1)a)**Key Idea:** swap the median of L[n…3] with 1 ≤ i ≤ n-2 since the median cannot be a minimum or maximum by definition. By time I moves to n-1, the only two elements that were never a median and therefore never swapped out from L[n…2] will remain at the end of the list. These two elements are the minimum and maximum

Algorithm **MedianMinMax**;

**Input**: L[lower…upper], whose elements are drawn from a totally ordered set

**Output:** Two elements that are the minimum and maximum elements of L

**begin**

1. n = |L|
2. j = 1
3. while j < n-1 do
4. swap(L, median3(L[n-2], L[n-1], L[n] ), j)
5. j = j+1
6. if j == n-1 then
7. return L[n-1], L[n]
8. if j == n then
9. return L[1], L[1]

**end,**

Correctness:

We must prove that for all lists L drawn from a totally ordered set, such that |L|≥1, MedianMinMax returns the minimum and maximum element in L.

**Proof:**

For |L| = 1:

In line 1, n is set to 1, and in line 2, j is set to 1.

Since j n-1, control does not enter the while loop

Since j ≠ n-1, control does not execute inside the if statement on line 6 and control skips to line 8

Since j == n, control executes line 9 inside the if statement on line 8 and MedianMinMax returns L[1], L[1] and since L only has one element, L[1] is both the min and max of L. Thus, algorithm MedianMinMax returns the minimum and maximum of a list L when |L| = 1

For |L| = 2:

In line 1, n is set to 2, and in line 2, j is set to 1.

Since j n-1, control does not enter the while loop

Since j == n-1, control executes line 7 inside the if statement on line 6 and MedianMinMax returns L[n-1], L[n] and since L only has two elements, L[n-1] and L[n] must be the min and max of L. Thus, algorithm MedianMinMax returns the minimum and maximum of a list L when |L| = 2

For |L| > 2:

**Lemma 1:** for 1 ≤ k < n-2, if and when control reaches line 3 for the kth time, L[n-2…n] contains the min and max of L[1…k-1]

**Proof:**(*Induction Basis*) When k = 2, The list L[1…k-1] has a single element, which was swapped into L[n-2…n] with the previous iteration of the while loop, thus L[n-2…n] contains the min and max of L[1…k-1]

(*Inductive Hypothesis*) Suppose Lemma 1 holds for k=m (m<n)

(*Inductive Step*) When control reaches Line 3 for the (m+1)th time, by the inductive hypothesis, we know that the min and max of L[1…m] is in L[n-1…n], and j≥2 since control did not exit the while loop.

Next control moves to line 4, where the median of L[n-2…n] is swapped with L[m+1]. Since the median of L[n-2…n] must be less then one element in L[n-2…n] and greater then another element in L[n-2…n], it is not the min or max of L[1…m] and therefore not the min or max of L[1…m+1] . So, the min or max elements of L[1…m] are still in L[n-2…n].

If L[m+1] is the min or max of L[1…m+1], then it is swapped with the median of L[n-2…n] and L[n-2…n] now contains the max and min of L[1…m+1]

otherwise, by lemma 1, L[n-2…n] already contained the min and max of L[1…m+1]

When control reaches line 3 for the (n-2)th time, j = n-2 and control enters the while loop. Now since by lemma 1, L[n-2…n] contains the min and max of L[1…n-3], and L[n-2…n] must contain its own min and max, therefore, L[n-2…n] contains the min and max of L.

In line 4 we swap the median of L[n-2…n] with L[j=n-2], there are three cases:

Case 1: The median is in L[n-2]

In this case, the minimum and maximum of L[n-2…n] are in L[n-1...n] (Since the minimum and maximum of three elements are the two elements that are not the mean). Furthermore, the index returned by median3 will be n-2, thus the swap will cause no change and the minimum and maximum will remain in L[n-1…n] after line 4 executes

Case 2: The median is in either L[n-1]

In this case, L[n-2] and L[n] are the minimum and maximum and the index returned by the call to median3 will be n-1. Thus, the median(L[n-1]) will be swapped with L[n-2] and L[n-1] will now hold the minimum and L[n] will hold the maximum, or vice versa. Thus, the minimum and maximum will be in L[n-1…n] after line 4 executes

