Report 1: Convex Hull Computation using Graham's Scan Algorithm

a. Introduction:

The Graham's Scan approach is implemented in the provided C++ code to determine the convex hull of a collection of 2D points. The smallest convex polygon that covers all of the specified points in the plane is the convex hull.

b. Input and Output:

The program takes input for the number of points and their coordinates.

It prints the computed convex hull points (taking only edge points for the colinear case as given in test case 2) and the total count that represents minimum number of points on convex hull.

c. Code Analysis:

1. **Min Y-Coordinate Selection:**
    - The code finds the point with the lowest y-coordinate (**px**) and swaps it with the first point. This point serves as the pivot for the Graham's Scan algorithm.
2. **Sorting by Polar Angles:**
    - The remaining points are sorted based on their polar angles concerning the pivot point. This sorting step is crucial for the Graham's Scan algorithm.
3. **Graham's Scan Algorithm:**
    - The sorted points are processed using the Graham's Scan algorithm, which constructs the convex hull incrementally.
    - A stack is used to keep track of the convex hull points as they are discovered.
    - The algorithm iterates through the sorted points, pushing points onto the stack and popping points off the stack based on the orientation of the current point, the top point on the stack, and the next point from the sorted list.
    - The stack ultimately contains the points of the convex hull in counter clockwise order.

d. Time Complexity:

Sorting Step:

- The sorting step, performed using the merge sort algorithm, has a time complexity of O(nlogn). Sorting the points based on their polar angles with respect to the minimum Y coordinate point (px) dominates the time complexity.

Graham's Scan Algorithm:

- After sorting, the Graham's Scan algorithm processes the sorted points, which takes O(n) time. This is because each point is pushed onto the stack once and popped from the stack once. So Total Time Complexity = O(nlogn) + O(n) = O(nlogn)

e. Source:

class notes and slides and https://en.wikipedia.org/wiki/Graham_scan.

Report 2: Radix Sort Implementation for Integer Sorting

1. Introduction: The Radix Sort algorithm, a non-comparative integer sorting technique, is implemented in the provided C++ code. Radix Sort sorts numbers digit by digit, from the least significant digit (LSD) to the most significant digit (MSD), in contrast to comparison-based methods.

2. Input and Output:

- The program takes input for the number of integers and their respective values.
- The output displays the integers in sorted order using the Radix Sort algorithm.

3. Code Analysis:

a. Radix Sort Logic:

- **Digit Extraction:** The code first separates positive and negative integers into separate arrays and negative integers are multiplied by -1 and stored in an array.
- **Counting Sort on Digits:** For each digit place (from LSD to MSD), Counting Sort is applied to sort the integers based on the current digit.
- **Combining Positive and Negative Integers:** Finally, the sorted positive and negative integers are merged to obtain the overall sorted array by taking negative integers by traversing the negative integer array from the last towards first and multiplying by -1 and traversing positive array from first to last.

b. Time Complexity:

- **Digit Extraction:** The time complexity for finding the maximum digit in the array is O(n).
- **Counting Sort:** For each digit (in an integer with d digits), the Counting Sort step takes O(n) time since it processes each element once.
- **Overall Time Complexity:** Since the algorithm processes all digits, the overall time complexity is O(d * (n + k)), where d is the number of digits, n is the number of elements, and k is the range of values (10 for base-10 integers).

4. Source:

Class notes and slides and
https://en.wikipedia.org/wiki/Radix_sort#:~:text=Radix%20sort%2C%20such%20as%20the,used%2C%20such%20as%20insertion%20sort.

Report 3: Median of Median Algorithm

1. Introduction: The problem is solved using the Median of Medians algorithm to efficiently find the Kth largest element in linear time complexity O(N).

2. Code Analysis:

a. Median of Medians Algorithm:

The Median of Medians algorithm is used to find an approximate median element in linear time.

The input array is divided into subgroups of size 5, and medians of these subgroups are calculated.

The median of these medians is chosen as the pivot element.

b. QuickSelect Algorithm:

The QuickSelect algorithm is used to find the Kth largest element in the array.

The partition function is modified to use the Median of Medians algorithm to select the pivot element.

The partition function ensures that elements greater than or equal to the pivot are on the left side and elements less than the pivot are on the right side.

The QuickSelect algorithm is recursively applied to the appropriate subarray based on the pivot index until the Kth largest element is found.

3. Time Complexity:

The Median of Medians algorithm ensures linear time complexity O(N) for finding the approximate median.

The QuickSelect algorithm also runs in linear average time complexity O(N).

Recurrence Relation: $T(n) = T(n/2) + O(n)$ So, $T(n) = O(n)$ [using masters algorithm]

Overall, the solution guarantees linear time complexity.

4. Source: Class notes and Slides .