1.1

```
ALGORITHM insertionSort (A[0…n - 1), n)
        //Sorts a given array by insertion sort
        //Input: An array A[0..n − 1] of n orderable elements
        //Output: Array A[0..n − 1] sorted in nondecreasing order
        for i ← 1 to n do
                sorted = A[i]
                unsorted = i - 1
                while unsorted  ≥ 0 and A[unsorted ] > sorted do
                        A[unsorted  + 1] ← A[unsorted]
                        counter ← counter + 1
                        unsorted ← unsorted - 1
                A[j + 1] ← sorted
        print “Count = “ counter
```

$$C_{worst}(n) = \sum_{i=1}^{n} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n} i = (n\,n)/2 \in O(n^2)$$

1.2

```
ALGORITHM mergeSort (data[], min, max)
        //sorts a given data array using modified mergeSort
        //input: an array data of max + 1 elements, min the index of the lower half of the array
        //max the index of the final element in the array
        //output: inversion the number of inversions required to sort

        inversion ← 0
        if min < max
                mid ← (min + max)/2
                //break up into lower half of array
                inversion ← inversion + mergeSort(data,min,mid)
                //break up into second half of array
                inversion ← inversion + mergeSort(data,mid+1,max)
                //sort elements
                inversion = inversion + merge(data,min,mid+1,max);

ALGORITHM merge (data[], min, max)
        //sorts a given data array using modified mergeSort
        //input: an array data of max + 1 elements, min the index of the lower half of the array
        //max the index of the final element in the array
        //output: inversion the number of inversions required to sort

        counter ← 0
```

```
low ← min
i ← min
index ← mid

while low < mid and index < max+1
        //sorts lower half
        if data[low] ≤ data[index]
                temp[i] ← data[low]
                low ← low + 1
        //sorts upper half
        else
                temp[i] ← data[index]
                index ← index + 1
                //adjust counter for amount sorted
                counter ← counter + mid - low
        i ← i + 1

//copy sorted values into temp
if low > mid - 1
        for k ← index to max + 1
                temp[i] ← data[k]
                i ← i + 1
else
        for k ← low to mid + 1
                temp[i] ← data[k]
                i ← i + 1

//save sorted values back into data
for k ← min to max + 1
        data[k] ← temp[k]

return counter
```

Let T(n) be the running time of Merge Sort on an input of size n. The Merge Sort algorithm can be divided into two main steps: the divide step, where the input is divided into smaller subproblems, and the conquer step, where the subproblems are combined and sorted. The divide step takes constant time, while the conquer step takes linear time proportional to the size of the input.

T (n) ∈
Θ(n^d) if a < b^d,
Θ(n^d log n) if a = b^d,
Θ(n^log_b a) if a > b^d

Thus, the recurrence relation for the running time of Merge Sort can be expressed as:

$T(n) = 2T(n/2) + O(n)$

By applying the Master Theorem, we can determine the time complexity of Merge Sort:

The first case of the Master Theorem states that if the recurrence relation can be expressed as $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$, and $f(n)$ is a function with a polynomial bound, then:

if $f(n) = O(n^{(\log_b a - \varepsilon)})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{(\log_b a)})$

if $f(n) = \Theta(n^{(\log_b a)})$, then $T(n) = \Theta(n^{(\log_b a)} * \log n)$

if $f(n) = \Omega(n^{(\log_b a + \varepsilon)})$ for some $\varepsilon > 0$, and if $a * f(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

In the case of Merge Sort, $a = 2$, $b = 2$, and $f(n) = O(n)$, so we have:

$\log_b a = \log_2 2 = 1$

$f(n) = O(n) = n^{(1 - \varepsilon)}$ for $\varepsilon = 1$

Therefore, we are in the second case of the Master Theorem, and the time complexity of Merge Sort is:

