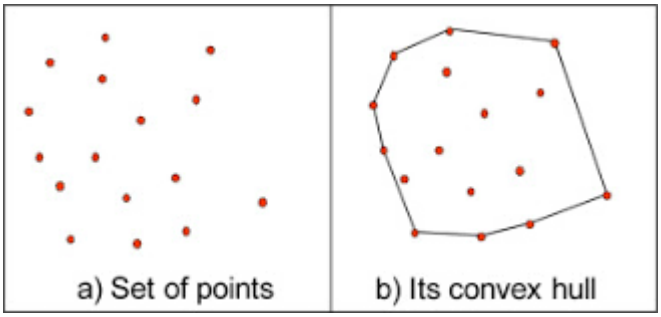
CONVEX HULL

Manoj kumar Nallamala

The convex hull of a set Q of points is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior. We denote the convex hull of Q by CH(Q). Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails. Figure below shows a set of points and its convex hull.



We have developed three algorithms that compute the convex hull of a set of n points.

1. Divide and Conquer
2. Graham's scan
3. Jarvis March

We have attached the 3 cpp files titled with the respective algorithm name. It can be run on cpp compilers.

Input of the problem :
Given n points (X_i, Y_i) on a two dimensional plane

Problem Statement :
The convex hull of the set of points Q is the convex polygon P that encompasses all of the points given. The goal of finding the smallest polygon P such that all the points of set Q are either on the boundary of P or inside P is known as the convex hull problem.

Output:
Return the list of points representing the Convex hull.

Programming Language Used : C++

Software used:
Clion IDE from JetBrains

Brute Force :

- The brute force method for determining a convex hull is to construct a line connecting two points and then verify whether all points are on the same side or not. There are such n(n – 1) /2 lines with n points, and each line is compared with the remaining n – 2 points to see if they fall on the same side.
- As a result, the brute force technique takes $O(n^3)$ time.
- Here each point while comparing on the same side of line takes constant time as it can be done by comparing the coordinates of a given point with the points forming the segment we are considering the final hull.

Divide and Conquer

1. Before calling the method to compute the convex hull, once and for all, we sort the points by x-coordinate. This step takes O(nlogn) time.
2. Divide Step: Find the point with median x-coordinate. Since the input points are already sorted by x-coordinate, this step should take constant time. Depending upon your implementation, sometimes it may take up to O(n) time.
3. Conquer Step: Call the procedure recursively on both the halves.
4. Merge Step: Merge the two convex hulls computed by two recursive calls in the conquer step. The merge procedure should take O(n) time.
5. For the Base case during the divide step of the hull we have applied brute force application of $O(n^3)$ approach for hull having less than 5 points which we can safely assume is comparable to a constant.

Algorithm to merge two hulls:

1. Find Upper Tangent of both hulls
2. Find Lower Tangent of both hulls
3. Merge hulls using two tangents by restricting points based on the tangent using a specific way of traversing

Algorithm for upper tangent:

```
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.
}
```

Algorithm for lower tangent:

```
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.
}
```

Some specific implementations used in this algorithm:

calculate_quadrant :

Arguments : Point *i.e x,y coordinates*
Task performed **:** Calculating the quadrant of a particular point.
Return value : Quadrant number *i.e 1,2,3 or 4.*

find_tangent :

Arguments : Three points out of which two are the one's which are supposed to form tangent
Task performed **:** If the line is touching the polygon. It gives the orientation of the line w.r.t to the hull we are verifying.
Return value : 0 or 1 based on the line pass or don't pass through the hull

compare :

Arguments : Two points *i.e x,y coordinates for each*
Task performed **:** This function will be passed as an argument to inbuilt sort which will return boolean. We use mid_point a global variable which is already calculated while traversing through the points.
Return value : True if the second point is counter clockwise & false vise versa

merging_hull :

Arguments : Two polygons which are two lists each consisting of points representing the two hulls that need to be merged.
Task performed **:** We will find the upper and lower tangents based on find_tangent call. Merging two polygons to get a convex hull.For forming the final hull after calculating tangents we will start by including the first point of upper tangent.Then we will traverse along the hull and add all points to result till we encounter first point of lower tangent. Then we will include two points of lower tangent and proceed to add other points in the second hull anticlockwise till we encounter the second point of the upper tangent.
Return value : List of points representing the merged convex hull from the two hulls.

brute_force_for_convex_hull :

Arguments : List of points which have size less than 5

Task performed **:** Bruteforce algorithm to find the convex hull.We will consider two points from the set of points and check if the line formed by these two points will be an edge for the final hull. For Checking if the edge is an edge is part hull we traverse through all points and verify if all the points are on the same side of the line formed.If so we will add this edge to the hull finally we will sort the result while returning.

