1. Consider the following restricted variant of the Tower of Hanoi puzzle. The pegs are numbered 0, 1, and 2, and your task is to move a stack of \( n \) disks from peg 1 to peg 2. However, you are forbidden to move any disk directly between peg 1 and peg 2; every move must involve peg 0.

Describe an algorithm to solve this version of the puzzle in as few moves as possible. Exactly how many moves does your algorithm make?

Solution: The following recursive algorithm moves the top \( n \) disks from the source peg \( \text{src} \) (either 1 or 2) to the destination peg \( \text{dst} \) (either 1 or 2), where every move uses peg 0. (The forbidden peg never changes, so we can hard-code it into the algorithm.)

\[
\text{FORBIDDENHANOI}(n, \text{src}, \text{dst}):
\begin{align*}
\text{if } n > 0 \\
&\text{FORBIDDENHANOI}(n - 1, \text{src}, \text{dst}) \\
&\text{move disk } n \text{ from peg } \text{src} \text{ to peg } 0 \\
&\text{FORBIDDENHANOI}(n - 1, \text{dst}, \text{src}) \\
&\text{move disk } n \text{ from peg } 0 \text{ to peg } \text{dst} \\
&\text{FORBIDDENHANOI}(n - 1, \text{src}, \text{dst})
\end{align*}
\]

The initial call is \( \text{FORBIDDENHANOI}(n, 1, 2) \).

The number of moves satisfies the recurrence \( T(n) = 3T(n - 1) + 2 \). We can easily verify by induction that \( T(n) = 3^n - 1 \).

Here is a complete proof of correctness. Let \( n \) be an arbitrary non-negative integer, and assume that \( \text{FORBIDDENHANOI}(m, a, b) \) correctly moves \( m \) disks from \( a \) to \( b \) for every non-negative integer \( m < n \). If \( n = 0 \) the algorithm correctly does nothing, so assume \( n > 0 \).

By the inductive hypothesis, the first recursive call correctly moves the top \( n - 1 \) disks from \( \text{src} \) to \( \text{dst} \). The second line moves disk \( n \) from \( \text{src} \) to peg 0, which is legal because all smaller disks are on peg \( \text{dst} \). By the inductive hypothesis, the second recursive call correctly moves the top \( n - 1 \) disks from \( \text{dst} \) to \( \text{src} \). The fourth line moves disk \( n \) from peg 0 to peg \( \text{dst} \) which is legal because all smaller disks are on peg \( \text{src} \). Finally, by the inductive hypothesis, the first recursive call correctly moves the top \( n - 1 \) disks from \( \text{src} \) to \( \text{dst} \).

Put more simply: The algorithm is correct because every line is correct (by the inductive hypothesis) and the lines are in the right order. Put even more simply: The algorithm is obviously correct.

\[
\text{Rubric: } 10 \text{ points} =
\begin{itemize}
  \item 2 for an English specification
    \begin{itemize}
      \item No credit for the rest of the problem if this is missing. — See Cardinal Sin #2: “Declare all your variables”
      \item A specification is a description of what problem the algorithm is supposed to solve, not how the algorithm solves that problem.
      \item The specification must actually use the input parameters of the algorithm.
    \end{itemize}
  \item 1 for specifying the initial function call
  \item 5 for a correct algorithm. This is not the only correct description.
  \item 2 for time analysis (1 for “}O(3^n)“)
  \item No proof of correctness required.
\end{itemize}
\]
2. Consider the following cruel and unusual sorting algorithm.

\[
\text{Cruel}(A[1..n]):
\begin{align*}
\text{if } n > 1 & \quad \text{Cruel}(A[1..n/2]) \\
& \quad \text{Cruel}(A[n/2 + 1..n]) \\
& \quad \text{Unusual}(A[1..n])
\end{align*}
\]

\[
\text{Unusual}(A[1..n]):
\begin{align*}
\text{if } n = 2 & \quad \text{if } A[1] > A[2] \quad \langle \text{the only comparison!} \rangle \\
& \quad \text{else} \\
& \quad \text{for } i \leftarrow 1 \text{ to } n/4 \quad \langle \text{swap 2nd and 3rd quarters} \rangle \\
& \quad \quad \text{swap } A[i + n/4] \leftrightarrow A[i + n/2] \\
& \quad \text{Unusual}(A[1..n/2]) \quad \langle \text{recurse on left half} \rangle \\
& \quad \text{Unusual}(A[n/2 + 1..n]) \quad \langle \text{recurse on right half} \rangle \\
& \quad \text{Unusual}(A[n/4 + 1..3n/4]) \quad \langle \text{recurse on middle half} \rangle
\end{align*}
\]

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called oblivious. Assume for this problem that the input size \(n\) is always a power of 2.