$T(n) = \Theta(n \log n)$

2.1

```
ALGORITHM orientation (p1,p2,nextPoint)
        //determines the position of point nextPoint relative to points p1 and p2
        //inputs: point p1, point p2, point nextPoint
        //returns 0 for point nextPoint is collinear to the points p1 and p2
        //returns 1 for point nextPoint is clockwise the points p1 and p2
        //returns 2 for point nextPoint is counter clockwise to the points p1 and p2
        val ← (p2.y - p1.y) * (nextPoint.x - p2.x) - (p2.x - p1.x) * (nextPoint.y - p2.y)
        if val == 0
                return 0
        if val > 0
                return 1
        else
                return 2

ALGORITHM convexHullBrute (points[], n)
        //finds the indexes of all the hull points using jarvis march
```

```
//input: points a set of Points, n the number of points
//output: extremePoints and array with all the indexes of the hull points

if n < 3
        return
leftMost ← 0
//finds the leftMost point in the data set
for i ← 1 to n
        if points[i].x < points[leftMost].x
                leftMost ← i
previousPoint ← leftMost
do
        //find next point
        nextPoint ← (previousPoint + 1) % n

        for i ← 0 to n
                //checks if point is counter clockwise relative to points
                if orientation(points[previousPoint], points[i], points[nextPoint]) == 2
                        nextPoint ← i
        //if array is full realloc to double the size
        if counter == size
                size ← size * 2
                extremePoints = (int*) realloc(extremePoints, sizeof(int)*size);
        extremePoints[counter] ← nextPoint
        counter ← counter + 1
        previousPoint ← nextPoint
//run until you reach the initial point again
while previousPoint != leftMost

return extremePoints
```

The while loop in the code runs once for every hull point found the for loop repeats for every point in the points data set this makes the time complexity of this section of the code O(nh) where h is the number of hull points. This has a worst case time complexity of O(n^2)

The first for loop in the code runs for n times and finds the leftmost point of the data set. However the main loop runs to the highest degree making the time complexity of this code O(n^2) at its worst case

2.2

```
ALGORITHM quickhull (points[], n, hullPoints)
        //finds the hull points of the given points S
        //input: points a set of Points, n the number of points
```

```
        //hullPoints a pointer to a data set which stores the hullPoints found
        //output: hullPoints the set of all hull points in the set of points S
        if n < 3
                return

        belowCounter ← 0
        aboveCounter ← 0

        leftMost.x ← ∞
        rightMost.x ← − ∞

        for i ← 0 to n
                //find leftMost and rightMost points
                if points[i].x < leftMost.x
                        leftMost ← points[i].x
                if points[i].x > rightMost.x
                        rightMost ← points[i].x
        //draw line between leftMost and rightMost point
        //all points above line go to above array
        //all points below line go to below array
        for i ← 0 to n
                //if below line
                if orientation(leftMost, rightMost, points[i]) == 1
                        below[belowCounter] ← points[i]
                        belowCounter ← belowCounter + 1
                //above including leftmost and rightmost
                else
                        above[aboveCounter] ← points[i]
                        aboveCounter ← aboveCounter + 1

        findHull(below,leftMost,rightMost,belowCounter,1,hullPoints)
        findHull(above,rightMost,leftMost,aboveCounter,0,hullPoints)

ALGORITHM findHull (points[],p1,p2,count,side,hullPoints[])
        //finds the hullPoints for the given points S
        //inputs: set of Points points, p1 the leftMost point, p2, the rightMost point
        //count the amount of points in S, hullPoints
        //output: hullPoints the set of all hull points in the set of points S

        //if there are no points in the set
        if count == 0
                return

        for i ← 0 to count
```

```
        //finds the total distance between p1,points[i] and p2
        dist ← distanceCompare(p1, points[i]) + distanceCompare(p2, points[i])
        if dist > maxD
                maxD ← dist
                extremePoint ← i
//if hullPoints is full realloc to double the size
if counter == size
        size ← size * 2;
        hullPoints = realloc(hullPoints, sizeof(struct Point)*size);
hullPoints[counter] ← points[extremePoint]
counter ← counter + 1

//side 1 for points above initial line between leftmost and rightmost points
//side 0 for points below initial line between leftmost and rightmost points
if side == 1
        //draws line between hull point and p1 and p2 making a triangle
        //all points above the line from p1 to hull point go into leftAbove
        //all points above the line from hull to p2 go into rightAbove
        //all points within the triangle are discarded
        for i ← 0 to count
                //line from p1 to hull point
                if(orientation(p1,points[extremePoint],points[i]) == 1)
                        leftAbove[leftCounter] ← points[i]
                        leftCounter ← leftCounter + 1
                if(orientation(points[extremePoint],p2,points[i]) == 1)
                        rightAbove[rightCounter] ← points[i]
                        rightCounter ← rightCounter + 1
if side == 0
        for i ← 0 to count
        //draws line between hull point and p1 and p2 making a triangle
        //all points above the line from p1 to hull point go into leftAbove
        //all points above the line from hull to p2 go into rightAbove
        //all points within the triangle are discarded
                if(orientation(p1,points[extremePoint],points[i]) == 2)
                        leftAbove[leftCounter] ← points[i]
                        leftCounter ← leftCounter + 1
                if(orientation(points[extremePoint],p2,points[i]) == 2)
                        rightAbove[rightCounter] ← points[i]
                        rightCounter ← rightCounter + 1

 //recursive calls to find hull points
findHull(leftAbove,p1,points[extremePoint],leftCounter, side, hullPoints);
findHull(rightAbove,points[extremePoint],p2,rightCounter, side, hullPoints);
```

```
ALGORITHM distanceCompare (P1, P2)
        //returns the distance from P1 to P2
        //inputs: P1 a point, P2 a point
        //output: a float with the distance between P1 and P2

        return sqrt((P1.x - P2.x) * (P1.x - P2.x) + (P1.y - P2.y) * (P1.y - P2.y))

ALGORITHM orientation (p1,p2,nextPoint)
        //determines the position of point nextPoint relative to points p1 and p2
        //inputs: point p1, point p2, point nextPoint
        //returns 0 for point nextPoint is collinear to the points p1 and p2
        //returns 1 for point nextPoint is clockwise the points p1 and p2
        //returns 2 for point nextPoint is counter clockwise to the points p1 and p2
        val ← (p2.y - p1.y) * (nextPoint.x - p2.x) - (p2.x - p1.x) * (nextPoint.y - p2.y)
        if val == 0
                return 0
        if val > 0
                return 1
        else
                return 2
```