Return value : List of points representing the convex hull

Time Complexity: The merging of the left and the right convex hulls take O(n) time and as we are dividing the points into two equal parts, so the time complexity of the above algorithm is O(n * log n) which is for sorting the points .

Space Complexity: O(n) as we are using a list of sixes maximum of input size for storing the hull for the merge step.

Challenges Faced and Observations:

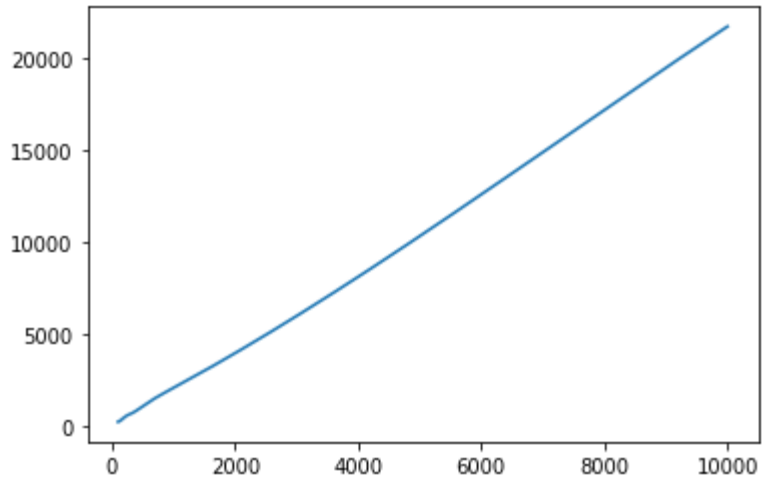
1. When input has been increased in the range of 10^6. We are making many calls to divide the input which is a lot of function calls for which we ran out of heap. We tried to increase heap size through some libraries in c++ but as the input grew exponential we hardly could squeeze space for executing the divide and conquer implementation for the convex hull.
2. Using divide and conquer for less than 5 points makes it really difficult to merge two hulls by finding the tangents as hulls for less number points, which mostly constitute all the points.
3. Finding the hull using brute force method for less points runs almost in a constant time so it helps us to eliminate the divide step to go really deep.
4. Runtime vs input size plot looks linear due to the scale of the graph on input size has been increased at a faster pace.But if we carefully observe by zooming in we could find a little bit of curvature.

Future Scope:

How about implementing the generation of two sub -hulls parallely using linear programming? So that both sub hulls must be constructed before starting the merge logic. Also we need to find a trade of when to parallelize finding sub hulls based on the input size otherwise we will end up in running too much space if we use threads and it can also affect runtime.

Runtime analysis with varying input sizes:

I/P Size	Run 1	Run 2	Run 3	Run 4	Run 5	Average Runtime
100	241	243	241	245	239	241.8
120	257	256	253	259	257	256.4
140	304	305	406	305	305	325
160	376	348	343	344	342	350.6
180	483	447	409	414	431	436.8
190	457	451	443	446	446	448.6
220	487	490	491	679	486	526.6
300	634	628	802	636	632	666.4
500	1025	1028	1024	1232	1138	1089.4
1000	2075	2087	2076	2081	2088	2081.4
5000	10370	10256	10255	10288	10443	10322.4
10000	21680	21652	21753	21663	21810	21711.6



Runtime v/s Input size plot

Graham's scan

1. Find the bottom point by comparing the y-coordinate. If there are two points with the same y value, then the point with smaller x coordinate value is considered.
2. Place the bottom-most point at first position
3. Sort n-1 points with respect to the first point.
4. A point p1 comes before p2 in sorted output if p2 has a larger polar angle (in counterclockwise direction) than p1 (or they don't make a left turn).
5. In above sorting, our criteria was to keep the farthest point at the end when more than one points have same angle
6. If the size of new array is less than 3 we cannot construct convex hull
7. Push the first three points into an empty stack

