{n-1}(1))$

$= \Sigma_{i=0}^{n-2}(2n \cdot ((n - 1) - (i + 1) + 1))$

$= \Sigma_{i=0}^{n-2}(2n(n - 1 - i - 1 + 1))$

$= 2n[\Sigma_{i=0}^{n-2}(n - 1 - i)]$

$= 2n[\Sigma_{i=0}^{n-2}(n) - \Sigma_{i=0}^{n-2}(1) - \Sigma_{i=0}^{n-2}(i)]$

$= 2n[n(n - 1) - (n - 1) - \frac{(n-2)(n-1)}{2}]$

$= 2n[n^2 - n - n + 1 - \frac{1}{2}n^2 + \frac{3}{2}n - 1]$

$= 2n[\frac{1}{2}n^2 - \frac{1}{2}n]$

$= n^3 - n^2$

$n^3 - n^2 \in \Theta(n^3)$

**Therefore the algorithm is in** $\Theta(n^3)$

## 2.2. Convex Hull

- The convex hull algorithm I designed is split into 2 parts; the first is the *QuickHull* function that does a few things like find the minimum/maximum x points, and partition the points into the upper & lower hulls.
- The second part (*UpperHull*) finds the maximum ($m$) part away from the line ($l, r$), then partitions points into the line from $l - to - m$ and $m - to - r$.

* NOTE: Wasn't sure how to represent the basic operation (multiplication) in pseudo code since it happens in a more *abstract way so I've* left comments indicating where multiplication happens.

```
ALGORITHM UpperHull(P[0..n − 1], L̄)
```

```
// P is the dataset we are working with
// L is the line to compare the points to P_l P_r
S ← [] // empty hull
if n = 0 then
        return S
else if n = 1 then
        S[0] ← P[0]
        return S

// (A) Find point of maximum distance from line
d ← 0
for i ← 0 to n − 1 do
        t ← distance from P[i] to L̄  // *5 multiplications here
        if t > d then
                swap P[0] and P[i]
                d ← t

// Create 2 new lines that represent the left Point to the max,
//        and the right point on the line to the max
L̄_{l,m} ← line from P_l to P[0]  // *2 multiplications
L̄_{m,r} ← line from P[0] to P_r // *2 multiplications

l ← 1,   r ← n − 1,   i ← 0
// (B) find all points left outside of L̄_{m,r} and partition them to end of P
while i < r do
        s ← side of P[i] on L̄_{m,r}  // *2 multiplications here
        if s = outside then
                r ← r − 1
                swap P[i] and P[r]
        else
                i ← i + 1

// (C) find all points outside of L̄_{l,m} and partition them to start of P
i ← r − 1
while i ≥ l do
        s ← side of P[i] on L̄_{l,m} // *2 multiplications here
        if s = outside then
                swap P[i] and P[l]
                l ← l + 1
        else
                i ← i − 1
```

```
    // recursively call upper hull with partitions for left line and right line
    left ← UpperHull(P[1..l − 1], L̄_{l,m})
    right ← UpperHull(P[r..n − 1], L̄_{m,r})
    append left to S
    append right to S
    return S
```

```
ALGORITHM QuickHull(P[0..n − 1])
    // let P[0] represent the point with the minimum x value
    // let P[n − 1] represent the point with the maximum x value
    // find the greatest and smallest x values
    for i ← 1 to n − 2 do
        if P[i].x > P[0].x then
            swap P[0] and P[i]
        else if P[i].x < P[n − 1].x then
            swap P[n − 1] and P[i]

    // let S represent the convex hull
    add P[0] to S
    add P[n − 1] to S

    L̄_{l,m} ← line from P[0] to P[n − 1]
    L̄_{m,r} ← line from P[n − 1] to P[0]

    l ← 1
    r ← n − 1
    // organize points by splitting them into upper & lower hulls
    while l < r do
        s ← side of P[l] on L1 // *2 multiplications here
        if s = (below line) then
            r ← r − 1
            swap P[l] and P[r]
        else
            l ← l + 1

    upper ← UpperHull(P[1..l], L̄_{l,m})
    lower ← UpperHull(P[r..n − 1], L̄_{m,r})
    append upper to S
    append lower to S
    return S
```

*** Common operation is multiplication**

Multiplications in 1 call of UpperHull $5n + 2n + n = 8n$

*$n$ because in the best case, for part (C), it will only pass half way through the second while loop

$M(n) = M(\frac{n}{2}) + M(\frac{n}{2}) + 5n + 2n + n$

$= 2M(\frac{n}{2}) + 8n$

With $M(1) = 0$

**Setup recurrence:**

Base case will occur when $n = 1$, or $n = 2^k$ where $k = 0$

$M(2^k) = 2M(2^{k-1}) + 8(2^k)$

$M(2^k) = 2[2M(2^{k-1}) + 8(2^k)] + 8(2^k) = 2^2M(2^{k-2}) + 8(2^{k-1}) + 8(2^k)$

$M(2^k) = 2^2[2M(2^{k-3}) + 8(2^{k-2})] + 8(2^{k-1}) + 8(2^k) = 2^3M(2^{k-3}) + 8(2^{k-2}) + 8(2^{k-1}) + 8(2^k)$

Clear Pattern Emerges:

$M(2^k) = 2^iM(2^{k-i}) + \sum_{j=0}^{i-1} 8(2^{k-j})$

$M(2^k) = 2^iM(2^{k-i}) + \sum_{j=0}^{i-1} 8(2^k2^{-j})$

$M(2^k) = 2^kM(2^{k-k}) + \sum_{j=0}^{k-1} 8(2^k2^{-j})$ (sub. in $i = k$)

$M(2^k) = 2^kM(2^0) - (2^k)(2^{-1})(2^{-k} - 1)$

$M(2^k) = 2^kM(1) - 2^{-1}2^k$

$M(2^k) = 2^k[0] - 2^{-1}2^k$

$M(2^k) = 2^{-1}2^k$

$M(n) = \frac{1}{2}\log_2 n$

Using master theorem:

$M(n) = 2M(\frac{n}{2}) + 8n$

$a = 2, b = 2, d = 1$

Since $a = b^2$ $(2 = 2^1)$ then $M(n) \in \Theta(n \log n)$

Although mine did not match the master theorem (calculation error), I believe the theorem and the algorithm to be in $\Theta(n \log n)$

**Therefore the function is in the algorithm is in** $\Theta(n \log n)$

## 2.3 Comparison

With the bruteforce in $\Theta(n^3)$ and the divide and conquer in $\Theta(n \log n)$, we can analyze the execution times.

- Each program I executed 10 times and computed the average.

Brute force average: 31,648,750ms
Divide & Conquer average: 2,605ms

* Please note, due to personal reasons this week I was not able to complete the full analysis, and I was able to get an extension. I put a lot of effort into the quickhull algorithm though.