(a) Prove by induction that Cruel correctly sorts any input array.

**Solution:** The only difference between Cruel and Mergesort is that Cruel calls Unusual instead of Merge. So to prove that Cruel correctly sorts, it suffices to prove that Unusual behaves exactly like Merge.

**Lemma 1.** For any array \(A[1..n]\) such that \(n\) is a power of 2, the subarray \(A[1..n/2]\) is sorted, and the subarray \(A[n/2 + 1..n]\) is sorted, calling Unusual\((A[1..n])\) correctly sorts the entire array \(A[1..n]\).

**Proof:** Let \(A[1..n]\) be an arbitrary array such that \(n\) is a power of 2, and the two subarrays \(A[1..n/2]\) and \(A[n/2 + 1..n]\) are each sorted. Assume that Unusual correctly sorts any array whose length is a power of 2 smaller than \(n\) and whose first and second halves are sorted.

To simplify notation, we name the four quarters of \(A\) as follows:

\[
\begin{align*}
A_1 & := A[1..n/4], \\
A_2 & := A[n/4 + 1..n/2], \\
A_3 & := A[n/2 + 1..3n/4], \\
A_4 & := A[3n/4 + 1..n].
\end{align*}
\]

These names refer to the array addresses, not the array contents. By assumption, the subarrays \(A_1 \cup A_2\) and \(A_3 \cup A_4\) are initially sorted. We separately track the four quartiles of \(A\) through the execution of Unusual\((A)\). There are three cases to consider.

- **First, consider the smallest \(n/4\) elements of \(A\).**
  - The smallest \(n/4\) elements of \(A\) all initially lie in the subarrays \(A_1\) and \(A_3\).
  - After the for-loop swaps \(A_2\) and \(A_3\), the smallest \(n/4\) elements of \(A\) lie in \(A_1 \cup A_2\). Moreover, the subarrays \(A_1\) and \(A_2\) are still sorted.
– The inductive hypothesis implies that the first recursive call to Unusual sorts $A_1 \cup A_2$. Thus, after this call, $A_1$ contains the $n/4$ smallest elements of $A$ in sorted order.
– The rest of Unusual does not modify $A_1$.

• Next, consider the largest $n/4$ elements of $A$.
  – The largest $n/4$ elements all initially lie in the subarrays $A_2$ and $A_4$.
  – After the for-loop swaps $A_2$ and $A_3$, the largest $n/4$ elements of $A$ lie in $A_3 \cup A_4$. Moreover, the subarrays $A_3$ and $A_4$ are still sorted.
  – The first recursive call to Unusual does not modify $A_3$ or $A_4$.
  – The inductive hypothesis implies that the second recursive call to Unusual sorts $A_3 \cup A_4$. Thus, after this call, $A_4$ contains the $n/4$ largest elements of $A$ in sorted order.
  – The rest of Unusual does not modify $A_4$.

• Finally, consider the $(n/4 + 1)$th through $(3n/4)$th smallest elements of $A$, which we call the middle elements.
  – The first recursive call to Unusual moves all middle elements out of $A_1$.
  – The second recursive call to Unusual moves all middle elements out of $A_4$.
  – At this point, all middle elements lie in the subarrays $A_2$ and $A_3$, but possibly in the wrong order. However, $A_2$ is sorted and $A_3$ is sorted.
  – The inductive hypothesis implies that the third recursive call to Unusual sorts $A_2 \cup A_3$, putting each middle element into its correct location.

We conclude that Unusual correctly sorts the entire array $A[1..n]$. $\square$

Cruel’s correctness now follows immediately from the correctness of MergeSort. ■

Rubric: 6 points =
+ 5 for proving Unusual = Merge (standard induction rubric)
+ 1 for proving Cruel = Mergesort

(b) Prove that Cruel would not correctly sort if we removed the for-loop from Unusual.

Solution: With this modification, Cruel([3, 4, 1, 2]) returns the array [3, 1, 4, 2]. ■

(c) Prove that Cruel would not correctly sort if we swapped the last two lines of Unusual.

Solution: With this modification, Cruel([3, 4, 1, 2]) returns the array [1, 3, 2, 4]. ■

(d) What is the running time of Unusual? Justify your answer.

