Brute Force inversions

This does a compare one with all for all elements and thus is a simplistic way of calculating inversion with no logic involved. It is expected to take a longer time compared to divide and conquer as it revisits many of the already computed elements in every iteration.

For data1.txt
Running Time: 5996 ms
Inversion Count: 623897407

Divide and Conquer Inversions

This algorithm leverages the technique of merge sort to split the array and sort the sub arrays to efficiently compute the comparisons. The division of problems to subproblems of half the size reduces the number of comparisons needed in each step. The merge creates a superset of the sorted halves while computing the inversions at the same time based on the inversion rule $A[i] > A[j]$ for $i < j$.

For data1.txt
Running Time: 12 ms
Inversion Count: 623897407

Comparison of the inversion algorithms: Theoretically, the inversion count in brute force is $O(n^2)$ and that of the divide and conquer is $O(n\log n)$. Comparing the running times the divide and conquer is 499.7 approximately 500 times faster than the brute force method. This is consistent with the theoretical analysis. The divide and conquer method is thus optimal in a sense that it computes correctly the 623897407 points in a fraction of time compared to brute force.

Brute Force Convex Hull

The convex hull brute force is again simplistic in a sense that it considers all possible pair of line segments with the set of points to determine if the remaining points for each line segment lie on the same side. Clearly this will take cubic time as there are n^2 line segments and n-2 possible points per line segments. This takes an exceptionally long time to compute results. Also, there are additional cases to check for collinearity which further hurts running time.

For data2.txt
Running Time: 23961 ms
Convex Hull Points: 24

Divide and Conquer Convex Hull: Quick Hull

The quick hull optimizes by drawing triangles from two extreme points that are part of the hull to another extreme point that lies in between the line segments between the two points. It then processes the remaining points by partitioning them in three splits caused by the triangle formed by the previous extremes and new extreme: between left most and newly found extreme lying outside the triangle, between rightmost lying outside triangle and discards the internal points in the triangle since they will never be a part of the convex hull set. Thus, it recursively computes the hull in each of the two partitions until there are no more points to process. The collinearity is naturally prevented in the removal phase by including the logic of any points inside and along edges of the triangle of the three points. This is a much smarter and more efficient way as the distance calculation steps are reduced in each recursive step alongside reducing the subproblem sizes. The result is a much more optimized convex hull construction strategy as seen in the results below:

For data2.txt
Running Time: 22ms
Convex Hull Points: 24

Comparison of Brute Force and Divide and Conquer Convex Hull Algorithms:
Theoretically, the brute force has O(n^3) complexity as opposed to the divide and conquer having O(nlogn) complexity. The running time is also drastically different which is consistent to the theory. The divide and conquer (22ms) is 1089.14 times faster than brute force (23961ms). This shows that the quickhull is optimal in the fact that it correctly calculates the 24 points and in a fraction of time compared to brute force.

```
inversionCountBruteForce(A)
    //A[0..n-1] array of distinct numbers of size n
    //inversion means, A[i] > A[j] for i < j
    count ← 0
    for each i ← 0 to n-2 do
        for each j ← i+1 to n-1 do
            if A[i] > A[j]
                count ← count + 1
    return count
```

Analysis

Basic Operation: Comparison

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

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

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

$$= (n-1)(n-2-0+1) - \frac{(n-2)(n-1)}{2}$$

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

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

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

```
convexHullBruteForce(A)
    // A[0..n-1] array of points of size n
    hull ← { }, collinear_points = { }
    swap the first point with leftmost extreme point
    for each i ← 0 to n-2 do
        for each j ← i+1 to n-1 do
            previousSign ← 0,        farthest_point A[j], rem_points = A - {A[i], A[j]
            a = A[j]y - A[i]y , b = A[i]x - A[j]x , c = A[i]x A[j]y - A[i]y A[j]x
            for each k ← 0 to n-2 do
                cursign = a * rem_points[k]x + b * rem_points[k]y - c
                if cur_sign = 0 do   //select farthest collinear point
                    if distance of A[i] to rem_points[k] > distance of A[i] to farthest_pt
                        collinear_points ∪ {farthest_point}
                        farthest_point = rem_points[k]
                    continue

                if prev_sign is 0 do:
                    prev_sign = cur_sign
                    continue

                if prev_sign ≠ cur_sign
                    break

            if k = n-2: hull ← hull ∪ ({A[i], farthest_point}
                                  ∩ collinear_points)

    return hull
```

Analysis

Basic operation. Sign detection which has 2 multiply and 3 add/sub. So, multiply most expensive operation happening every iteration

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

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

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

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

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

```
inversion Count Divide Conquer(A)
    // A[0..n-1] array of distinct numbers size n
    // inversion means A[i] > A[j] for i < j
    count ← 0

    count Inversion(0, n-1, count)

count Inversion(s, e, count)
    if s == e-1 return
    m = ⌊(s+e)/2⌋
    count Inversion(s, m, count)
    count Inversion(m, e, count)
    merge(s, m, e, count)

merge(s, m, e, count)                    // m..e is right sub array
    l ← s, r ← m, k ← 0                  // l..m is left sub array
    sorted Subarray [0... e-s]
    while (l < m and r < e) do
        if A[l] > A[r]
            sorted Subarray [k] = A[l]
            r ← r+1
            count ← count + m - l        all items to the right of current item A[l] is greater than A[r]
        else
            sorted Subarray [k] = A[r]
            l ← l+1
        k ← k+1

    if l < m do
        copy remaining m-l items from left sub array
    else
        copy remaining e-r item from right sub array

    copy all items from sortedSubarray to A[s..e]
```

