Question1:

```
Input: X[1..n], A[1..n]    // 1-based index
Output: X reordered according to A
1. Create array of pairs P[i] = (A[i], i) for i = 1..n
2. Sort P by first element (A[i])
3. For i = 1 to n:
        if i is not visited:
            temp = X[i]
            j = i
            while j not visited:
                k = P[j].second
                swap(X[k],temp)
                mark j visited
                j = k
```

Question2:

(a)

Step 1: Represent the algorithm as a decision tree

Each comparison x?y is a node with two possible outcomes:

x>y

x<y (no ties because all numbers are distinct)

Leaves correspond to final outcomes: the algorithm outputs max and min.

Step 2: Count the number of possible outcomes

For n distinct numbers, the number of possible (min, max) pairs:

#leaves=n(n−1)

Because there are n choices for the maximum and n−1 remaining choices for the minimum.

Step 3: Relate leaves to height

Let h = height of the decision tree = number of comparisons in the worst case

Each comparison has 2 outcomes → binary tree:

#leaves⩽2^h

To distinguish all possibilities:

2^h >= n(n-1)

Take logarithms:

h >= [log2(n(n-1)]

Step 4: Lower bound from decision tree

Worst-case comparisons >= [log2(n(n-1)]

This is a valid lower bound, but note that: [log2(n(n-1)]≈2log2n(for large n)

Observation: This is weaker than the adversary argument bound [3n/2]−2 for large n

Decision tree bound:

Worst-case >= [log2(n(n-1)]

Adversary argument bound:

Worst-case ≥ [3n/2]−2

(b)

Explanation:

Decision tree argument counts the total number of possible outputs.

It assumes each comparison gives at most 1 bit of information.

Gives a general information-theoretic lower bound.

Adversary argument considers structural constraints of max/min:

Each element(except maximum) must lose at least once to be maximum, or win at least once to be minimum(except minimum).

Gives a tighter bound based on the pairwise comparison structure

Both bounds are correct, but the adversary argument is stronger for this problem because it uses specific structure, not just counting possible outcomes.

Question 3:

Step 1: Observing the property

Since $|A[i+1]-A[i]| \leq 1$, the array is “almost consecutive”:

Each step increases or decreases by at most 1

If A[i]=v, then z can be at most $|z-v|$ positions away from i

Step 2: Designing the optimal search algorithm (Jump-Step Search)

Idea:

Start at position i=1

While $A[i] \neq z$: Jump exactly $|A[i]-z|$ positions, because each step can change by at most ±1

Pseudocode:

```
i = 1
while A[i] != z and i <= n:
    i = i + abs(A[i] - z)
if i > n: return "not found"
return i
```

Explanation:

Each step moves as far as possible without skipping the target

This ensures minimal number of comparisons

Step 3: Complexity

Each comparison eliminates as many positions as possible

Worst-case number of comparisons ≤ n

Time complexity: $O(|z-A[1]|) \leq O(n)$

This is optimal for arrays with $|A[i+1]-A[i]| \leq 1$

Step 4: Decision Tree Argument for Optimality

Each comparison A[i]?=z is a decision node in a tree:

1.A[i]=z → Found

2.A[i]<z → Must move right

3.A[i]>z → Must move left

Observation:

Due to $|A[i+1]-A[i]| \leq 1$, each comparison can eliminate at most $|A[i]-z|$ positions

No algorithm can skip more than $|A[i]-z|$ elements without risking missing z

Jump-step search maximizes the number of positions eliminated per comparison

Hence, its decision tree has minimal height → minimum comparisons → algorithm is optimal

Step 5: Summary

Algorithm: Jump-step search $i \leftarrow i+|A[i]-z|$ until A[i]=z

Time complexity: $O(|z-A[1]|) \leq O(n)$

Optimality proof:

Decision tree argument shows any algorithm must make at least as many comparisons, because each comparison can only reduce the candidate positions by ±1 per element.

Jump-step search achieves the shortest possible decision path, therefore optimal.

Question4:

Step 1: Build a tournament tree

1.Consider finding the maximum element as a tournament:

Compare elements in pairs.

The larger element "wins" and advances to the next round.

Continue until only one element remains — this is the maximum.

2.In this tournament:

The number of comparisons to find the maximum is exactly $n-1$.

Each element except the maximum loses exactly once.

Step 2: Identify candidates for the second largest element

1.The second largest element must have lost to the maximum at some point during the tournament.

2.Trace the path from the maximum back to the leaves in the tournament tree.

This path has length $[\log n]$, because the height of a complete binary tree with n leaves is $[\log n]$.

Only the elements that lost to the maximum along this path are candidates for the second largest.

Step 3: Find the second largest among the candidates

1.There are at most $[\log n]$ candidates.

2.Compare them in pairs (or sequentially) to find the largest among them.

This requires $[\log n]-1$ additional comparisons.

Step 4: Total comparisons

Comparisons to find the maximum: $n-1$

Comparisons to find the second largest among candidates:$[\log n]-1$

Total comparisons$=(n-1)+([\log n]-1)=n+[\log n]-2$

Step 5: Conclusion

Using the tournament method, we can find the second largest element in at most $n+[\log n]-2$ comparisons in the worst case.