Case 2: The median is in either L[n]

In this case, L[n-2…n-1] contain the minimum and maximum and the index returned by the call to median3 will be n. Thus, the median(L[n]) will be swapped with L[n-2] and L[n-1] will now hold the minimum and L[n] will hold the maximum, or vice versa. Thus, the minimum and maximum will be in L[n-1…n] after line 4 executes

In line 5, J is incremented to n-1( n-1) and control exits the loop with the minimum and maximum in L[n-1…n].

Now control moves to line 7 since the conditions in line 6 was met( j == n-1 ) and L[n-1] and L[n] are returned. Thus, algorithm MedianMinMax returns the minimum and maximum of a list L when |L| > 2

Therefore, Algorithm MedianMinMax returns the minimum and maximum of list L when |L| ≥ 1

█

Time Complexity:

**Theorem:** Algorithm MedianMinMax performs Θ(n) calls to median3 in the worst-case scenario to find the min and max of a list of n elements

**Proof:** The worst-case occurs when the n>2. In this case, the while loop on line 2 is executed n-2 times and in each iteration, one call is made to median3. No calls are made to median3 outside the while loop therefore algorithm MedianMinMax makes at most n-2 calls to median3. Thus, algorithm MedianMinMax is Θ(n-2) = Θ(n)

1)b) **Key Idea:** Create a comparison operator out of Median3 that is equivalent to either a less then or greater then operator and replace the comparison operator in an O(nlgn) sorting algorithm, such as merge sort, with this median3 comparison operator.

Algorithm **MedianSortMain:**

**Input:** L[lower…upper];

**Output:** L[lower…upper] sorted in ascending or descending order;

**begin**

1. **if**(|L| > 2)
2. Edge[2] = **MedianMinMax(**L**)**
3. return **MedianSort**(L)
4. **else** return L

Note: Edge is a global accessible list of size two containing the min and max element of L

**end.**

Algorithm **MedianSort**;

**Input**: L, lower, upper;

**Output:** L[lower…upper] sorted in ascending or descending order;

**begin**

1. **if** (lower < upper) **then**
2. **MedianSort**(L, lower, );
3. **MedianSort**(L, , upper);
4. return **MedianMerge** (L[lower…], L[+1…upper])
5. return L

**end.**

Algorithm **MedianMerge;**

**Input:** Two list A[1…m] and B[1…n], sorted In descending order if Edge[1] is the max of L and ascending order if Edge[1] is the min of L;

**Output:** A sorted list C[1…m+n] which contains all the elements of A and B

**begin**

1. indexA = 1; indexB = 1; indexC = 1;
2. **while**(indexA ≤ m and indexB ≤ n) **do**
3. **if**(**CloserToEdge**(A, B, indexA, indexB))
4. **then** C[indexC] = A[indexA]; indexA = indexA + 1;
5. **else** C[indexC] = B[indexB]; indexB = indexB + 1;
6. indexC = indexC+1;
7. **if** (indexA>m)
8. **then** copy B[indexB…n] into C[indexC…m+n];
9. **else** copy A[indexA…n] into C[indexC…m+n];

**end.**

Algorithm **CloserToEdge;**

**Input:** Two lists, A and B, the index of the element being compared from A, indexA, the index of the element being compared from B, indexB, and globally accessible list, Edge

**Output:** A true or false value

**begin**

1. c = median3(A[indexA], B[indexB], Edge[1])
2. **if**(c == 1)
3. **then** return True
4. **if**(c == 2)
5. **then** return False

**end.**

Correctness:

We must prove that for all lists L drawn from a totally ordered set, such that |L|≥1, MedianSortMain returns the list L sorted in ascending or descending order.

**Proof:**

If |L| < 2

On line 1 control does not enter the conditional if-block because |L| is less then two. Control skips to line 4 where L is returned and is already sorted because a list of one element is always sorted and a list of two elements is always sorted either in ascending or descending order.