Some specific implementations and utilities used in this algorithm:

find_next_to_top :

Arguments : A pointer to a stack
Task performed **:** Extracts next to top value from stack
Return value : returns point which has the value

swap :

Arguments : A pointer to a stack
Task performed **:** Extracts next to top value from stack
Return value : returns point which has the value

square_distance :

Arguments : Two points x1,x2

Task performed **:** Calculates the euclidean distance between points

Return value : returns integer which is distance between the point

find_orientation :

Arguments : Three points p,q,r

Task performed **:** Calculates the orientation of the points with respect to middle

Return value : returns 0 if points are collinear and 1 if the points are clockwise and 2 if anticlockwise

sort_by_compare :

Arguments : Pointers to two points

Task performed **:** we will decide which points needs to be swapped with which point

Return value : returns -1 if points are anticlockwise and points which points are collinear and one come before other on comparing ordinates if the points are clockwise we return 2 if anticlockwise

draw_convex_hull :

Arguments : List of points and an integer which has value equal to number of points

Task performed **:** First we find the bottommost point.Sorts n-1 points with respect to the first point.A point p1 comes before p2 in sorted output if p2 has larger polar angle (in counterclockwise direction) than p1. If a modified array of points has less than 3 points, convex hull is not possible so we return nothing in this case.Process remaining n-3 points otherwise.Keep removing top while the angle formed by points next-to-top, top, and points[i] makes a non-left turn. When encountering a polar angle at midpoint while comparing three points leads to the formation of non left, we will backtrack, popping out the point from the already added point in the stack we will continue to pop till we encounter a left turn. It prints all the stack printing the final hull.

Return value : No return value

Time Complexity: Breaking the steps in the algorithm, at first we are finding the bottom-most point takes O(n) time and then sorting the points takes O(nLogn) time. Then it takes O(n) time as we push and pop a point at most one time into the stack. So the step to process points one by one takes O(n) time, assuming that the stack operations take O(1) time. Overall complexity is O(n) + O(nLogn) + O(n) + O(n) which is O(nLogn). This O(nLogn) will be a contribution from sorting which dominates the most.

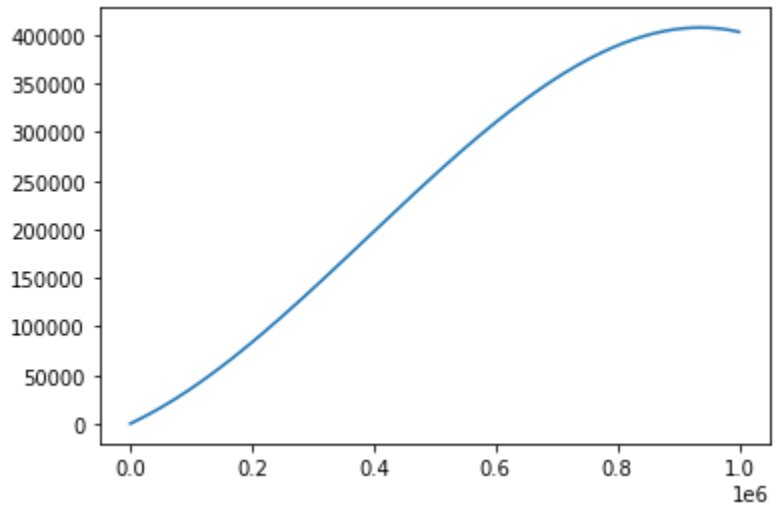
Space Complexity: O(n) for the stack we used to push all the points, which in the worst case we have all the points in space are in the hull which has all the points which contribute to the final convex hull.

Challenges Faced and Observations:

1. When we randomized points generation at one instance we have got a really large hull having many points exhausted the stack we have declared, but this turns out to be an extremely rare case.
2. The problem with the Graham algorithm is that it has no obvious extension to three dimensions. The reason is that the Graham scan depends on angular sorting, which has no direct counterpart in three dimensions.