The algorithm proceeds by recursively partitioning the point set into subsets lying to the left and right of a line segment, and then finding the extreme points of the set with respect to each line segment. The time complexity of Quickhull can be expressed using Master's Theorem.

T (n) ∈
Θ(n^d) if a < b^d,
Θ(n^d log n) if a = b^d,
Θ(n^log_b a) if a > b^d.

Let T(n) be the time complexity of Quickhull on a set of n points. The algorithm partitions the point set into two subsets, each of size at most (n-1), and then performs a linear-time scan over the point set to find the extreme points, so we can write:

T(n) = 2T((n-1)/2) + O(n)

Using Master's Theorem, we have a = 2, b = 2, and f(n) = n.
The critical exponent c is log2(2) = 1
Since f(n) = n is polynomially greater than n^c

T(n) = O(n log n)

Therefore, the time complexity of Quickhull on a set of n points is O(n log n).

2.3

The jarvis march brute force algorithm is consistently faster than the divide and conquer quickhull for this data set. Because jarvis march has a time complexity of O(nh) where h is the number of hull points in the data set it runs much faster than the divide and conquer implementation. This is mainly because of the relatively small amount of hull points (22) in the data set of 30000 points. Because of Jarvis marches time complexity, if the number of hull points were to significantly increase, so would the execution time. This makes the worst case time complexity of jarvis march O(n^2).

Quickhull uses a divide-and-conquer approach to compute the convex hull. It works by recursively dividing the set of points into two subsets and finding the convex hull of each subset. The program continues to do this until the convex hull is found and no points are left to split into subsets. The algorithm has a time complexity of O(n log n) in the average case and the worst case.

Overall both algorithms are viable with these algorithms having their own strengths and weaknesses. With Jarvis march being better in cases where there are less hull points and quickhull being better for cases where there are more hull points, your choice of algorithm depends on the input data.