Therefore if |L| < 2 then *MedianSortMain* returns the list L sorted in ascending or descending order.

If |L| > 2

On line 1 of Algorithm *MedianSortMain*, control enters the conditional if-block because |L| > 2. On line 2 the Minimum and Maximum element are saved in unknown order in the globally accessible list Edge and on line 3 the result of *MedianSort(1,2, L)* is returned. Thus, if the result of *MedianSort(1,2,L)* when |L|>2, and Edge contains the Minimum and Maximum elements of L, is a List sorted in ascending or descending order then *MedianSortMain* returns the list L sorted in ascending or descending order.

**Lemma 2:** If Edge[1] is the max of L, then;

CloserToEdge returns true when A[indexA] > B[indexB] and false otherwise

…and, If Edge[1] is the min of L, then;

CloserToEdge returns true when A[indexA] < B[indexB] and false otherwise

**Proof:** Case 1: Edge[1] is the max of L;

Since Edge[1] is the max element in L, and there are no duplicates, Edge[1] > A[indexA] and Edge[1] > B[indexB]. There are two sub cases:

Case 1.i: A[indexA] > B[indexB]

By transitivity of >, Edge[1] > A[indexA] > B[indexB]

Thus, the median of A[indexA], B[indexB], and Edge[1] will be A[indexA],

and on line 1, c will be set to 1.

Next control will move to line 2, pass the conditional if-statement and execute line 3 returning true

Case 1.ii: B[indexB] > A[indexA]

By transitivity of >, Edge[1] > B[indexB] > A[indexA]

Thus, the median of A[indexA], B[indexB], and Edge[1] will be B[indexB],

and on line 1, c will be set to 2.

Next control will move to line 2, fail the conditional if-statement and move to line 4. It will the pass the conditional if-statement and execute line 5 returning false

Thus, when Edge[1] is the max of L, CloserToEdge returns true when A[indexA] > B[indexB] and false otherwise

Case 2: Edge[1] is the min of L;

Since Edge[1] is the min element in L, and there are no duplicates, Edge[1] < A[indexA] and Edge[1] < B[indexB]. There are two sub cases:

Case 1.i: A[indexA] < B[indexB]

By transitivity of <, Edge[1] < A[indexA] < B[indexB]

Thus, the median of A[indexA], B[indexB], and Edge[1] will be A[indexA],

and on line 1, c will be set to 1.

Next control will move to line 2, pass the conditional if-statement and execute line 3 returning true

Case 1.ii: B[indexB] < A[indexA]

By transitivity of <, Edge[1] < B[indexB] < A[indexA]

Thus, the median of A[indexA], B[indexB], and Edge[1] will be B[indexB],

and on line 1, c will be set to 2.

Next control will move to line 2, fail the conditional if-statement and move to line 4. It will the pass the conditional if-statement and execute line 5 returning false

Thus, when Edge[1] is the min of L, CloserToEdge returns true when A[indexA] < B[indexB] and false otherwise

Thus, If Edge[1] is the max of L, then;

CloserToEdge returns true when A[indexA] > B[indexB] and false otherwise

…and, If Edge[1] is the min of L, then;

CloserToEdge returns true when A[indexA] < B[indexB] and false otherwise

**Lemma 3 :** Algorithm MedianMerge correctly merges the two sorted lists; That is, the ith element moved into the output list is the ith largest element if Edge[1] is the max of L or the ith smallest element if Edge[1] is the min of L.

**Proof:**(*Induction Basis*) When 1st element is being moved into the list, i = 1. We know that control has reached line 3 for the first time, therefore indexA = 1, indexB = 1, indexC = 1 and the output list C is empty.

Regardless of whether the if statement evaluates to true or false, after control moves to line 6, there will be one element in C that is the 1st element moved into C and is trivially the 1st smallest and 1st largest element in C. Thus the 1st smallest element moved into the output list is the ith largest element if Edge[1] is the max of L or the ith smallest element if Edge[1] is the min of L.