I/P Size	Run 1	Run 2	Run 3	Run 4	Run 5	Avg
100	32	34	33	53	39	38.2
120	37	36	37	37	41	37.6
140	44	41	42	41	42	42
160	50	49	51	51	49	50
180	80	53	54	59	53	59.8
190	55	57	55	81	54	60.4
220	66	67	66	94	66	71.8
300	82	82	84	84	81	82.6
500	139	140	140	140	143	140.4
1000	275	278	277	280	280	278
5000	1468	1470	1464	1471	1782	1531
10000	3076	3086	3091	3165	3102	3104
100000	34789	34889	34851	34834	41540	36180.6

1000000	456769	384797	385256	402827	385375	403004.8



Jarvis March

1. Find the leftmost point ‘p’. Needed minimum 3 points
2. Add the current point ‘p’ to the result.
3. From the leftmost point start moving counterclockwise until we reach the same point. This step will take O(h) time where h is the no. of points in output.
4. Now we will search for a point ‘q’ such that the orientation(p,q,x) is counterclockwise for all ‘x’ points. If any point ‘i’ is more counterclockwise than ‘q’, update ‘q’.
5. The orientation(p,q,r) will return:
   - ➔ 0, if p,q, and r are collinear
   - ➔ 1, if clockwise
   - ➔ 2, if counterclockwise
6. Now ‘q’ will be the most counterclockwise with respect to p.
7. Now update ‘p’ with ‘q’ for the next iteration i.e; ‘q’ is added to the result.

Some specific implementation used in the algorithm:

draw_conevex_hull:

Arguments : List of points and an integer which has value equal to number of points

Task performed **:** If there are less than 3 points, the function will terminate. We will find the leftmost point and add it to the result. Now move counterclockwise until we reach the samepoint.This loop runs in O(h) where ‘h’ is the no.

of points in the result set. Search for a point ‘q’ such that orientaion is counterclockwise for all points. If any point ‘i’ is more counterclockwise than ‘q’ , update the value of ‘q’ with ‘i’. Now ‘q’ will be the most counterclockwise to ‘p’. For the next iteration, set ‘p’ as ‘q’, so that ‘q’ will be added to the result set.

Return value : No return value

find_orientation :

Arguments : Three points p,q,r

Task performed**:** Calculates the orientation of the points with respect to middle

Return value : returns 0 if points are collinear and 1 if the points are clockwise and 2 if anticlockwise

Challenges Faced and Observations:

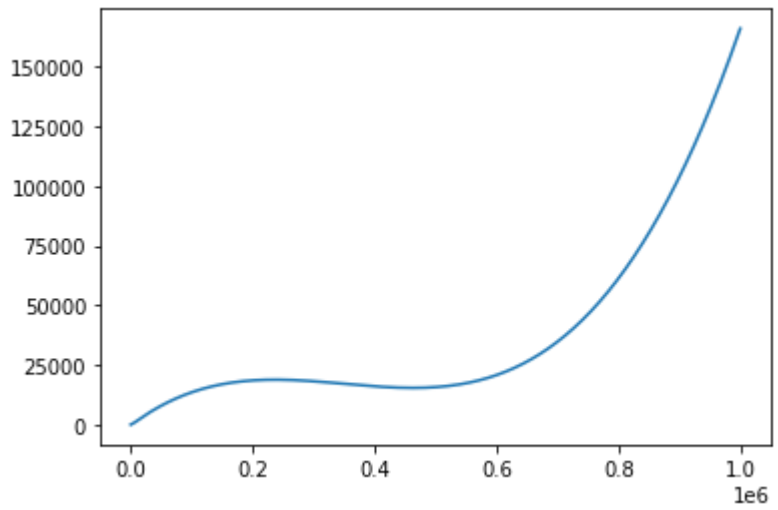
1. For increasingly larger inputs we ran out of stack especially many points are being pushed on to stack at given particular point

Time Complexity: O(m*n) where n is number of input points and m is number of output or hull points (m <= n). For every point on the hull we examine all the other points to determine the next point. In the worst case it is $O(n^2)$ we will be pushing all the points into the hull vector which is a very rare case to happen.

