

CS 456: Advanced Algorithms

Programming Assignment 02

Bhavana Kurapati (800665827)

bkurapa@siue.edu

Program Execution:

A make file compiles all the algorithms and to run the program we use:

```
./ConvexHull
```

Then it shows two options;

Please select input file [1] or random generation [2]:

If “1” is selected it asks “Please input file name”:

Please input file name: set10.dat

Then the output will be generated as set10.out.

If “2” is selected is asks “Please input points number”:

Please input points number: 50

Then the output will be generated as set50.out.

Motivation and Background:

The convex hull of a set of points is characterized as the smallest convex polygon, that encases the entirety of the points in the set. Convex implies, that the polygon has no corner that is bowed inwards. We can solve problems like Tracking disease Epidemic, Nuclear Leak Evacuation using convex hull. Also, not necessarily a best solution but we can use convex hull in image processing.

Pseudocode and Implementation:

Jarvis March: (Gift Wrapping Algorithm):

algorithm jarvis(S) is

 // S is the set of points

```

// P will be the set of points which form the convex hull. Final set size is i.
pointOnHull = leftmost point in S // which is guaranteed to be part of the CH(S)
i := 0
repeat
    P[i] := pointOnHull
    endpoint := S[0]    // initial endpoint for a candidate edge on the hull
    for j from 0 to |S| do
        // endpoint == pointOnHull is a rare case and can happen only when j == 1 and
        // a better endpoint has not yet been set for the loop
        if (endpoint == pointOnHull) or (S[j] is on left of line from P[i] to endpoint) then
            endpoint := S[j] // found greater left turn, update endpoint
    i := i + 1
    pointOnHull = endpoint
until endpoint = P[0]    // wrapped around to first hull point

```

Quick Hull:

Input = a set S of n points

Assume that there are at least 2 points in the input set S of points

QuickHull (S)

```

{
    // Find convex hull from the set S of n points
    Convex_Hull := {}
    Find left and right most points, say A & B
    Add A & B to convex hull
    Segment AB divides the remaining (n-2) points into 2 groups S1 and S2
        where S1 are points in S that are on the right side of the oriented line from A to B, and
        S2 are points in S that are on the right side of the oriented line from B to A
    FindHull (S1, A, B)
    FindHull (S2, B, A)
}

```

```

}
FindHull (Sk, P, Q)
{
    // Find points on convex hull from the set Sk of points
    // that are on the right side of the oriented line from P to Q
    If Sk has no point, then return.
    From the given set of points in Sk, find farthest point, say C, from segment PQ
    Add point C to convex hull at the location between P and Q .
    Three points P, Q and C partition the remaining points of Sk into 3 subsets: S0, S1, and S2
        where S0 are points inside triangle PCQ,
            S1 are points on the right side of the oriented line from P to C
            S2 are points on the right side of the oriented line from C to Q
    FindHull(S1, P, C)
    FindHull(S2, C, Q)
}
Output = Convex Hull

```

Merge Hull:

```

MergeHull(points P)
    Sort points P according to x
    Return hull(P)

Hull(points P)
    If |P| <= 3 then
        Compute CH by brute force,
        Return

    Partition P into two sets L and R ( with lower & higher coords x)
    Recursively compute  $H_L = \text{hull}(L)$ ,  $H_R = \text{hull}(R)$ 
     $H = \text{Merge hulls } (H_L, H_R) \text{ by computing}$ 

```

Upper_tangent (H_L , H_R) // find nearest points, H_L CCW , H_R CW

Lower_tangent (H_L , H_R) // (H_L CW, H_R CCW)

Discard points between these two tangents

Return H

For Upper Tangent:

Upper_tangent (H_L , H_R)

L <- line joining the rightmost point of a
and leftmost point of b.

while (L crosses any of the polygons)

{

while(L crosses b)

L <- L' : the point on b moves up.

while(L crosses a)

L <- L' : the point on a moves up.

}

For Lower Tangent:

Lower_tangent (H_L , H_R)

L <- line joining the rightmost point of a
and leftmost point of b.

while (L crosses any of the polygons)

{

while (L crosses b)

L <- L' : the point on b moves down.

while (L crosses a)

L <- L' : the point on a moves down.

}

Proof of correctness:

Jarvis March Algorithm:

Proof by induction:

Base Case: if there is only one point P, invariant is true, Point P is the final convex hull.