(*Inductive Hypothesis*) Suppose Lemma 3 holds for k=m (m<n)

(*Inductive Step*) When the (m+1)th element is being moved into the list, by the inductive hypothesis, we know that the each of the ith elements from 1...m are the ith greatest element in the combined list C if Edge[1] is the max of L and the ith smallest element in the combined list C if Edge[1] is the min of L. We also know that indexC = i, indexA = j, and indexB = k, when control is at line 3. Furthermore, we know that C has i elements, and that A and C are in descending order if Edge[1] is the max of L, and in ascending order if Edge[1] in the min of L.

Now there are two cases, each with two subcases:

Case 1: A[indexA] > B[indexB].

Case 1.i: Edge[1] is the max of L,

By Lemma 3, control will move to line 4, and A[indexA] will be placed in the (i+1)th place in C. Since A and B are in descending order we know that all i elements in C are larger then A[indexA] and the remaining elements in A and B are smaller, therefore the (i+1)th element added to C is the (i+1)th largest element in C

Case 1.ii: Edge[1] is the min of L,

By Lemma 3, control will move to line 5, and B[indexB] will be placed in the (i+1)th place in C. Since A and B are in ascending order we know that all i elements in C are smaller then B[indexB] and the remaining elements in A and B are larger, therefore the (i+1)th element added to C is the (i+1)th smallest element in C

Case 2: A[indexA] < B[indexB].

Case 2.i: Edge[1] is the max of L,

By Lemma 3, control will move to line 5, and B[indexB] will be placed in the (i+1)th place in C. Since A and B are in descending order we know that all i elements in C are larger then B[indexB] and the remaining elements in A and B are smaller, therefore the (i+1)th element added to C is the (i+1)th largest element in C

Case 2.ii: Edge[1] is the min of L,

By Lemma 3, control will move to line 4, and A[indexA] will be placed in the (i+1)th place in C. Since A and B are in ascending order we know that all i elements in C are smaller then A[indexA] and the remaining elements in A and B are larger, therefore the (i+1)th element added to C is the (i+1)th smallest element in C

Thus, Algorithm MedianMerge correctly merges the two sorted lists; That is, the ith element moved into the output list is the ith largest element if Edge[1] is the max of L or the ith smallest element if Edge[1] is the min of L.

**Theorem 1:** Algorithm *MedianSort* correctlysorts a list L with |L|>2 into ascending or descending order

Note: In the Inductive Basis, and Inductive Step, Edge[1] is consistently the max element of L for the entire operation on L or the minimum element of L for the entire operation on L. Thus, without loss of generality, we can assume that MedianMerge will always consistently merge two sorted lists into a list that is in the same order of the two sublists it is merging.

**Proof:**(*Induction Basis*) When |L| = 3, MedianSort is called on the two sublists L[1…2] and L[3]. The first list is further divided into two sublists L[1] and L[2]. The two sublists L[1] and L[2] are already sorted since they only contain one element, and by lemma 3, are correctly merged to create the sorted sublist L’[1...2]. The list L[3] is already sorted since it only contains one elements and is then merged with L’[1…2]. By lemma 3, we know these two lists are merged correctly resulting in a sorted list L’[1…3] which is then return by Median sort. Thus, MedianSort correctly sorts a list L, were |L| = 3.

(*Inductive Hypothesis*) Assume Lemma Theorem 1 holds for |L|=m(m<n)

(*Inductive Step*) When |L| = m+1 then the list L is divided into two sublists

L[1…] and L[… m]. Since both lists are of a size less then m, by the Inductive Hypothesis, they are both correctly sorted by MedianSort on lines 2 and 3. Then by lemma 3, the two lists are correctly merged by MedianMerge resulting a sorted list L’[1…m] that is then returned on line 5.

Therefore, MedianSort correctly sorts a list L with |L|>2 into ascending or descending order

Thus, the result of *MedianSort(1,2,L)* when |L|>2, and Edge contains the Minimum and Maximum elements of L, is a List sorted in ascending or descending order

Therefore, Algorithm *MedianSortMain* returns the list L sorted in ascending or descending order for |L| > 2

Therefore, Algorithm *MedianSortMain* returns the list L sorted in ascending or descending order for |L| ≥ 1

■

Time Complexity:

Theorem: Algorithm MedianSortMain performs Θ(nlgn) calls to median3 in the worst-case scenario to find the min and max of a list of n elements.

