Convex Hull:

Time and Space Complexity overview sudo code for actual report go to the bottom.

Space Complexity: O(n log n)

Time Complexity: O(n log(n))

2T n/2 + O(3n)   A=2 B=2 D=1

Space Complexity

```
O(n) - def compute_hull(points: list[tuple[float, float]]) -> list[tuple[float, float]]:
O(n)     points = sorted(points, key=lambda x: (x[0], x[1]))   - n log(n)
         convex_hull = div_hull(points)
         return convex_hull

O(log n) - def div_hull(points):
         points
         if len(points) <= 2: - C
             return points   - C
         else:
O(1) ---     mid = len(points) // 2    - a = 2        2T n/2 O(n+n+n) => n log(n)
O(n)         left = points[:mid]
             right = points[mid:]
O(log n)     rightHull = div_hull(right)
             leftHull = div_hull(left)     > b = 2, a

             return combine_hull(leftHull, rightHull) -- O(n)

O(n) - def combine_hull(left, right):
         if (len(left) == 1) and (len(right) == 1):   > Base Case: O(1)
             return [left[0], right[0]]
         else:
             rightmost_left_index = max(range(len(left)), key=lambda i: left[i][0])
             leftmost_right_index = min(range(len(right)), key=lambda i: right[i][0])   > Find index points O(n)

             upper_tangent = find_upper_tangent(left, right, rightmost_left_index,
         leftmost_right_index)   - O(n)
             lower_tangent = find_lower_tangent(left, right, rightmost_left_index,
         leftmost_right_index)   - O(n)
```

$O(1)$

temp variables

```
upper_right = upper_tangent[1]
lower_right = lower_tangent[1]
upper_left = upper_tangent[0]
lower_left = lower_tangent[0]
```

$O(1)$: Set points for combination

While combining the space complexity is at worst $O(n)$ because every point is added

```
combined_hull = []
currentPoint = upper_left
combined_hull.append(left[currentPoint])
```

- Add all points from 0 -> left upper $O(1)$

```
currentPoint = upper_right
combined_hull.append(right[currentPoint])
while currentPoint != lower_right:
    currentPoint = (currentPoint + 1) % len(right)
    combined_hull.append(right[currentPoint])
```

Add all points from upper right to lower right — at worst $O(n)$

```
if upper_left != lower_left:
    currentPoint = (lower_left) % len(left)
    while currentPoint != upper_left:
        combined_hull.append(left[currentPoint])
        currentPoint = (currentPoint + 1) % len(left)
return combined_hull
```

add from lower left until end of left or upper left — at worst $O(n)$

Space Complexity $O(n)$

```
def find_upper_tangent(left, right, left_index, right_index):
```

-> $O(n)$

$O(1)$

```
    xl, yl = left[left_index]
    xr, yr = right[right_index]
```

Get x, y in

```
    if xl == xr:
        start_slope = float('inf')
```

Don't divide by 0

```
    else:
        start_slope = (yr - yl) / (xr - xl)
```

get slope

```
changed = True

while changed:  — O(n)
    left_visited = set()     ) points visited
    right_visited = set()
    changed = False
    while True:  ———— at worst O(log(n))
        left_visited.add(left_index)  O(1)
        next_left_index = (left_index - 1) % len(left)  O(1)
        if next_left_index in left_visited:  O(1)
            break
        left_visited.add(next_left_index)  O(1)
        lx, ly = left[next_left_index]  O(1)
        new_slope = (yr - ly) / (xr - lx)  O(1)
        if new_slope <= start_slope:  O(1)
            start_slope = new_slope  O(1)
            xl, yl = lx, ly  O(1)
            left_index = next_left_index  O(1)
            changed = True
        else:
            break

    while True:
        right_visited.add(right_index)
        next_right_index = (right_index + 1) % len(right)
        if next_right_index in right_visited:
            break
        right_visited.add(next_right_index)
        rx, ry = right[next_right_index]
```

O(n)

O(1) Reusing the same values over again

Add current point to set and then iterate through next points checking if slope is <= if it is store

Repeats here with constant space

repeat for other side O(log n)

```
        new_slope = (ry - yl) / (rx - xl)
        if new_slope >= start_slope:
            start_slope = new_slope
            xr, yr = rx, ry
            right_index = next_right_index
            changed = True
        else:
            break
    return left_index, right_index
```

Space Complexity O(n)