Induction Step:

At point p, for iteration i while searching for next point q, Orientation of (p, point at i, q) is counter clockwise.

Loop invariant:

The Orientation of the point P, and the point at 'i'th iteration and point that we are looking to fit in the convex hull are in counter-clockwise direction.

Initialization:

For a given graph, there is only one Convex Point P1(x, y). which is in counter clockwise direction.

Maintenance:

At iteration i, there is point P2, and we are looking for some point(p3) to add to convex hull, if P1, P2 and P3 are in counter clockwise direction then move to next iteration i+1(new point). If P1, P2, P3 aren't in clockwise direction, update P3 and P1.

- At iteration i, we have run through i-1 points and checked the orientations of the points and we have found convex hull points for those points.

Termination:

At the end, we come to the first point and we have an array which has the convex Hull points. All the remaining points lie inside the convex Hull.

Quick Hull Algorithm:

Base Case: if there is only one point P, invariant is true, Point P is the final convex hull.

Induction Step:

For 'n' number of points, consider k+1 point Where the Convex Hull C chooses farthest from the base line.

Loop Invariant:

Convex Hull C chooses and prints a point P1 from the whole set, which is farthest from base line and a line L is drawn parallel to base line passing through P1, then all the points fall inside the line passing through P1.

Initialization:

For only one point P0 right to the base line it is true that all points fall left to line passing through P0.

Maintenance:

Let's say base line is line segment AB and farthest point is P1, at iteration i, there is a point P2. ABP1 forms a triangle and not necessarily all the points fall inside the convex, we have to run through BP1 right part considering BP1 as base line and recursively call quick Hull.

By divide and conquer approach, After end of iteration i, we have a convex Hull C that follows our loop invariant.

Termination:

At the end of the iteration, we will be left with no points to add to Convex Hull C. Now C has the convex Hull points.

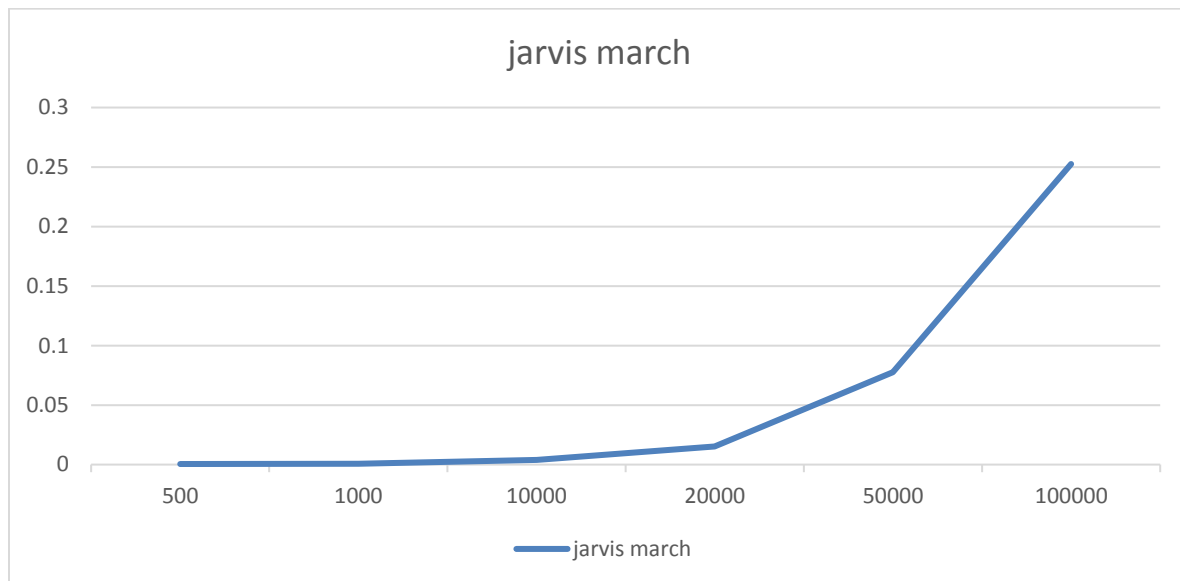
Testing Plan and Test Results:

Three algorithms run as follows:

Jarvis March Algorithm:

input size	jarvis march
500	0.000468526
1000	0.000796051
10000	0.00388393
20000	0.0152088
50000	0.0775782
100000	0.25255

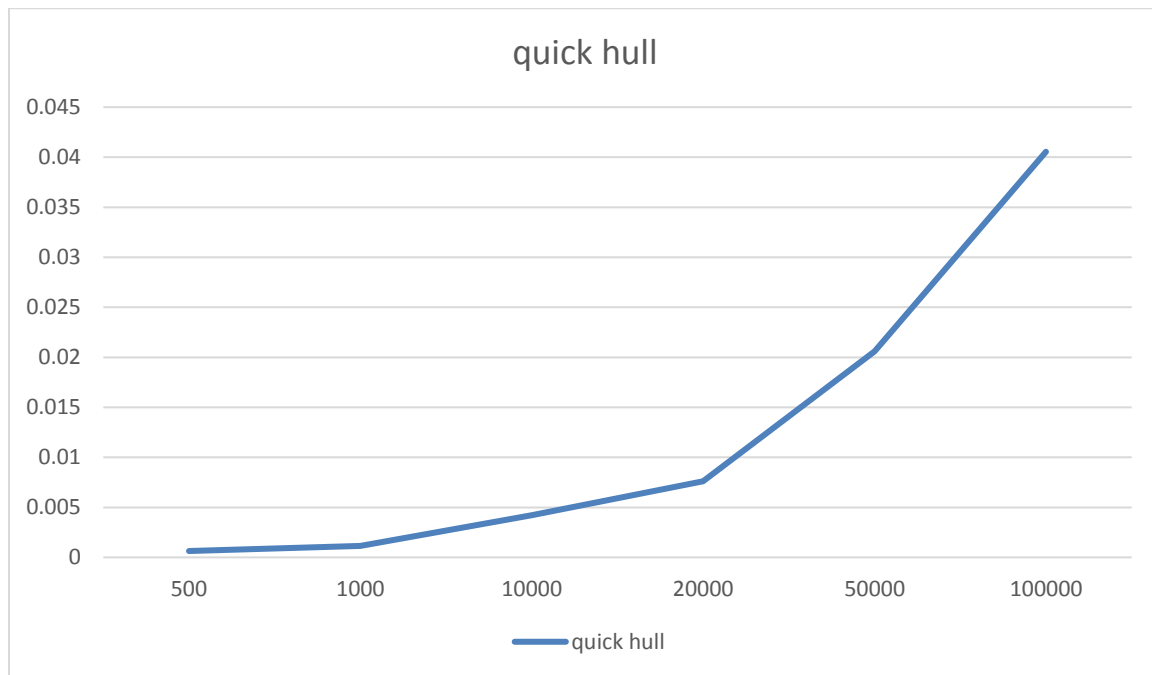
At $n_0=20000$, there is an asymptotic behavior in the graph,



Quick Hull Algorithm:

input size	quick hull
500	0.0006376
1000	0.0011518
10000	0.0042243
20000	0.0076144
50000	0.0205827
100000	0.0405418

