Test cases specification for both algorithms:

Use Python scripts (Appendix 1) to generate 3 categories of tests as specified in the assignment: random points, points on a circle and points contained within 4 points making a square-shaped convex hull (simple shape).

For the square with side 2r, the 4 points picked will have coordinates:
(r, r), (-r, r), (-r, -r), (r, -r)

The input size is specified in the table below, along with a parameter value picked for practical purpose during test generations. The parameters are:

1. Random points - Range: Since it's not practical to create random points from (-infinity, +infinity), a range is picked, and random points is chosen from [0, range) in both x and y directions.
2. Points on a circle - Radius: Radius of the circle for the convex hull
3. Points inside the square – Half the side of the square (r)

Parameter values:

Input size	Random - Range	Circle - Radius	Square – Half side r
5,000	50	50	50
10,000	100	100	100
20,000	200	200	200
40,000	400	400	400
80,000	800	800	800

The counting of the basic operations is shown in the tables below. For Jarvis March, the input size is tested up to 40,000, since the algorithm complexity ($O(n^2)$) for 80,000 is beyond the limit of the int counting variable.

General assumptions:

1. Assume each point has a unique co-ordinate (approved by teaching staff), thus it can be represented uniquely by its index in the array storing all points read from the input.

Jarvis March:

1. Choice of data structure:

For Jarvis March, besides storing the convex hull points for the result, linear scanning from all points are used, so no additional data structure required. For storing the convex hull points, **doubly linked list** is used.

2. Algorithms:

Strictly follow the pseudocode given in assignment 1A, combined with the use of the given "orientation" function to determine the orientation using cross product.

Time Complexity expected: O(n h), with n being the total points, and h being the number of points on the convex hull ($h \leq n$), since for each point on the hull (h total), n orientation comparison is carried out.

Space Complexity expected: O(1) since only constant spaces are used (ignoring the results' storage)

3. Experimental evaluation:

Basic operations: A comparison between the angle (or orientation) of the points

Input size (n)	Random	Circle	Square
5,000	99,980	24,994,999	19,995
10,000	289,971	99,989,999	39.995
20,000	519,974	399,979,999	79,995
40,000	999,975	1,599,959,999	159,995

To show the growth rate better, we can normalize by dividing the counting and input size with the counting and input size in the first row:

Input size (n)	Random	Circle	Square
1	1	1	1
2	2.9	4	2
4	5.2	16	4
8	10	64	8

Conclusion:

1. **Worst case growth rate:** With the 3 test conditions, the random points on a circle, with all n points lying on the convex hull yields the worst performance, growing at n^2 compared to the input size growth, so worst case performance is at least $O(n^2)$.
2. The algorithm performance **depends on the proportion of points lying on the convex hull**. For the square case, where a fixed number of points lying on the convex hull, the performance grows linearly. For the random points, performance lies between linear and quadratic. Combined with 1, this confirms the complexity of O(n x h) expected.

Graham's scan:

1. Choice of data structure:

1. Storing the Convex hull points: Doubly linked list (since no random access required, and required O(1) insertion).
2. The items are required to be sorted based on their polar angle with the lowest point. For simplicity of implementation without worsening the algorithm's performance, an array of index of points are used for Mergesort.
3. Scanning process: Stack (provided in stack.c) is used.

2. Algorithms:

Strictly follows the provided pseudocode, with the choice of sorting algorithm explained below.

Sorting all points by its polar angle with respect to the lowest point:

1. **Merge sort** is chosen, **with time complexity: O(n log(n)) and space complexity: O(n)**
2. Perform on an **array of indices**, since each point can be uniquely represented by its index in the array of points stored inside the problem struct p.
3. **Comparator function:** Let's call the lowest point "pivot". Given 2 points c and d, c has a lower polar angle than d if the orientation(c, pivot, d) is CLOCKWISE. d has a lower angle than c if the orientation(c, pivot, d) is COUNTER-CLOCKWISE. Otherwise, if the orientation is COLLINEAR, then compare the distance squared between (c, pivot) and (d, pivot), and the closer point will come first.
4. **Distance squared and orientation** is used instead of the actual distance and angle to enhance performance and precision, since square root and inverse trigonometric function is replaced with simply multiplications, addition and subtraction.