(*) The algorithm is very similar to merge sort with additional logic added to count inversions

Best case analysis

Basic operation: comparison

Best case is when largest item of one subarray is smaller than smallest item of other subarray

→ $\frac{n}{2}$ comparisons will be needed

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

$n = 2^k$

$C(n) = 2[2C(n/4) + n/4] + n/2$

$= 2^2 C(n/4) + n/2 + n/2$

$= 2^2 C(n/4) + 2 \cdot n/2$

$C(n) = 2^2[2C(n/8) + n/8] + 2 \cdot n/2$

$= 2^3 C(n/8) + n/2 + 2 \cdot n/2$

$= 2^3 C(n/8) + 3 \cdot n/2$

$C(n) = 2^i C(n/2^i) + i \cdot n/2$

$i = k$

$C(n) = 2^k \overbrace{C(1)}^{0} + k \cdot n/2$

$= \frac{n \cdot \log_2 n}{2}$ $[\because n = 2^k]$

$\in \Theta(n \log n)$

<u>Using master Theorem</u>

$C(n) = \underbrace{2}_{a} C(n/\underbrace{2}_{b}) + \frac{n^{1 \to d}}{2}$

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

$\therefore a = b^d$, so $\Theta \in (n^d \log n)$

$= \Theta(n \log n)$

master thm says:

$a < b^d$, $T(n) \in \Theta(n^d)$

$a = b^d$, $T(n) \in \Theta(n^d \log n)$

$a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

The master theorem verifies analysis

```
Quick Hull (A):
  // A[0 .. n-1] set of points of size n
  // all points lie in positive X-Y axis
  Sort A in ascending order of x co-ordinates, break ties with ascending y coordinates
  hull ← {A[0], A[n-1]}
  leftMost = A[0], rightMost = A[n-1]
  remove leftMost and rightMost points from A

  // Divide A into two subarrays with points on top half and points of bottom half of
  // the line formed by A[0] and A[n-1]
  topPoints ← {}, bottomPoints ← {}
  for each i ← 0 to n-2 do
     if orientation of A[i] on top side of leftMost and rightMost points
         topPoints ← topPoints ∪ {A[i]}
     else if orientation of A[i] on bottom side of leftMost and rightMost points
         bottomPoints ← bottomPoints ∪ {A[i]}
     else : skip if collinear orientation

  partition (topPoints, leftMost, rightMost, hull)
  partition (bottomPoints, leftMost, rightMost, hull)

partition(points, left, right, hull)
  farthest_pt = NIL, farthest_dist = 0
  for i ← left to right do
     if distance of points[i] from line formed by points[left] and points[right]
          farthest_pt = points[i], farthest_dist = dist of points[i] to line(points[left], points[right])

  hull ← hull ∪ {farthest_pt}
  remove all points inside and on the edges of Δ(points[left], points[right], farthest_pt)
  left_to_farthest_points ← {}, farthest_to_right_points ← {}
  for i ← 0 to # of remaining points
     if points[i] is between line(points[left], farthest_point)
         left_to_farthest_points ← left_to_farthest_points ∪ {points[i]}
     else
         farthest_to_right_points ← farthest_to_right_points ∪ {points[i]}

  partition(left_to_farthest_points, left, farthest_pt, hull)
  partition(farthest_to_right_points, farthest_pt, right, hull)
```

Analysis Best Case

Basic operation: distance calculation of points within left and right in the partition phase which requires multiplication operation

In best case, there are equal number of points on top and bottom of the line between left and rightMost points

$C(n) = 2C(n/2) + n \quad , C(1) = 0 \qquad ; n = 2^k$

$= 2C(n/2) + 2 \cdot n/2$

$C(n) = 2[2C(n/4) + 2 \cdot n/4] + 2\, n/2$

$= 2^2 C(n/4) + 2^2\, n/4 + 2 \cdot n/2$

$= 2^2 C(n/4) + 2n$

$= 2^2 [2C(n/8) + 2 \cdot n/8] + 2n$

$= 2^3 C(n/8) + 3n$

$\therefore C(n) = 2^i C(n/2^i) + in$

$i = k \rightarrow C(n) = 2^k \underbrace{C(1)}_{0} + kn$

$= n \log_2 n$

$\in \Theta(n \log n)$

Using Master Theorem

$$\rightarrow \begin{cases} a < b^d & , T(n) \in \Theta(n^d) \\ a = b^d & , T(n) \in \Theta(n^d \log n) \\ a > b^d & , T(n) \in \Theta(n^{\log_b a}) \end{cases}$$

$C(n) = 2C(n/2) + n$

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

here, $a = 2$ and $b^d = 2^1 = 2$

$a = b^d \rightarrow T(n) \in \Theta(n^1 \log n)$

$\in \Theta(n \log n)$

So, Master theorem verifies correctness