

AHSANIA MISSION UNIVERSITY OF SCIENCE & TECHNOLOGY

Lab Report-7

Lab No: 07

Course Code: CSE 2202

Course Title: Computer Algorithm Sessional.

Submitted By:

Md Sakline Hossen
ID: 1012320005101027
1st Batch, 2nd Year, 2nd 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,
Ahsania Mission University of Science & Technology

Task 1: Convex Hull using Brute Force Algorithm

1. Objective

To implement the Convex Hull problem using the brute force approach in C++. Students will learn to understand the geometric concept of convex hulls and how to identify extreme points among a set of points.

2. Theory

The convex hull of a set of points is the smallest convex polygon that contains all the points. In the brute force approach, we check every pair of points and determine whether all other points lie on one side of the line formed by the pair. If they do, that edge is part of the convex hull.

3. Algorithm Steps

1. Take a set of points in 2D space. 2. For every pair of points (p1, p2), form a line. 3. Check whether all other points lie on the same side of the line. 4. If so, (p1, p2) 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.

Source 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 if (d < 0) 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:

```

"C:\Users\Polash01\Desktop\c" X + v
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 : 0.731 s
Press any key to continue.
|

```

Observation Table for Task 1: Brute Force

| Dataset | Input Points | Convex Hull Edges Identified | Time Complexity |
|---------|---|---|-----------------|
| 1 | (0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3) | (0,0)-(3,0), (0,0)-(0,3), (3,0)-(3,3), (0,3)-(3,3) | $O(n^3)$ |
| 2 | (0,0), (2,1), (1,2), (3,3), (4,0), (2,4), (0,3), (3,1) | (0,0)-(4,0), (4,0)-(3,3), (3,3)-(0,3), (0,3)-(0,0) | $O(n^3)$ |
| 3 | (2,2), (4,4), (6,2), (5,0), (3,0), (1,1), (4,2), (3,3) | (1,1)-(5,0), (1,1)-(4,4), (6,2)-(4,4), (6,2)-(5,0) | $O(n^3)$ |

Task 2: Convex Hull using Graham's Scan Algorithm

1. Objective

To implement the Convex Hull problem using Graham's Scan algorithm in C++. This algorithm is efficient and makes use of sorting and orientation concepts to find the convex hull.

2. Theory

Graham's Scan is an efficient algorithm to compute the convex hull of a set of 2D points. It works in $O(n \log n)$ time and is based on sorting the points by polar angle and then processing them to build the hull.

3. 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 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.

Source Code

```

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

```

```

struct Point {
    int x, y;

```

```
};
```

```
Point p0;
```

```
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;  
}
```

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

```
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();  
  
    int ymin = points[0].y, minIndex = 0;  
    for (int i = 1; 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);  
            }  
        }  
        hull.push(points[i]);  
    }  
}
```

```

        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:

```

"C:\Users\Polash01\Desktop\G... x + v
Convex Hull Points (Graham's Scan):
(0, 3)
(4, 4)
(3, 1)
(0, 0)

Process returned 0 (0x0) execution time : 0.612 s
Press any key to continue.
|

```

Observation Table for Task 2: Graham's Scan

| Dataset | Input Points | Convex Hull Points (unordered or recursive order) | Time Complexity |
|---------|---|---|---|
| 1 | (0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3) | (0,0), (3,0), (3,3), (0,3) | Avg: $O(n \log n)$, Worst: $O(n^2)$ |
| 2 | (0,0), (2,1), (1,2), (3,3), (4,0), (2,4), (0,3), (3,1) | (0,0), (4,0), (3,3), (2,4), (0,3) | Avg: $O(n \log n)$, Worst: $O(n^2)$ |
| 3 | (2,2), (4,4), (6,2), (5,0), (3,0), (1,1), (4,2), (3,3) | (1,1), (5,0), (6,2), (4,4) | Avg: $O(n \log n)$, Worst: $O(n^2)$ |

Task 3: Convex Hull using QuickHull Algorithm

1. Objective : To implement the Convex Hull problem using the QuickHull algorithm in C++. QuickHull is a divide-and-conquer algorithm useful for understanding recursive problem-solving in computational geometry.

2. Theory

QuickHull is a computational geometry algorithm to find the convex hull of a set of points. It operates similarly to QuickSort by recursively finding the outermost points and eliminating inner points that cannot be part of the convex hull. Time complexity is expected $O(n \log n)$, though it may degrade to $O(n^2)$ in the worst case.

3. 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.

Source Code

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

struct Point {
    int x, y;
};

int distance(Point A, Point B, Point C) {
    return abs((C.y - A.y) * B.x - (C.x - A.x) * B.y + C.x * A.y - C.y * A.x);
}

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;
}

void quickHull(vector<Point>& points, Point A, Point B, int side, vector<Point>& hull) {
    int index = -1, 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);
}

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;
    }

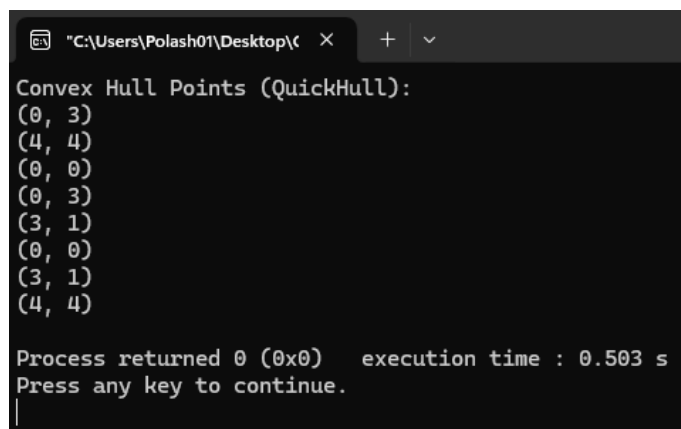
    vector<Point> hull;
    quickHull(points, points[min_x], points[max_x], 1, hull);
    quickHull(points, points[min_x], points[max_x], -1, hull);

    cout << "Convex Hull Points (QuickHull):\n";
    for (auto& p : hull)
        cout << "(" << p.x << ", " << p.y << ")\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:



```

C:\Users\Polash01\Desktop\<
Convex Hull Points (QuickHull):
(0, 3)
(4, 4)
(0, 0)
(0, 3)
(3, 1)
(0, 0)
(3, 1)
(4, 4)

Process returned 0 (0x0)   execution time : 0.503 s
Press any key to continue.

```

Observation Table for Task 3: QuickHull

| Dataset | Input Points | Convex Hull Points (in counterclockwise order) | Time Complexity |
|---------|---|--|-----------------|
| 1 | (0,3), (2,2), (1,1), (2,1), (3,0), (0,0), (3,3) | (0,0), (3,0), (3,3), (0,3) | $O(n \log n)$ |
| 2 | (0,0), (2,1), (1,2), (3,3), (4,0), (2,4), (0,3), (3,1) | (0,0), (4,0), (3,3), (2,4), (0,3) | $O(n \log n)$ |
| 3 | (2,2), (4,4), (6,2), (5,0), (3,0), (1,1), (4,2), (3,3) | (1,1), (5,0), (6,2), (4,4), (2,2) | $O(n \log n)$ |