Ahsania Mission University of Science & Technology

Lab Report

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional

Experiment No: 07

Experiment Date: 07.05.25

Submitted By:

Mehrin Nusrat Chowdhury

Roll: 1012320005101026

1st Batch, 2nd Year, 1st Semester

Department of Computer Science and Engineering

Ahsania Mission University of Science & Technology

Submitted To:

Md. Fahim Faisal

Lecturer

Department of Computer Science and Engineering

Faculty of Engineering, Ahsania Mission University of Science & Technology

Submission Date: 21.05.25

Task 01: A C++ program that will implement Convex Hull using the Brute Force Algorithm.

Theory:

The Convex Hull problem aims to determine the smallest convex boundary enclosing a set of points in a 2D plane. The brute force method systematically examines all pairs of points and verifies whether the line segment formed is part of the convex hull. This method is computationally expensive (**$O(n^3)$**) but helps build foundational understanding.

Algorithm Steps:

1. Take a set of points in 2D space.
2. For every pair of points **(p_1, p_2)**, form a line.
3. Check whether all other points lie on the same side of the line.
4. If so, **(p_1, p_2)** is part of the convex hull.
5. Repeat the process for all pairs of points.
6. Store and display the set of convex hull edges.

Code:

```
#include<iostream>
#include<vector>
using namespace std;

struct Point
{
   int x , y;
};

int direction(Point a , Point b , Point c)
{
   return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}
void convexHull(vector<Point>& points)
{
   int n = points.size();
   cout << "Convex Hull Edges:\n";
   for(int i = 0 ; i < n ; i++)
   {
      for(int j = i + 1 ; j < n ; j++)
      {
         int pos = 0 , neg = 0;
         for(int k = 0 ; k < n ; k++)
         {
            if(k == i | k == j)
            {
```

```
                continue;
            }
            int d = direction(points[i] , points[j] , points[k]);
            if(d > 0)
            {
                pos++;
            }
            else
            {
                neg++;
            }
        }
        if(pos == 0 || neg == 0)
        {
            cout << "(" << points[i].x << "," << points[i].y << ")-(" << points[j].x << "," << points[j].y << ")\n";
        }
    }
  }
}
int main()
{
  vector<Point> points = {{0 , 3} , {2 , 2} , {1 , 1} , {2 , 1} , {3 , 0} , {0 ,0} , {3 , 3}};
  convexHull(points);

  return 0;
}
```

Output:

```
Convex Hull Edges:
(0,3)-(0,0)
(0,3)-(3,3)
(3,0)-(0,0)
(3,0)-(3,3)

Process returned 0 (0x0)   execution time : 0.081 s
Press any key to continue.
```

Conclusion:

Brute force methods provide foundational understanding but are inefficient for real-world scenarios. Algorithms like Graham's scan and Quickhull offer significant speed improvements, making them preferable for large-scale applications. Selecting the best method depends on the problem's complexity and available computational resources.

Task 02: A C++ program that will implement Convex Hull using Graham's Scan Algorithm.

Theory:

Graham's Scan is an efficient algorithm for computing the convex hull of a set of points in **O(n log n)** time. It begins by identifying the point with the lowest **y-coordinate**, which serves as the pivot. The remaining points are sorted by their **polar angle** relative to this pivot. Using a stack-based approach, the algorithm processes each point while ensuring left turns, discarding points that would lead to a right turn, ultimately forming the convex hull boundary.

Algorithm Steps:

1. Find the point with the lowest y-coordinate (break ties using x-coordinate). This is the starting point (pivot).
2. Sort all the other points based on the **polar angle** with respect to the pivot.
3. Traverse the sorted points and maintain a stack to determine **left turns** (using the orientation test).
4. If a **right turn** is detected, pop the top of the stack.
5. Repeat until all points are processed. The stack will contain the **convex hull vertices**.

Code:

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<stack>
using namespace std;

struct Point
{
   int x , y;
};

Point p0;

// A utility function to find the orientation of ordered triplet (p, q, r)
// Returns 0 if colinear, 1 if clockwise, 2 if counterclockwise
int orientation(Point p , Point q , Point r)
{
   int val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y);
   if(val == 0)
   {
```

```
        return 0;
    }
    return (val > 0) ? 1 : 2;
}

// Distance squared
int distSq(Point p1 , Point p2)
{
    return (p1.x -p2.x) * (p1.x -p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
}

// Compare function for sorting by polar angle
bool compare(Point p1 , Point p2)
{
    int o = orientation(p0 , p1 , p2);
    if(o == 0)
    {
        return distSq(p0 , p1) < distSq(p0 ,p2);
    }
    return (o == 2);
}

void grahamScan(vector<Point>& points)
{
    int n = points.size();

    // Find the bottom-most point
    int ymin = points[0].y , minIndex = 0;
    for(int i = 0 ; i < n ; i++)
    {
        if((points[i].y < ymin) || (points[i].y == ymin && points[i].x < points[minIndex].x))
        {
            ymin = points[i].y;
            minIndex = i;
        }
    }
    swap(points[0] , points[minIndex]);
    p0 = points[0];

    sort(points.begin() + 1 , points.end() , compare);

    stack<Point> hull;
    hull.push(points[0]);
    hull.push(points[1]);
    hull.push(points[2]);
```