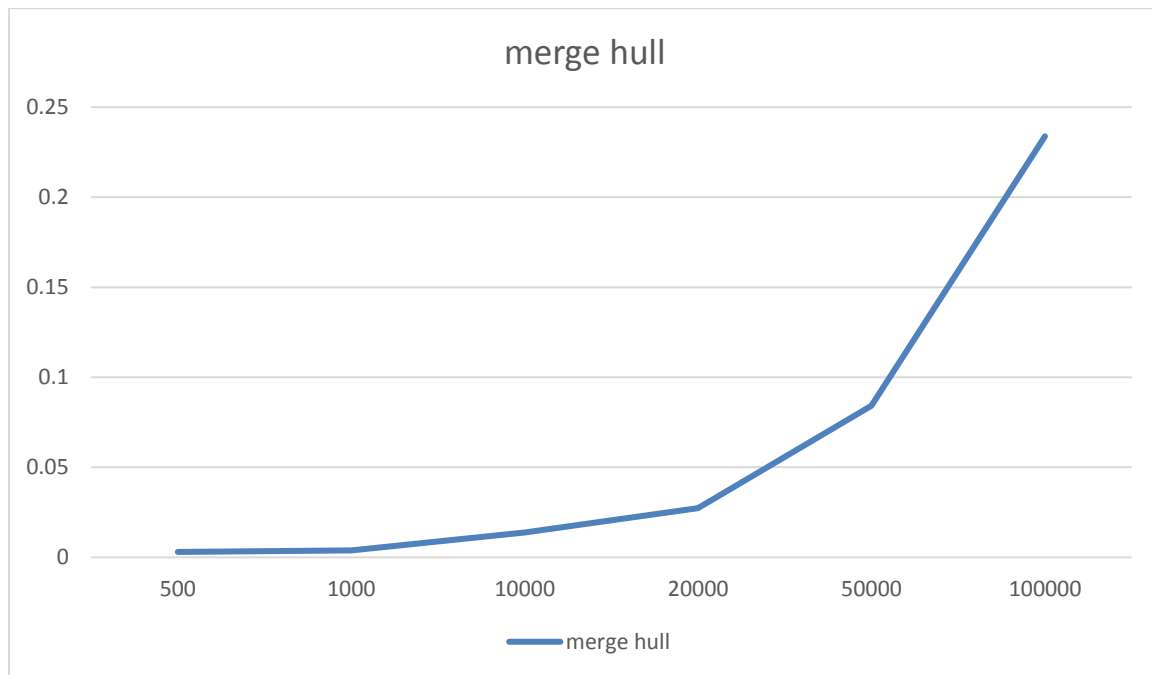
At $n_0=20000$, there is an asymptotic behavior in the graph,



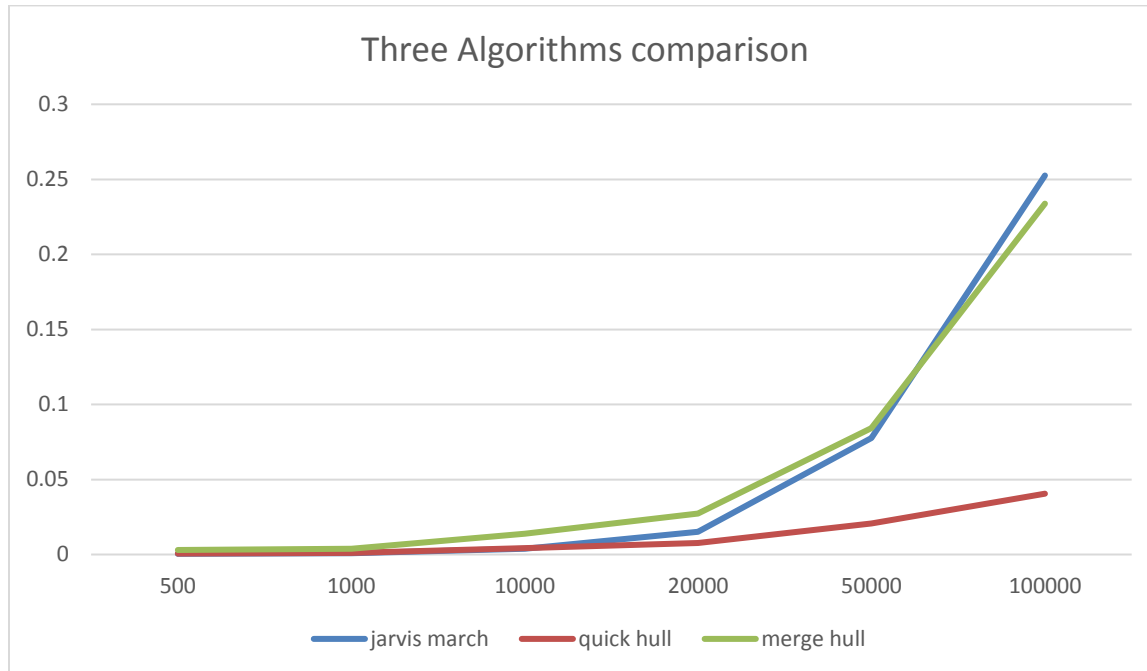
Merge Hull Algorithm:

input size	merge hull
500	0.00302777
1000	0.00389125
10000	0.0138372
20000	0.0273482
50000	0.0842624
100000	0.233798