O(n)

```
def find_lower_tangent(left, right, left_index, right_index):
    xl, yl = left[left_index]
    xr, yr = right[right_index]

    if xl == xr:
        start_slope = float('-inf')
    else:
        start_slope = (yr - yl) / (xr - xl)

    changed = True

    while changed:
        left_visited = set()
        right_visited = set()
        changed = False
        while True:
            left_visited.add(left_index)
            next_left_index = (left_index + 1) % len(left)
            if next_left_index in left_visited:
                break
```

repeate again for lower tangents

This will follow the same space complexity as that in upper tangent,

```
            left_visited.add(next_left_index)
            lx, ly = left[next_left_index]
            new_slope = (yr - ly) / (xr - lx)
            if new_slope >= start_slope:
                start_slope = new_slope
                xl, yl = lx, ly
                left_index = next_left_index
                changed = True
            else:
                break

        while True:
            right_visited.add(right_index)
            next_right_index = (right_index - 1) % len(right)
            if next_right_index in right_visited:
                break
            right_visited.add(next_right_index)
            rx, ry = right[next_right_index]
            new_slope = (ry - yl) / (rx - xl)
            if new_slope <= start_slope:
                start_slope = new_slope
                xr, yr = rx, ry
                right_index = next_right_index
                changed = True
            else:
                break

    return left_index, right_index
```

The algorithm's total time complexity is O(n log n).

The initial sorting operation in compute_hull takes O(n log n) time. After sorting, the algorithm follows a divide-and-conquer pattern, where we recursively divide the points into left and right halves until we reach base case. Due to the Master Theorem where we have a = 2, b = 2, and d = 1 giving us O(nlog(n)).

For the recursive process:

- Each division splits the input into two equal halves
- The combination step, which includes finding and merging hulls, takes O(n) time

1. Division Phase - O(1):
   - Splitting points into left and right halves uses simple array slicing
   - The midpoint calculation is constant time
2. Finding Tangents - O(n):
   - Both find_upper_tangent and find_lower_tangent scan points in left and right hulls
   - While loops may iterate multiple times, but each point is visited at most once
   - Total operations remain linear in the size of input
3. Combining Hulls - O(n):
   - Iterating around the hulls to create the combined result
   - Each point is visited at most once during the merge

Space Complexity Analysis:

The total space complexity is O(n log n), determined by:

1. Recursive Space:
   - At each recursion level, we create new left and right subarrays
   - The recursion depth is log n
   - At each level, we need O(n) space for:
   - Divided point arrays

- Temporary storage in combination operations

- Visited sets while finding tangents

2. Additional Storage:

- Initial sorted array: O(n)

- Combined hull arrays at each level: O(n)

- Sets for tracking visited points: O(n)

The space accumulates across recursion levels since we maintain arrays at each level before combining results. With log n levels each requiring O(n) space, this gives us a total space complexity of O(n log n).

