1.1 ALGORITHM Counting Inversions (A[0..n-1])

// This algorithm counts the number of inversions in an array using a brute force approach
// Input: An array A[0..n-1] with n distinct numbers
// Output: Number of inversions in array A

```
inversions ← 0
for i ← 0 to n-1 do
    for j ← i+1 to n-1 do
        if A[i] > A[j] then
            inversions ← inversions + 1
return inversions
```

Effeciency

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-1} (n-1-i)$$

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

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

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

The effeciency class is $\Theta(n^2)$

1.2

```
ALGORITHM Mergesort(A[0..n-1], (B[0..n-1]), left, right
// Sorts array A[0..n-1] by recursive mergesort
// Input: An array A[0..n-1] of orderable elements
// Output: total number of inversions in Array

  mid ← 0; count1 ← 0; count2 ← 0; count3 ← 0
  if right > left then
    mid ← (right + left)/2
    count1 ← count1 + Mergesort(A[0..n-1], B[0..n-1], left, mid)
    count2 ← count2 + Mergesort(A[0..n-1], B[0..n-1], mid +1, right)
    count3 ← count3 + Merge(A[0..n-1], B[0..n-1], left, mid +1, right)

  return count1 + count2 + count3

ALGORITHM Merge(A[0..n-1], B[0..n-1], left, mid, right)
// Merges two sorted arrays into one sorted array
// Input: Arrays B[0..p-1] and C[0..q-1] both sorted
// Output: Number of inversions in Array

  inversions ← 0; i ← left; j ← mid; k ← left
  while i ≤ mid - 1 and j ≤ right, do
    if A[i] ≤ A[j], then
      B[k] ← A[i]
      k ← k+1
      i ← i+1
    else
      B[k] ← A[j]
      k ← k+1
      j ← j+1
      inversions ← inversions + (mid - i)

  while i ≤ mid - 1, do
    B[k] ← A[i]
    k ← k+1
    i ← i+1

  while j ≤ right, do
    B[k] ← A[j]
    k ← k+1
    j ← j+1

  for i ← left to right, do
    A[i] ← B[i]

  return inversions
```

Analysis of the best case

$C(n) = 2C(\frac{n}{2}) + \frac{n}{2}, \quad C(1) = 0$

$n = 2^k$

$C(n) = 2C(\frac{n}{2}) + \frac{n}{2} = 2[2C(\frac{n}{4}) + \frac{n}{4}] + \frac{n}{2}$

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

$C(n) = 2^2 C(\frac{n}{2^2}) + 2(\frac{n}{2}) = 2^2[2C(\frac{n}{8}) + \frac{n}{8}] + 2(\frac{n}{2})$

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

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

Let $i = k$

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

$= 2^k C(1) + k(\frac{n}{2}) = 2^k \times 0 + k(\frac{n}{2})$

$= k(\frac{n}{2})$

$n = 2^k: \quad k = \log_2 n$

$C(n) = k(\frac{n}{2}) = \frac{1}{2} n \log_2 n \in \Theta(n \log n)$

$C_{best}(n) = 2C_{best}(\frac{n}{2}) + \frac{n}{2}, \quad C_{best}(1) = 0$

$f(n) = n/2 \in \Theta(n^1), \quad a = 2, \; b = 2, \; d = 1$

$b^d = 2^1 = 2 = a$

$C_{best}(n) \in \Theta(n^d \log n) = \Theta(n \log n)$

$T(n) = aT(n/b) + f(n)$
where $f(n) \in \Theta(n^d), \; d \geq 0$

Master Theorem
if $a < b^d$, $T(n) \in \Theta(n^d)$
if $a = b^d$, $T(n) \in \Theta(n^d \log n)$
if $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Comparisons

The brute force algorithm took about 3669ms where as the recursive divide-and-conquer took about 15ms. This is due to the effeciency of each algorithm. Brute force was $\Theta(n^2)$ and recursive divide-and-conquer was $\Theta(n \log n)$ which is much more effecient for counting inversions. Hence why $\Theta(n \log n)$ was significantly faster than $\Theta(n^2)$ for 60 000 integers

2.1

```
ALGORITHM BruteForceConvexHull(P, Pcount)
// This algorithm computes the convex hull from a set of points in a brute
   force approach
// Input: Set of points P and number of points
// Output: Set of points in the convex hull


  for i<-0 to Pcount, do
     for j<-0 to Pcount where P[j] != P[i]
       Compute line segment for P[i] and P[j]
       for k<-0 to Pcount where P[k] != P[i] ad P[k] != P[j]
          if P[k] is on one side of line segment, then
            convexP[k] <- P[k]
            k <- k+1
  return convexP
```