This method is optimal, because each element except the maximum must be compared at least once to ensure it is not the largest.

Question 5:

General analysis for odd group size g

Number of groups $\approx n/g$.

In each group of g (odd), the group median is the (g+1)/2-th smallest element in that group, so it is $\geqslant (g+1)/2$ elements of its group (including itself).

The median-of-medians/pivot is at least the median of the [n/g] group-medians, hence at least half of the group-medians are $\geqslant$ pivot.

Therefore a lower bound on the number of elements ≥ pivot is (number of medians $\geqslant$ pivot) $*(g+1)/2 > \approx n/2g*(g+1)/2 = n*(g+1)/4g$

So a fraction g+1/4g of the whole array is guaranteed to be $\geqslant$ pivot, and symmetrically the same fraction is $\leqslant$ pivot.

Consequently the size of the larger recursive side after partitioning is at most $n*(1-(g+1/4g))=n*(3g-1/4g)$

The usual recurrence for the running time T(n) becomes T(n) <= T(n/g) + T((3g-1/4g)*n)+O(n)

T(n/g) is the work to find the median of medians (we recursively find the median of the ≈n/g medians).

T((3g−1)n/(4g)) is the cost of recursing on the larger side after partitioning.

O(n) is the linear cost for grouping, finding per-group medians, and partitioning.

When Case g=3:

Guaranteed $\geqslant$ pivot fraction =(3+1)/(4*3)=1/3.

Worst-case larger side =1−1/3=2/3.

Recurrence:$T(n) \leqslant T(n/3)+T(2n/3)+O(n)$.

The coefficients of the subproblems sum to 1(1/3+2/3=1). Recurrences with two subproblems whose size fractions sum to 1 and with an additive O(n) term solve to T(n)=O(nlogn)(O(n)*O(logn)). Thus group size 3 does not guarantee linear-time selection — it yields a worse asymptotic bound.

When Case g=7:

Guaranteed ≥ pivot fraction: (7+1)/(4*7)=2/7≈0.286

Explanation: Each group has 7 elements, the median is the 4th element, so at least 3 elements per group are ≤ median. Median-of-medians is ≥ half of the medians, giving a guaranteed fraction of (3+1)/(2*7)=4/14. Adjusting for tighter bound, ≥ 2n/7 are ≤ pivot.

Worst-case larger side: 1−2/7=5/7

Recurrence:

$T(n) \leq T(n/7)+T(5n/7)+O(n)$

Subproblem coefficients sum: $n/7+(5/7)n=6n/7<n$

Since the sum of the recursive subproblem sizes is strictly less than n, the recurrence solves to T(n)=O(n).

Conclusion:

Groups of 3: the recurrence becomes $T(n) \leq T(n/3)+T(2n/3)+O(n)$. The subproblem fractions sum to 1, so the algorithm does not guarantee linear time (it leads to O(nlogn) behavior in the worst case). Thus g=3 is insufficient.

Groups of 5: classic choice. Recurrence $T(n) \leq T(n/5)+T(7n/10)+O(n)$. The fractions sum to 0.9<1, so T(n)=O(n). Linear-time guaranteed.

Groups of 7 or larger (odd): the same style of analysis shows T(n)=O(n) for any fixed odd g≥5,

because the sum of subproblem fractions is <1. But the constant factors increase with g, so larger g are asymptotically linear but practically worse constants.

Question 6:

We have: two sorted lists

We want to merge them into a single sorted list

A and B, each of size n: A = [a1, a2, ... , an], B = [b1, b2, ... , bn],

A comparison-based merge algorithm works by repeatedly comparing the current smallest elements from each list and taking the smaller one.

Step 1: Understanding the worst-case scenario

To get the worst-case number of comparisons, we need to imagine the scenario that forces the algorithm to make the maximum number of comparisons.

Suppose the elements are interleaved perfectly:

a1 < b1 < a2 < b2 < a3 < b3 < ... < an < bn

In this case, every time we pick an element, we must compare the front elements of both lists to decide which one to take.

Step 2: Counting the comparisons

Define:

i = number of elements taken from A

j = number of elements taken from B

At each step:

1.Compare the current ai with bj

2.Remove the smaller one and advance the pointer.

Observation:

Once all elements from one list are used, no further comparisons are needed; you just append the remaining elements of the other list.

Therefore, the maximum comparisons occur just before one list is exhausted.

Step 3: Maximum number of comparisons

Total elements in merged list: 2n

Let's say we exhaust list A first:

To take all n elements from A, each element (except the last one) must be compared with an element from B.

How many comparisons?

1.The first element of A is compared with the first element of B

2.The second element of A is compared with the next element of B, etc.

When we take the last element of A, all elements of B except one may have been compared. At this point, we don't need to compare anymore; we just append the remaining B elements.

Hence: The total number of comparisons is: n + (n - 1) = 2n - 1

Step 4: Why we cannot do fewer comparisons

In a comparison-based model, each element must be compared enough times to establish its order relative to elements from the other list.

Consider the last element in the merged list:

It must be compared (directly or indirectly) with all elements of the other list to ensure it is indeed the largest.

Therefore, no algorithm can guarantee fewer than 2n - 1 comparisons in the worst case.