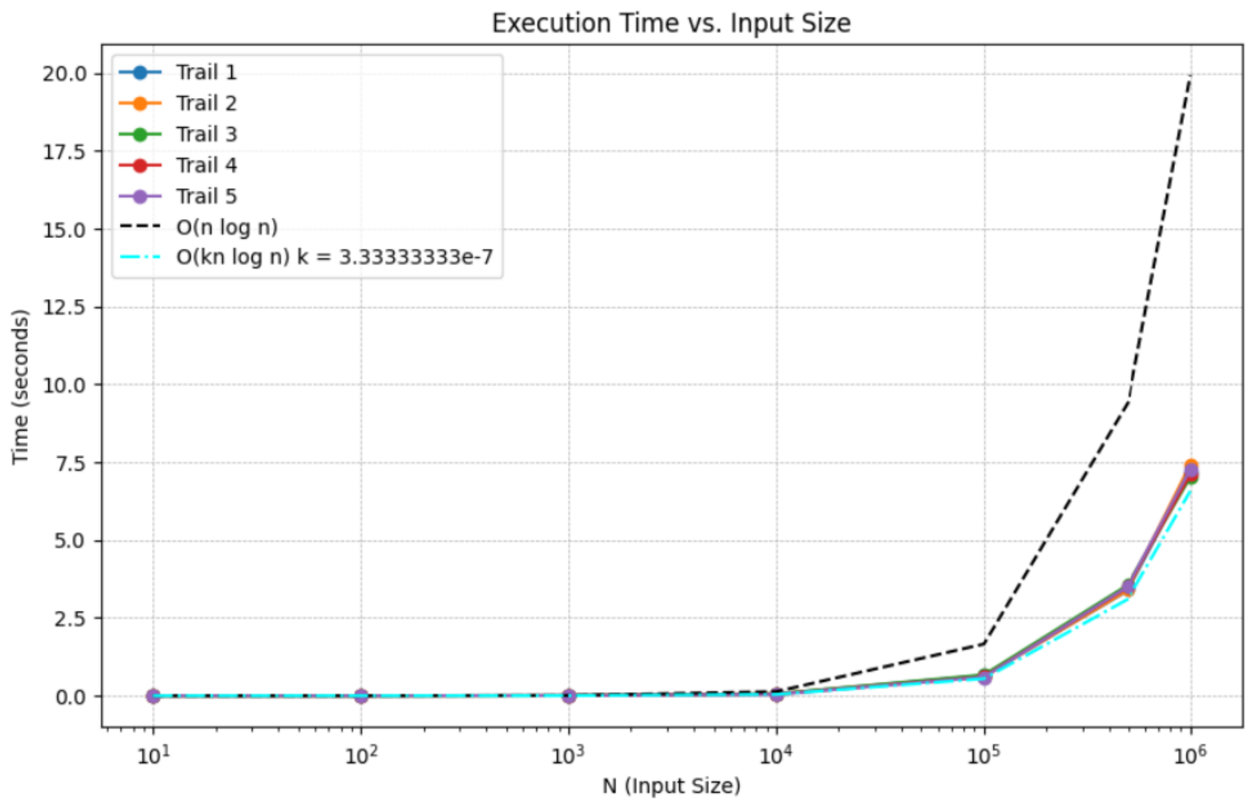
Analyzing Differences

The measured execution times generally follow the expected O(n log n), but there are some deviations at larger input sizes (see below). While the empirical data aligns with the theoretical complexity, the actual runtime appears to be scaled down by a constant factor.

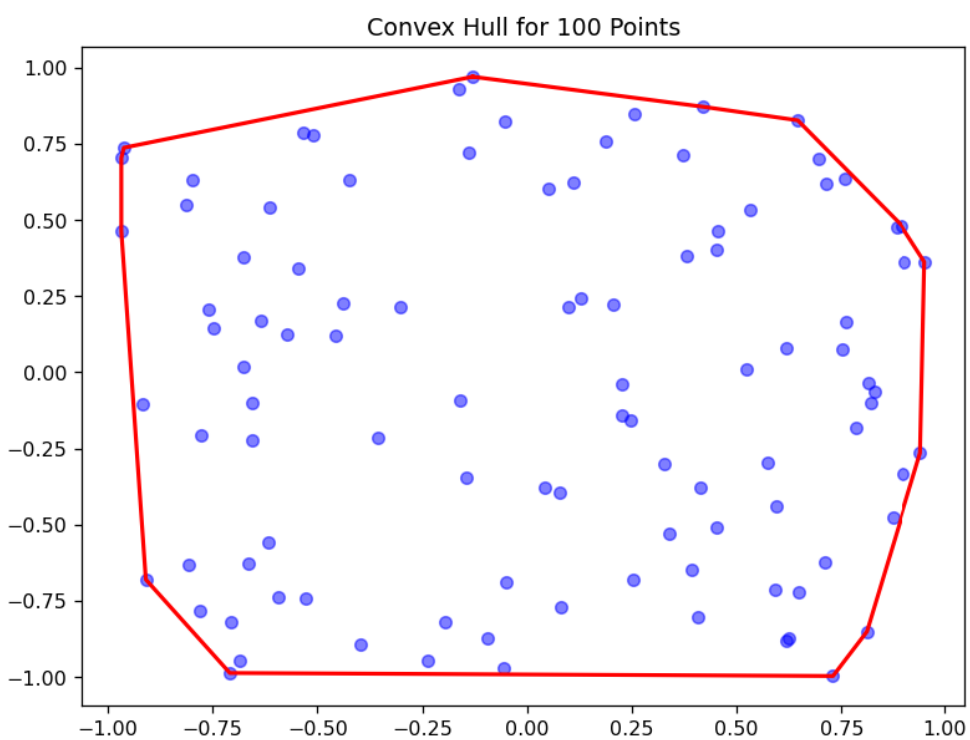
One notable observation is that the measured times are consistently about k = 3.33333333e-7 or O((3.33333333e-7)n log(n)). This suggests that while the algorithm's growth rate matches the theoretical prediction, the actual execution time is lower.

- **Hidden Constants** – The theoretical analysis does not account for constant-time operations like function calls, memory allocation, which could impact the performance.
- **Hardware Optimizations** – Modern CPUs, caching, and parallelism may contribute to a lower observed runtime than expected.