**Proof:** The worst-case occurs when |L|>3. In this case, MedianMinMax is called on L once, which has already been shown to be Θ(n) followed by a call to MedianSort on L.

Let T(n) be the number of calls to median3 required to sort a list of n elements with MedianSort.

Let: and

Using general formula for solving recurrences, we have

and

f(n) = (n-1) - Θ(n) = Θ()

Θ() = Θ()

Similarly, = Θ() = Θ()

Thus, T(n) = Ω(nlgn) and T(n) = O(nlgn)

By Lemma 0.1(b), T(n) = Θ()

Therefore, the time complexity of MedianSortMain is Θ(n) + Θ(nlgn) = Θ(nlgn) by theorem 0.7

2)

**Key Idea:** Find a O(nlgn) subset of all the O(n2) intersections and use an already known O(nlgn) convex hull finding algorithm. To do this we sort the lines by their slope and return the intersections of successive lines in the sorted list.

Algorithm **ConvexHull;**

**Input:** A list of straight lines, L[1…n]

**Output:** A set of intersections that map to the vertices of the smallest convex polygon containing all the intersections of all the lines in the L

**Begin**

1. Obtain the slope of every line
2. Sort the lines by their slope in descending order in list L[0…N-1]
3. while m < N-1
4. I[m] = intersection(L[m], L[(m+1) mod N])
5. return Graham-Scan(I)

**end.**

Correctness

**Proof of Correctness:** We must prove that the Graham Scan of I = the Convex Hull of the set of all intersections of the lines in L. To do this, we must prove Theorem 2.

**Theorem 2:** let CH be the list of vertices that make up the smallest convex polygon containing all the intersections of the lines in L, (i.e. the convex hull of set of points that correspond to the intersections of the lines in L), then CH ⊆ I.

**Proof**(Proof by contradiction)

Suppose CH⊈I.

CH⊈I

⇒ ~(CH⊆I)

⇒ ~(∀x∈CH, x∈CH → x∈I)) Definition of subset

⇒ ∃x∈CH, ~(x∉CH ∨ x∈I)

⇒ ∃x∈CH, x∈CH ∧ x ∉ I

⇒ x∈CH ∧ x ∉ I E.I.

Since x is an element of CH, and CH is a set of points corresponding to intersection of the lines in L, we know that x itself is an intersection. Thus, we can say, with out loss of generality, that x is the intersection of two lines Lq and Lr such that *slope(Lq) > slope(Lr).*

Additionally, since x is not an element of I, we know that these two lines are not consecutive in L, implying that there exists a line Ls such that *slope(Lq) > slope(Ls) > slope(Lr).*

Finally, since x is not in I, we also know that *Lq ≠L[0] or Lr ≠ L[N-1],* since intersection(L[N], L[N mod N]) is an element of I.

Now, Ls must intersect with Lq and Lr, let these two intersects be *sq* and *sr* respectively

There are two cases

Case 1: Point *sq* is to the right of *sr* on straight line Lk (Case 2 is symmetrical)

Now since if Lq ≠L[N] or L[0] *≠ Lr*,we know that there exists a line Lt such that the slope of slope(Lq) > slope(Lt) or slope(Lq) < slope(Lt). let *tq* and *tr* be the intersection of Lt with Lq and Lr, respectively.

a)slope(Lt) > slope(Lq).

If *tr* is to the right of *tq*, then x would be on the line segment joining *tq* and *sq*. (Figure 1) This means that x is not a convex vertex (since convex vertex’s have an interior angle of less then 180°). Thus, x cannot be a vertex in the convex hull and therefore is not an element of CH. A contradiction.

Figure 1

A picture containing object, skiing, sky

Description generated with high confidence

If *tr* is to the left of *tq,* then x would be on the line segment joining *tr* and sr.