Solution: The running time of Unusual satisfies the recurrence $T(n) = 3T(n/2) + O(n)$. The recursion tree technique (with geometrically increasing level sums)—or the analysis of Karatsuba’s algorithm in the lecture notes—gives us the solution $T(n) = O(n \log^3 n)$. ■

(e) What is the running time of Cruel? Justify your answer.

Solution: The analysis in part (d) implies that the running time of Cruel satisfies the recurrence $T(n) = 2T(n/2) + O(n \log^3 n)$. The recursion tree technique (with geometrically decreasing level sums) gives us the solution $T(n) = O(n \log^3 n)$. ■

Rubric: 1 point each for parts (b)–(e).
3. You are a visitor at a political convention (or perhaps a faculty meeting) with \( n \) delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to. In particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the same party or not simply by introducing them to each other. Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

(a) Suppose more than half of the delegates belong to the same political party. Describe and analyze an efficient algorithm that identifies every member of this majority party.

**Solution (divide and conquer; 6/7):** I’ll describe a recursive algorithm to identify one member of the majority party. After this algorithm runs, we can identify all members of the majority party in \( O(n) \) additional time, by introducing the chosen representative to everyone else.

1. Split the delegates roughly in half. Send one half to the left side of the room, the other half to the right side of the room.
2. Recursively find a member of the majority party for the delegates on the left. Call her Elle.
3. Recursively find a member of the majority party for the delegates on the right. Call him Ari.
4. Introduce Elle and Ari to everyone (including each other) in \( O(n) \) time.
5. Whoever gets more friendly handshakes is a member of the majority party.

Here’s the same algorithm in pseudocode:

```plaintext
ONEINMAJORITY(D[1..n]):
if n = 1
    return D[1]
ℓ ← ONEINMAJORITY(D[1..[n/2]])
r ← ONEINMAJORITY(D[[n/2] + 1..n])
ℓcount ← 0; rcount ← 0
for i ← 1 to n
    if SAMEPARTY(ℓ, D[i])
        ℓcount ← ℓcount + 1
    if SAMEPARTY(r, D[i])
        rcount ← rcount + 1
if ℓcount > rcount
    return ℓ
else
    return r
```

To prove this algorithm correct, we must verify two claims:

**Lemma 2.** \( \text{ONEINMAJORITY} \) always returns an element of its input array, even if there is no majority.

**Proof:** If \( n = 1 \), the algorithm returns \( D[1] \). Otherwise, the algorithm always returns either \( ℓ \) or \( r \); the inductive hypothesis implies that \( ℓ \) is an element of \( D[1..[n/2]] \) and \( r \) is an element of \( D[[n/2] + 1..n] \). \( \square \)

**Lemma 3.** \( \text{ONEINMAJORITY} \) returns a member of the majority party if there is one.
**Proof:** Assume there is a majority party. That party must contain more than half of the first \(\lceil n/2 \rceil\) delegates, or more than half of the last \(\lfloor n/2 \rfloor\) delegates, or possibly both. Thus, the inductive hypothesis implies that either \(\ell\) or \(r\) is a member of the majority party. The previous lemma implies that both \(\ell\) and \(r\) are delegates, even if their half of the delegates has no majority party, so their parties are always well-defined. The for-loop counts how many people are in \(\ell\)'s party and in \(r\)'s party and returns whichever party is larger.

The running time of this algorithm obeys the mergesort recurrence \(T(n) = 2T(n/2) + O(n)\), so the algorithm runs in \(O(n \log n)\) time.

**Rubric:** 6 points = 1 for base case + 2 for recursive case + 2 for proof of correctness
(= \(1/2\) for Lemma 2 + 1\(1/2\) for Lemma 3) + 1 for time analysis.

**Solution (divide and conquer, 7/7):** I’ll describe a recursive algorithm to identify one member of the majority party. After this algorithm runs, we can identify all members of the majority party in \(O(n)\) additional time, by introducing the chosen representative to everyone else.

(1) Pair up the delegates, and introduce each pair to each other.
   (a) If a pair hate each other, remind them of the first rule of Fight Club and kick them out.
   (b) If a pair like each other, invite one of the pair to the VIP room and send the other one home.

(2) If \(n\) is odd, invite the last delegate the to VIP room.

(3) Recursively find a member of the majority party for the VIPs.

We prove this algorithm correct by induction, paying careful attention to the case where \(n\) is odd. There are two cases to consider.

- The algorithm is trivially correct when \(n = 1\).
- So suppose \(n > 1\). Again, let \(m\) denote the number of majority delegates, and let \(\ell\) be the number of majority delegates invited to the VIP room. Then exactly \(m - \ell\) majority delegates are not invited to the VIP room; of those, at most \(\ell\) were paired with other majority delegates.\(^1\) So at least \(m - 2\ell\) majority delegates were paired with other majority delegates.