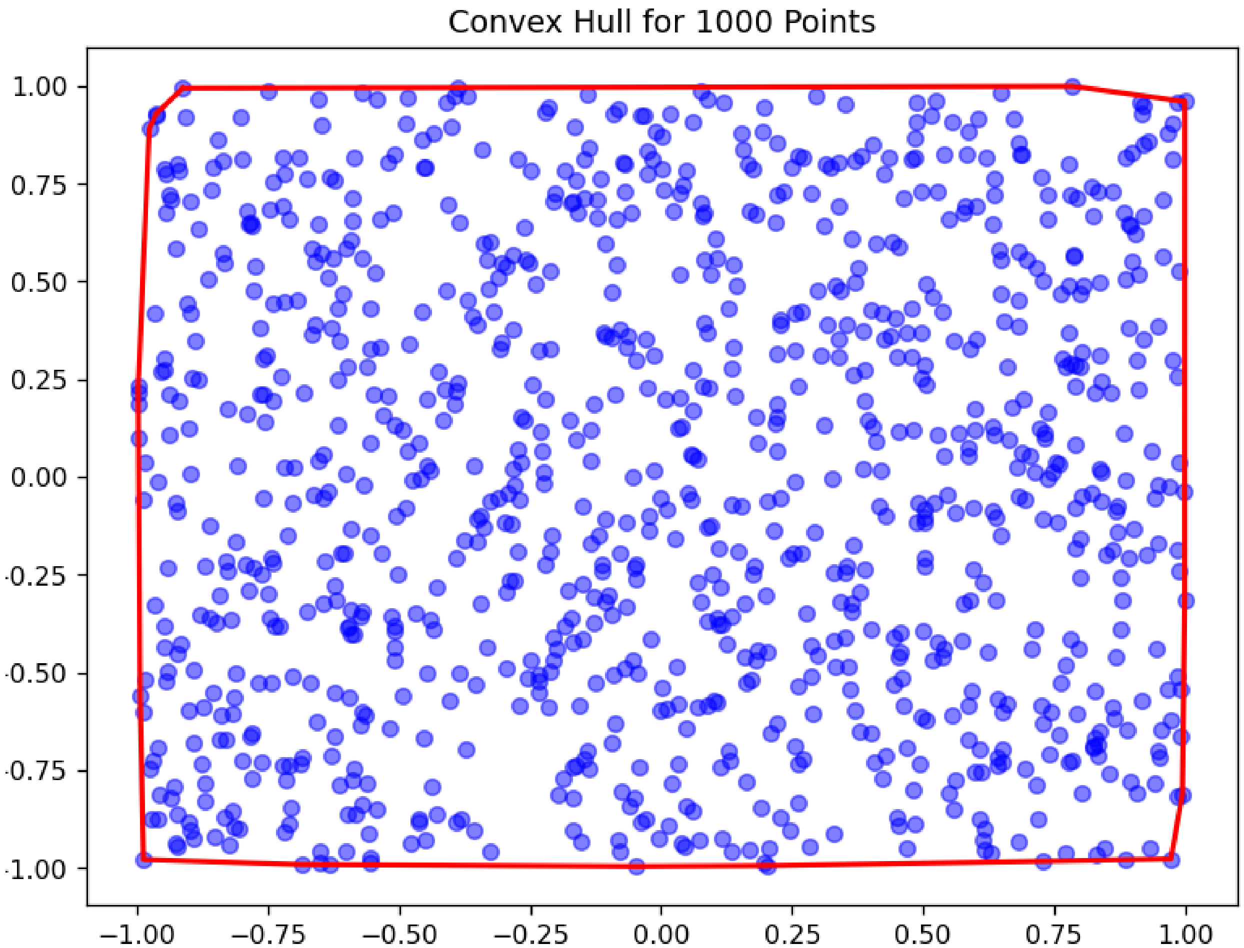
N =	10	100	1000	10000	100000	500000	1000000
Trail 1 time	0.0	0.0	0.008411	0.062716	0.652557	3.400225	7.186492
Trail 2 time	0.0	0.0	0.007681	0.065222	0.620895	3.415905	7.4317
Trail 3 time	0.0	0.0	0.008272	0.061779	0.673041	3.584279	7.024147
Trail 4 time	0.0	0.0	0.008129	0.050271	0.606982	3.51028	7.114009
Trail 5 time	0.0	0.0	0.010193	0.040866	0.597968	3.525613	7.28373



Execution Time vs. Input Size
Trail 1
Trail 2
Trail 3
Trail 4
Trail 5
O(n log n)
O(kn log n) k = 3.33333333e-7
Time (seconds)
N (Input Size)



Convex Hull for 100 Points


Convex Hull for 1000 Points
1.00
0.75
0.50
0.25
0.00
-0.25
-0.50
-0.75
-1.00
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00