At $n_0=20000$, there is an asymptotic behavior in the graph,



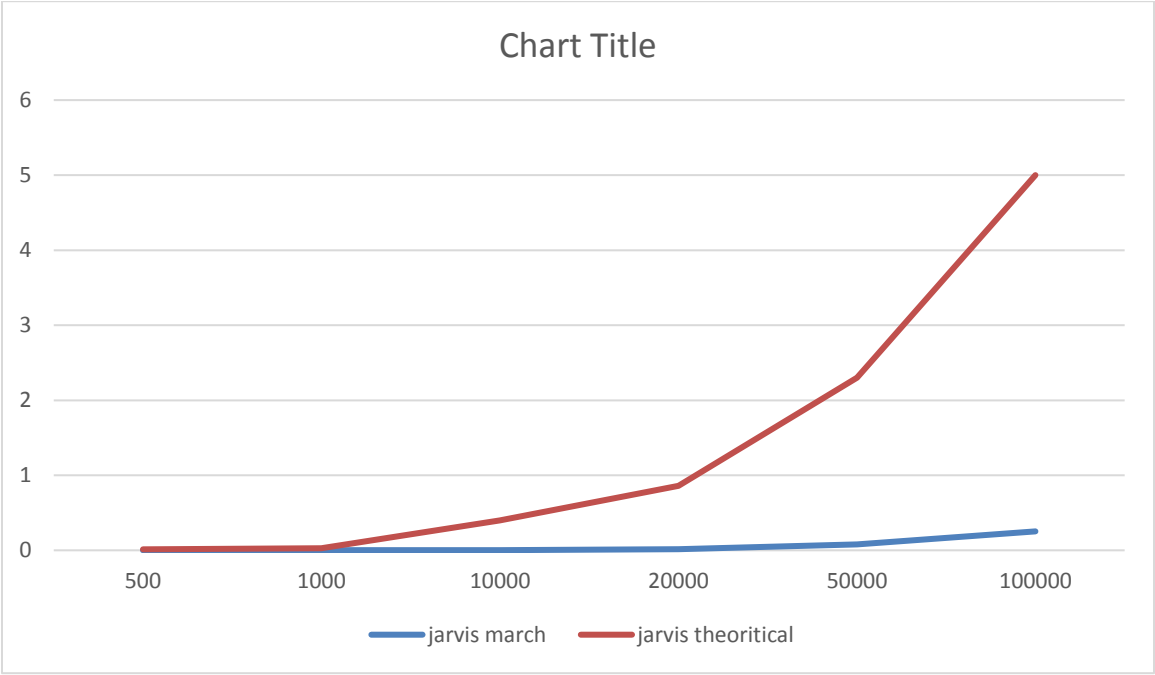
Three Algorithms comparison:



Theoretical Behaviour:

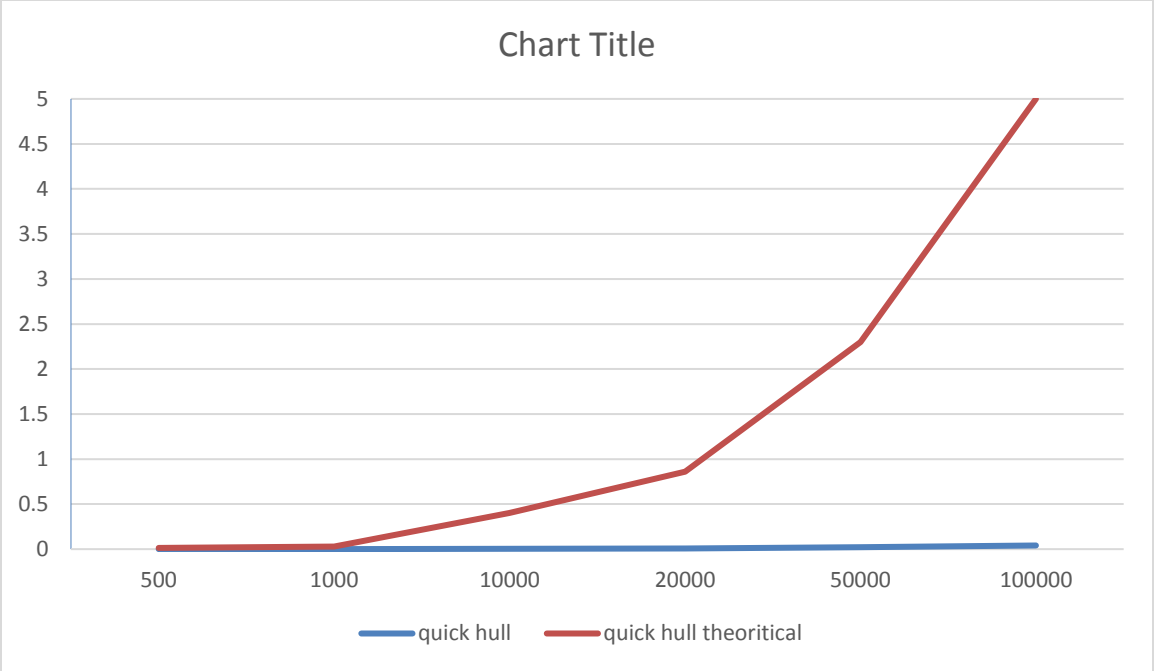
Jarvis March:

input size	jarvis march	jarvis theoretical
500	0.0004685	0.01349
1000	0.0007961	0.03
10000	0.0038839	0.4
20000	0.0152088	0.86
50000	0.0775782	2.3
100000	0.25255	5



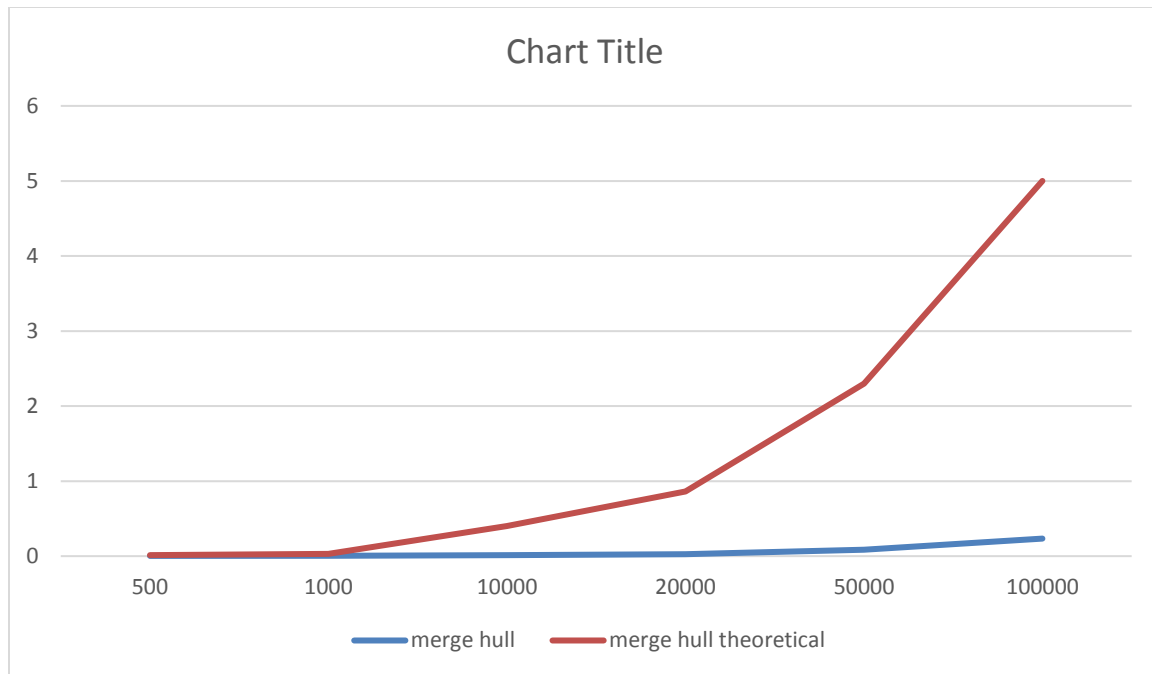
Quick Hull:

input size	quick hull	quick hull theoretical
500	0.0006376	0.01349
1000	0.0011518	0.03
10000	0.0042243	0.4
20000	0.0076144	0.86
50000	0.0205827	2.3
100000	0.0405418	5



Merge Hull:

input size	merge hull	merge hull theoretical
500	0.0030278	0.01349
1000	0.0038913	0.03
10000	0.0138372	0.4
20000	0.0273482	0.86
50000	0.0842624	2.3
100000	0.233798	5