(Figure 2) This means that x is not a convex vertex (since convex vertex’s have an interior angle of less then 180°). Thus, x cannot be a vertex in the convex hull and therefore is not an element of CH. A contradiction.

A close up of a wire fence

Description generated with high confidence

Figure 2

b) slope(Lt) < slope(Lr)

If *tr* is to the left of *tq*, then x would be on the line segment joining *tq* and *sq*.

(Figure 3) This means that x is not a convex vertex (since convex vertex’s have an interior angle of less then 180°). Thus, x cannot be a vertex in the convex hull and therefore is not an element of CH. A contradiction.

Figure 3

A picture containing sky, skiing, object

Description generated with very high confidence

If *tr* is to the left of *tq*, then x would be on the line segment joining *tq* and *sq*.

(Figure 4) This means that x is not a convex vertex (since convex vertex’s have an interior angle of less then 180°). Thus, x cannot be a vertex in the convex hull and therefore is not an element of CH. A contradiction.

A picture containing sky, skiing, object, antenna

Description generated with very high confidence

Figure 4

Thus, by contradiction, we have CH ⊆ I

Since, by Theorem 3 CH ⊆ I, and CH is the convex hull of L, Finding the convex hull of I will give us the convex hull of our input list. Thus, it follows the algorithm is correct

■

Time Complexity:

On line 1: Obtaining the slope of a line can be done in constant time O(1), Thus, obtaining the slope of all n lines would take O(n) time.

On line 2: Sorting with an algorithm like merge sort would take O(nlgn) time.

In the while loop, obtaining the intersection of one line can be done in O(1) time so obtaining the intersection of all n lines would take O(n) time.

Finally, the algorithm Graham Scan takes O(nlgn) time so the algorithm ConvexHull takes

O(n) + O(nlgn) + O(n) + O(nlgn) time. By theorem 0.7, we have O(nlgn) time.

Therefore, ConvexHull takes O(nlgn) time

3) Show the problem of determining if a list of n elements drawn from a totally ordered set are all distinct has a lower bound of Ω(nlgn) number of comparisons needed to solve the problem.

**Proof:**

**Theorem 2:** An algorithm that can determine if there are any duplicates in a list drawn from a totally ordered set must compare each element xi with the smallest element larger than or equal to xi

**Proof:**(proof by contradiction) Suppose an algorithm that can determine if there are any duplicates in a list drawn from a totally ordered set didn’t have to compare each element xi with the smallest element larger then xi.

Suppose we have an input lists L of size n and sorted in ascending order, where all elements of L are distinct except that L[i] = L[i+1]. When the algorithm gets to L[i], it will not compare it to L[i+1] since L[i+1] is the smallest element larger then L2[i] and will not determine that they are the same, thus will determine that all elements in L2 are distinct even those this is not true. A contradiction.

Thus. by contradiction, an algorithm that can determine if there are any duplicates in a list drawn from a totally ordered set must compare each element xi with the smallest element larger than or equal to xi

(proof by contradiction) Assume there was an algorithm, Π1, that can determine if a list of n elements drawn from a totally ordered set are all distinct with o(nlgn) number of comparisons.

By Theorem 2, we know an algorithm that solves Π1 must compare each element xi with the smallest element larger then xi. If we modify that algorithmto output the element compared with xi,and call this modified algorithm Π2,then we can see the output of Π2 is the sorted input list. Furthermore, this modification does not change the number of comparisons done.

Π2 and sorting algorithms both take lists drawn from a totally ordered set as input and both produce the corresponding sorted list. Therefore, sorting algorithms are O(1)-reducible to Π1 … (I)

It is known that sorting algorithms preform Ω(nlgn) comparisons … (II)

O(1) is o(nlgn) … (III)

By (I), (II), and (III), Π2 is Ω(nlgn)

Since the modifications to change Π1 to Π2 did not change the number of comparisons made, Π1 is also making Ω(nlgn) comparisons. This contradicts our assumption; Thus, our assumption must be wrong.

Therefore, by contradiction an algorithm that can determine if a list of n elements drawn from a totally ordered set are all distinct has a lower bound of Ω(nlgn) number of comparisons needed to solve the problem.