```
    for(int i = 3 ; i < n ; i++)
    {
        while(hull.size() > 1)
        {
            Point top = hull.top();
            hull.pop();
            Point nextToTop = hull.top();
            if(orientation(nextToTop , top , points[i]) != 2)
            {
                continue;
            }
            else
            {
                hull.push(top);
                break;
            }
        }
        hull.push(points[i]);
    }

    cout << "Convex Hull Points (Graham's Scan);\n";
    while(!hull.empty())
    {
        Point p = hull.top();
        cout << "(" << p.x << "," << p.y << ")\n";
        hull.pop();
    }
}

int main()
{
    vector<Point> points = {{0 , 3} , {1 , 1} , {2 , 2} , {4 , 4} , {0 , 0} , {1 , 2} , {3 , 1} , {3 ,
3}};
    grahamScan(points);

    return 0;
}
```

Output:

```
Convex Hull Points (Graham's Scan);
(0,3)
(4,4)
(3,1)
(0,0)

Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```

Conclusion:

The comparative analysis highlights the importance of selecting the right convex hull algorithm based on dataset size, efficiency, and implementation complexity. Graham's Scan strikes a balance between speed and simplicity, making it a widely used method in computational geometry. Its structured approach ensures reliable performance for moderate to large datasets, making it a preferred choice in many practical applications.

Task 03: A C++ program that will implement Convex Hull using QuickHull Algorithm.

Theory:

QuickHull is an efficient convex hull algorithm that follows a **divide-and-conquer** approach. It starts by identifying the **leftmost and rightmost points**, forming an initial boundary segment. By recursively selecting the farthest points from the boundary, QuickHull efficiently constructs the convex hull while eliminating unnecessary points. This process ensures faster execution compared to brute-force methods, making it practical for large datasets. The expected time complexity is **O(n log n)**, though in worst cases, it can be **$O(n^2)$**.

Algorithm Steps:

1. Find the points with the **minimum and maximum x-coordinates**; these form the initial segment.
2. Divide the remaining points into **two groups**: left and right of this segment.
3. For each group, **find the point farthest** from the segment.
4. Form a **triangle** using this farthest point and the original segment.
5. **Recursively** find the farthest point from the triangle sides.
6. When **no more points** are found outside the triangle, the hull is complete.

Code:

```
#include<iostream>
#include<vector>
#include<cmath>
#include<set>
using namespace std;

struct Point
{
   int x , y;
};

// Function to find distance of point C from line AB
int distance(Point A , Point B , Point C)
{
   return abs((B.y - A.y) * (C.x - A.x) - (B.x - A.x) * (C.y - A.y));
}

// Determine the side of point P with respect to line AB
int findSide(Point A , Point B , Point P)
{
```

```
    int val = (P.y - A.y) * (B.x - A.x) - (B.y - A.y) * (P.x - A.x);
    if(val > 0)
    {
        return 1;
    }
    if(val < 0)
    {
        return -1;
    }
    return 0;
}

// Recursive QuickHull function
void quickHull(vector<Point>& points , Point A , Point B , int side , vector<Point>& hull)
{
    int index = -1;
    int max_dist = 0;

    for(int i = 0 ; i < points.size() ; i++)
    {
        int temp = distance(A , B , points[i]);
        if(findSide(A , B , points[i]) == side && temp > max_dist)
        {
            index = i;
            max_dist = temp;
        }
    }

    if(index == -1)
    {
        hull.push_back(A);
        hull.push_back(B);
        return;
    }

    quickHull(points , points[index] , A , -findSide(points[index] , A , B) , hull);
    quickHull(points , points[index] , B , -findSide(points[index] , B , A) , hull);
}

// Driver function
void findConvexHull(vector<Point>& points)
{
    if(points.size() < 3)
    {
        cout << "Convex hull not possible\\n";
        return;
```

```
    }

    int min_x = 0 , max_x = 0;
    for(int i = 1 ; i < points.size() ; i++)
    {
        if(points[i].x < points[min_x].x)
        {
            min_x = i;
        }
        if(points[i].x > points[max_x].x)
        {
            max_x = i;
        }
    }

    vector<Point> hull;
    quickHull(points , points[min_x] , points[max_x] , 1 , hull);
    quickHull(points , points[min_x] , points[max_x] , -1 , hull);

    set<pair<int , int>> uniqueHull;
    for(auto& p : hull)
    {
        uniqueHull.insert({p.x , p.y});
    }

    cout << "Convex Hull Points (QuickHull):\n";
    for(auto& p : uniqueHull)
    {
        cout << "(" << p.first << "," << p.second << ")\n";
    }
}

int main()
{
    vector<Point> points = {{0 , 3} , {1 , 1} , {2 , 2} , {4 , 4} , {0 , 0} , {1 , 2} , {3 , 1} , {3 ,
3}};
    findConvexHull(points);

    return 0;
}
```

Output:

```
Convex Hull Points (QuickHull):
(0,0)
(0,3)
(3,1)
(4,4)

Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

Conclusion:

The choice of the convex hull algorithm depends on factors like dataset size, point distribution, and performance needs. QuickHull offers a recursive divide-and-conquer approach, making it effective for scattered points, while Graham's Scan provides a structured method, ensuring stability in ordered sets. Selecting the optimal algorithm is essential for achieving efficient solutions in computational geometry applications.