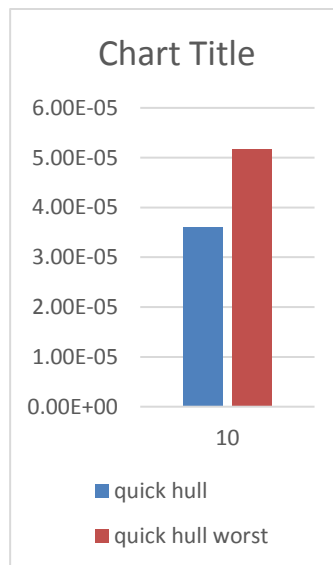
Worst Case complexity for Quick Hull Algorithm:

Quick hull runs faster because it eliminates a group of points at a same time. The Worst Case for Quick hull is points doesn't fall inside the triangle. One of the example for the worst case behavior can be points on the circle.

Example for 10 points:

5 0
 0 5
 0 -5
 -5 0
 3 4
 3 -4
 -3 4
 -3 -4
 4 3
 -4 -3

When we compare the 10 points convex hull, where the one file taken such as all the points taken are on the circle(i.e, worstcase) and another file taken such that some of the points are inside the circle, there is lot of difference in the execution time which implies that there is time complexity difference i.e, for the best and average case it is $O(n \log n)$ but for worst case the time complexity is $O(n^2)$. The below graph shows that:



Observations:

From the above graphs,

For the best and average cases Scenario,

Jarvis March runs at $O(nh)$ time $\sim O(n \log n)$

Quick hull runs at $O(n \log n)$ time

Merge hull runs at $O(n \log n)$ time

For the worst case Scenario,

Quick hull runs at $O(n^2)$ time

Jarvis March runs at $O(n^2)$ time

Merge hull runs at $O(n^2)$ time

Justifications for observations:

- In Jarvis March case, time complexity is $O(nh)$, It is an output sensitive algorithm. The total run time depends on size of input (n) and number of hull vertices. In the best case number of hull vertices run linearly or closer to logarithmic time so the run time is $O(n \log n)$. In worst case number of hull vertices are equal to total number of points ($n=h$) hence the time complexity is $O(n^2)$.
- In Quick hull Case, It follows Divide and Conquer approach so the best and average run time is at $O(n \log n)$. Worst case is $O(n^2)$ and this occurs when one half is heavily biased.
- In Quick hull Case, It follows Divide and Conquer approach so the best and average run time is at $O(n \log n)$. Worst case is $O(n^2)$.

Key Insights:

- From the above test plan, we can say that almost for any input the worst case of all three algorithms are equivalent to $O(n^2)$.
- For an average or best case, mostly algorithms are running at $O(n \log n)$.
- For jarvis march algorithm, the inner loop checks every point in the set S , and the outer loop repeats for each point on the hull. Hence the total run time is $O(nh)$. The run time depends on the size of the output, so Jarvis's march is an **output-sensitive algorithm**. In worst case, time complexity is $O(n^2)$. The worst case occurs when all the points are on the hull ($h = n$)

Performance Comparisons:

input size	jarvis march	quick hull	merge hull
500	0.000468526	0.000637557	0.00302777
1000	0.000796051	0.0011518	0.00389125
10000	0.00388393	0.00422425	0.0138372
20000	0.0152088	0.00761438	0.0273482
50000	0.0775782	0.0205827	0.0842624
100000	0.25255	0.0405418	0.233798

From the above table it is clear that in my case for different input sizes Quick Hull gives me the best results when compared to others. When comparing merge hull and Jarvis march, until the size is 50000, Jarvis march is performing better than merge hull but for the next sizes Jarvis is taking more time than Jarvis march. But I think I can consider merge hull is better than Jarvis march when taking into account of larger sizes

In my case, performance wise:

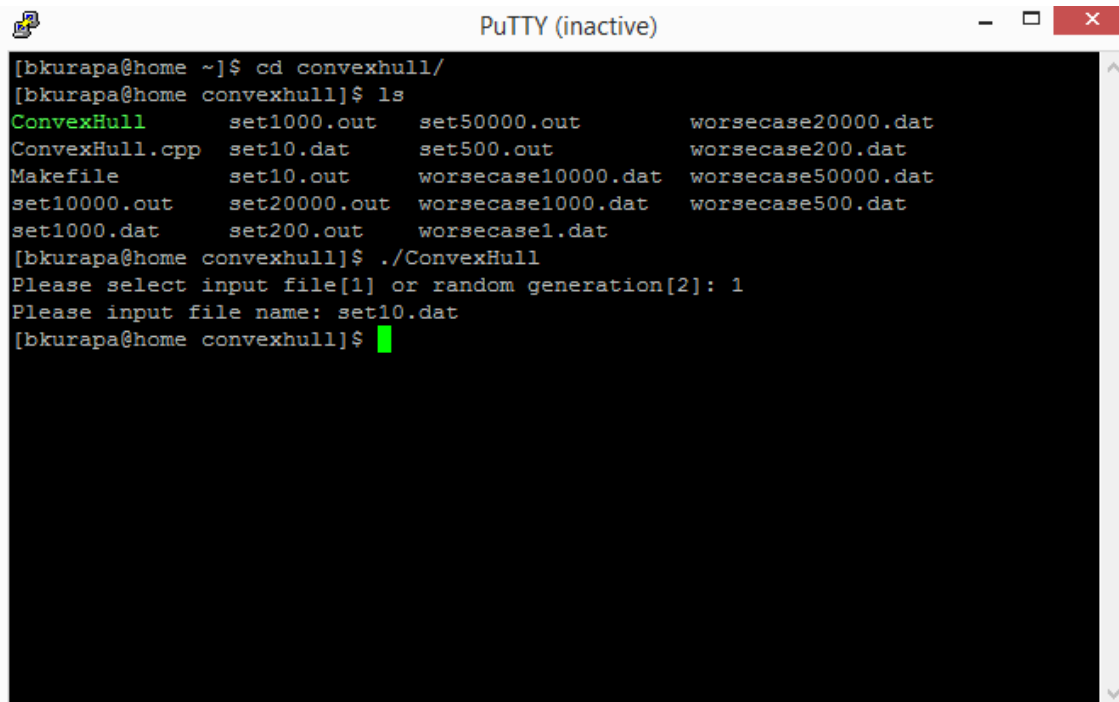
Quick Hull > Merge Hull > Jarvis March.

But in the Worst case Scenario of Quick Hull Algorithm, the file that was created has points in such a way that there are no points that fall inside the triangle and no points that can be eliminated. So from the above graphs we can say the run time is higher. (Quadratic time).

Conclusion:

In this Project, we have discussed and implemented Convex Hull Using Jarvis March Algorithm, Merge Hull Algorithm and Quick Hull Algorithm. We have compared the results and found the asymptotic behavior of the algorithm. We have discussed about the time complexities and reasons for those run times and evaluated their performance.

Proof of Work:



```
PuTTY (inactive)
[bkurapa@home ~]$ cd convexhull/
[bkurapa@home convexhull]$ ls
ConvexHull      set1000.out    set50000.out    worstcase20000.dat
ConvexHull.cpp  set10.dat      set500.out      worstcase200.dat
Makefile        set10.out      worstcase10000.dat worstcase50000.dat
set10000.out    set20000.out  worstcase1000.dat worstcase500.dat
set1000.dat     set200.out    worstcase1.dat
[bkurapa@home convexhull]$ ./ConvexHull
Please select input file[1] or random generation[2]: 1
Please input file name: set10.dat
[bkurapa@home convexhull]$
```



```

/home/bkurapa/convexhull/set10.out - bkurapa@home.cs.siue.edu - Editor - WinSCP
Jarvis March: 1.1795e-05 sec
QuickHull: 3.5989e-05 sec
MergeHull: 4.7259e-05 sec
10 -2
-3 -7
8 -5
9 5
-3 1
-7 -3
10 4

```

```

bkurapa@vm-02:~/convexhull
[bkurapa@home convexhull]$ ./ConvexHull
Please select input file[1] or random generation[2]: 2
Please input points number: 75
[bkurapa@home convexhull]$

```

```

/home/bkurapa/convexhull/set75.out - bkurapa@home.cs.siue.edu - Editor - WinSCP
[Icons] [Encoding ▾] [Color ▾] [?]
Jarvis March: 5.2238e-05 sec
QuickHull: 0.00013295 sec
MergeHull: 0.000420206 sec
977 373
830 23
258 52
107 78
880 32
466 979
806 990
888 989
29 551
44 963
27 348
989 798

```