Effeciency

$\Theta(n)$ complexity tests for each $\Theta(n^2)$ edges and for each of $n(n-1)/2$ pairs of distinct points, we may need to find the sign of $dx+by-c$ for each of the other $n-2$ points

$\therefore \Theta(n^3)$

```
ALGORITHM ShortestPath(S1, S2, convexP, Pcount)
// This algorithm calculates the shortest path between a set of points
// Input: Point S1, Point S2, convex hull, and # of points in the hull
// Output: Shortest distance between points

  first; last;     // first is the index of S1 from the hull and last is S2
  Startpoint <- first
  index <- 0
  while Startpoint != last, do
     shortest[index].x <- convexP[Startpoint].x
     shortest[index].y <- convexP[Startpoint].y
     temp <- (Startpoint +1) % Pcount
     for i<-0 to Pcount, do
        if orientation(shortest, startpoint, i, temp) = 1, then     // clockwise
           temp <- i
     Startpoint <- temp
     index <- index +1
  shortest[index].x <- convexP[startpoint].x
  Shortest[index].y <- convexP[startpoint].y
  index <- index +1
```

```
first; last;     // first is the index of S1 from the hull and last is S2
Startpoint2 ← first
index2 ← 0
while Startpoint2 != last, do
    shortest[index2].x ← convexP[Startpoint2].x
    shortest[index2].y ← convexP[Startpoint2].y
    temp2 ← (Startpoint2+1) % Pcount
    for i ← 0 to Pcount, do
        if orientation(shortest, startpoint2, i, temp2) = 2, then     // Counter Clockwise
            temp2 ← i
    Startpoint2 ← temp2
    index2 ← index2+1
shortest[index2].x ← convexP[startpoint2].x
shortest[index2].y ← convexP[startpoint2].y
index2 ← index2+1

distance1 ← 0
for i ← 0 to index-1, do
    distance1 ← distance1 + sqrt((shortest[i].x - shortest[i+1].x)^2 + (shortest[i].y - shortest[i+1].y)^2)

distance2 ← 0
for i ← 0 to index2-1, do
    distance2 ← distance2 + sqrt((shortest[i].x - shortest[i+1].x)^2 + (shortest[i].y - shortest[i+1].y)^2)

if distance1 > distance2, then
    return distance2
else return distance1
```

```
ALGORITHM orientation(convexP, p, q, r)
// This algorithm finds the orientation of ordered triplet (p, q, r)
// Input: Convex hull points, p, q, r
// Output: 0 → p, q, r are collinear; 1 → clockwise, 2 → counter clockwise

    value ← (convexP[q].y - convexP[p].y) · (convexP[r].x - convexP[q].x) -
            (convexP[q].x - convexP[p].x) · (convexP[r].y - convexP[q].y)

    if value = 0, then
        return 0
    else if value > 0, then
        return 1  // clockwise
    else return 2  // counter clockwise
```

Effeciency

For every point we look at all other points to determine the next.
This means that it is $\theta(n \cdot h)$ where n is the number of input points and h is the number of hull points.
Worst case would be $\theta(n^2)$ as all points could be on the hull where $n=h$

2.2 ALGORITHM QuickHull(S)

// This algorithm finds the convex hull from the set S of n points
// Input: set S of points
// Output: Convex Hull

ConvexHull ← left (L) and right (R) points // guranteed in the hull

Line LR divides the remaining n-2 points into S1 and S2
S1 ← right side of line LR
S2 ← right side of line RL
result ← FindHull(S1, L, R)
result ← FindHull(S2, R, L)
return result

ALGORITHM FindHull(Sp, P, Q)
// This algorith finds points on the hull from Sp points that are on the right side of line P to Q
// Input: Set of points Sp, P, Q
// Output: Points on the hull

C ← farthest point from segment PQ
ConvexHull ← C between P and Q
Three points P, Q, and C partition the remaining points of Sp into S0, S1, S2 where S0 are points inside triangle PCQ

S1 ← right side points of line P to C
S2 ← right side point of line C to Q

FindHull(S1, P, C)
FindHull(S2, C, Q)

NOTE ✯

- used the same shortest path algorithm for quickhull

Effeciency

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$, $C_{best}(1) = 0$

$f(n) = n \in \Theta(n^1)$, $a = 2$, $b = 2$, $d = 1$
$b^d = 2^1 = 2 = a$

$C_{best}(n) \in \Theta(n^d \log n) = \Theta(n \log n)$

$T(n) = aT(n/b) + f(n)$
where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem
if $a < b^d$, $T(n) \in \Theta(n^d)$
if $a = b^d$, $T(n) \in \Theta(n^d \log n)$
if $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Comparisons

The brute force hull took about 25 seconds to compute whereas the quickhull algorithm took 14 ms which is because of the $\Theta(n^2)$ vs $\Theta(n \log n)$ time complexity difference