Time Complexity (as described in the assignments):

O(n) for scanning, comparing orientation and stack manipulation (since each point is pushed/popped from the stack once and O(1) for push/pop from stack, O(1) for orientation comparison)

O(n log n) for sorting the points polar angles with merge sort

Thus, O(n log(n)) overall with n being the total number of points

Space Complexity: O(n) as Merge sort uses a temp array to merge arrays, and the stack used during scanning

3. Experimental evaluation:

Basic operations: Comparison between angles of points during the sort

Input size (n)	Random	Circle	Square
5,000	55,231	55,204	55,179
10,000	120,491	120,472	120,356
20,000	260,825	260,903	260,985
40,000	561,651	561,608	561,860
80,000	1,203,138	1,203,422	1,203,752

To show the growth rate better, we can normalize by dividing the counting and input size with the counting and input size in the first row:

Input size (n)	Random	Circle	Square	[n log(n)]/ [5000 log(5000)]
1	1	1	1	1
2	2.18	2.18	2.18	2.16
4	4.722	4.726	4.73	4.65
8	10.17	10.17	10.18	9.95
16	21.78	21.80	21.82	21.2

Conclusion:

1. The algorithm performance **depends entirely on the input size**, and is not affected by the 3 different distribution tested.
2. The algorithm performance is clearly between linear and quadratic. By roughly compare the growth rate with the estimate of the growth rate of n log (n) (by simply taking n log(n) divides by 5000 log(5000), which is the rough estimate of the complexity of the 1st input size), we can observe that the growth rate is close to the n log n growth rate, confirming the **n log n** complexity expected.

APPENDIX 1: Python script for points generation

```python
import random
import math

# Output all points to a file
def writeFile(points: list, filename: str):
    with open(filename, "w") as f:
        f.write(str(len(points)) + "\n")
        for point in points:
            f.write(f"{point[0]} {point[1]}\n")

# Create n random points from 0 -> r for both axes
# and write the points to testId file
def randomPoints(n: int, r: int, testId: int):
    points = []
    for i in range(n):
        points.append((random.random() * r, random.random() * r))
    writeFile(points, f"test_random_{testId}.txt")

# Create n random points in a circle with radius r
def randomPointsCircle(n: int, r: int, testId: int):
    points = []
    for i in range(n):
        x = random.random() * r
        y = math.sqrt(r * r - x * x)
        points.append((x, y))
    writeFile(points, f"test_circle_{testId}.txt")

# Create n random points in a square with side 2r
def randomPointsSquare(n: int, r: int, testId: int):
    points = []

    # Add 4 corners of the square
    points.append((-r, r))
    points.append((r, r))
    points.append((-r, -r))
    points.append((r, -r))

    for i in range(n - 4):
        x = random.random() * 2 * r - r
        y = random.random() * 2 * r - r
        points.append((x, y))
    writeFile(points, f"test_square_{testId}.txt")
```

```
def testGenerate():
    # Random points
    randomPoints(5000, 50, 1)
    randomPoints(10000, 100, 2)
    randomPoints(20000, 200, 3)
    randomPoints(40000, 400, 4)
    randomPoints(80000, 800, 5)

    # Random points on a circle
    randomPointsCircle(5000, 50, 1)
    randomPointsCircle(10000, 100, 2)
    randomPointsCircle(20000, 200, 3)
    randomPointsCircle(40000, 400, 4)
    randomPointsCircle(80000, 800, 5)

    # Random poinst in a square
    randomPointsSquare(5000, 50, 1)
    randomPointsSquare(10000, 100, 2)
    randomPointsSquare(20000, 200, 3)
    randomPointsSquare(40000, 400, 4)
    randomPointsSquare(80000, 800, 5)

testGenerate()
```