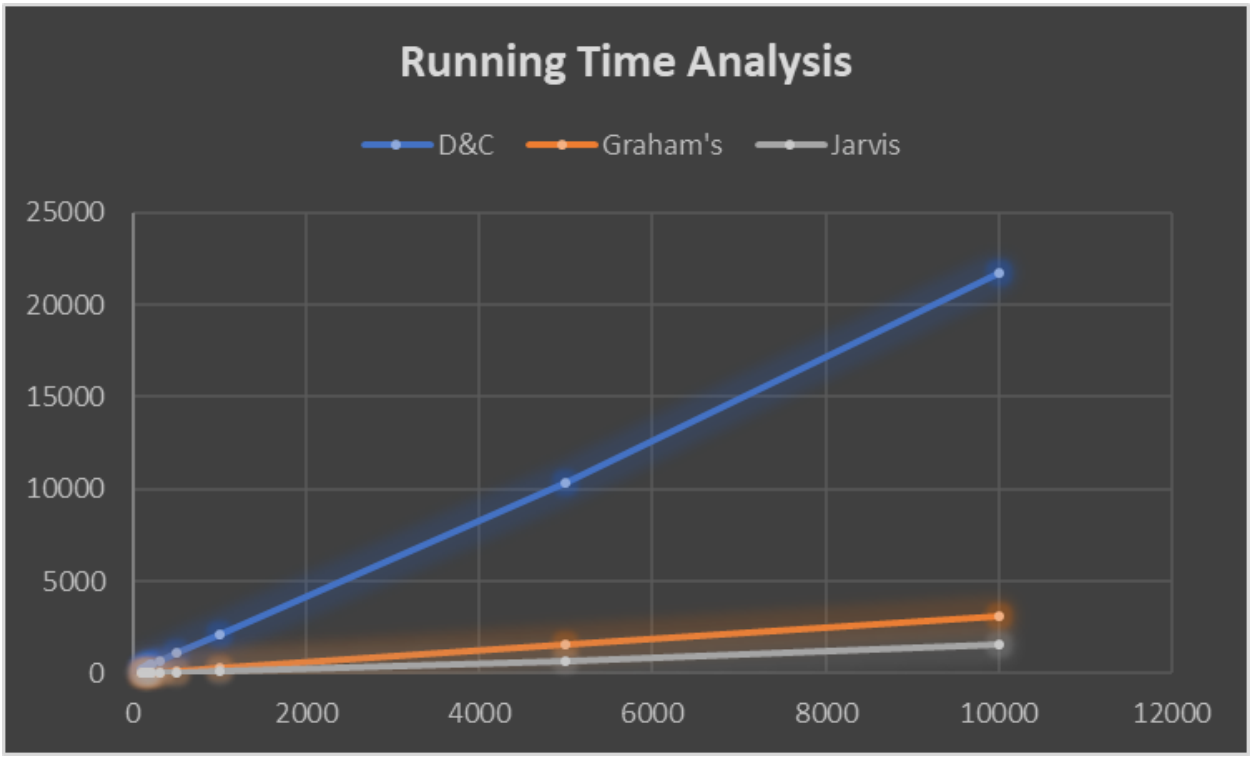
Space Complexity: O(n) for the vector we used to push all the points, which in the worst case we have all the points in space are in the hull which has all the points which contribute to the final convex hull.

I/P Size	**Run 1**	**Run 2**	**Run 3**	**Run 4**	**Run 5**	**Avg**
100	11	10	13	16	16	13.2
120	16	16	16	16	23	17.4
140	14	13	13	13	13	13.2
160	16	17	15	15	15	15.6

180	17	17	18	18	18	17.6
190	17	16	16	23	15	17.4
220	19	19	18	19	19	18.8
300	25	24	24	36	24	26.6
500	37	35	35	36	37	36
1000	88	130	89	88	88	96.6
5000	628	629	631	885	638	682.2
10000	1546	1564	1567	1538	1560	1555
100000	13720	13911	13429	13354	13377	13558.2
1000000	165504	162319	172105	164324	165500	165950.4



Comparing the Algorithms



- Naturally as we are making many recursive calls for Divide and Conquer when compared to other two there will be many stack operations happening as input goes large so there is a greater slope for the Divide and Conquer graph.
- For the other two algorithms it is almost comparable. But Jarvis march can have extremely bad running times particularly when the most of the points input constitutes the final hull, but as for now it is comparable with the graham's scan.

Work Distribution for the project:

1. Manoj Kumar Nallamala : Divide and Conquer, Graham's Scan, Generating input files
2. Raghavendra Koganti : Jarvis March, Preparing Report including plotting graphs for runtimes.