\(^1\)When \(n\) is odd and \(D[n]\) is a majority element, only \(\ell - 1\) majority delegates are paired with other majority delegates.
paired with minority delegates. Thus, at least \( m - 2\ell \) minority delegates were paired with majority candidates, leaving at most \( (n - m) - (m - 2\ell) = n - 2m + 2\ell \) remaining minority delegates, so at most \( n/2 - m + \ell \) minority delegates are VIPs. Since \( m > n/2 \), the number of minority VIPs is strictly less than \( \ell \). So the induction hypothesis implies that the recursive call correctly returns a majority delegate.

The running time obeys the recurrence \( T(n) \leq O(n) + T(\lceil n/2 \rceil) \), because at most half the delegates become VIPs. Ignoring the ceiling in the recursive argument, we get a descending geometric series, so overall the algorithm runs in \( O(n) \) time.\(^2\)  

**Rubric:** 7 points = 1 for base case + 2 for recursive pairing + 1 for correctly dealing with odd \( n \) + 2 for proof of correctness + 1 for time analysis.

**Solution (iterative, 7/7):** I’ll describe a simple iterative algorithm to identify one member of the majority party in \( O(n) \) time; this algorithm was discovered by Robert S. Boyer and J Strother Moore in 1980.\(^3\) After this algorithm runs, we can identify all members of the majority party in \( O(n) \) additional time, by introducing the chosen representative to everyone else.

\[
\text{BoyerMooreVote}(D[1..n]):
\begin{align*}
\text{count} &\leftarrow 0 \\
\text{for } i &\leftarrow 1 \text{ to } n \\
&\quad \text{if } \text{count} = 0 \\
&\quad \quad \text{winner} \leftarrow i \\
&\quad \quad \text{count} \leftarrow i \\
&\quad \quad \text{else if } \text{SAMEParty}(\text{winner}, i) \\
&\quad \quad \quad \text{count} \leftarrow \text{count} + 1 \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{count} \leftarrow \text{count} - 1 \\
\text{return } \text{winner}
\end{align*}
\]

We can prove this algorithm correct as follows. At the end of the \( i \)th iteration of the loop, the first \( i \) delegates can be partitioned into two (possibly empty) subsets:

- Exactly \( \text{count} \) members of \( \text{winner} \)'s party, and
- Exactly \( (i - \text{count})/2 \) pairs of delegates, where the delegates in each pair are in two different parties.

In particular, among the first \( i \) delegates, at most \( (i - \text{count})/2 \) are not in \( \text{winner} \)'s party. Thus, when the algorithm ends, at most \( (n - \text{count})/2 \leq n/2 \) delegates are not in \( \text{winner} \)'s party. Since some party has strictly more than \( n/2 \) members (by assumption), that must be the final \( \text{winner} \)'s party.  

**Rubric:** 7 points = 3 for correct algorithm + 3 for proof of correctness + 1 for time analysis.

---

\(^2\)In fact, the number of introductions is exactly \( n - \#1(n) \), where \( \#1(n) \) is the number of 1s in the binary representation of \( n \). This is the best possible.

\(^3\)No, I’m not missing a period; J was Moore’s complete first name. Single-letter names are more common than most people think; Jeff’s wife’s middle name is the letter H (no period). On the other hand, Jeff’s dad’s middle name is Jay.
Rubric (general for part (a)): These are not the only correct solutions, and these solutions are more detailed than necessary for full credit. Partial credit for slower/more limited solutions:

- 4 points for any algorithm that is correct and runs in $O(n)$ time when there are exactly two parties.
- 2 points for any correct $O(n^2)$-time (or slower) algorithm. Beware that many algorithms run in quadratic time when the first $n/4$ delegates belong to distinct parties.

(b) Now suppose precisely $p$ political parties are present and one party has a plurality: more delegates belong to that party than to any other party. Please present a procedure to pick out the people from the plurality party as parsimoniously as possible. Do not assume that $p = O(1)$.

Solution: The following algorithm finds one member of the plurality party in $O(np)$ time by brute force. We repeatedly find a delegate that is not already assigned to a party, and introduce them to everyone. After $p$ passes, we know the party of every delegate.

\[
\text{OneInPlurality}(D[1..n]):
\begin{align*}
&\text{for } i \leftarrow 1 \text{ to } n \\
&\quad \text{Party}[i] \leftarrow 0 \\
&\quad \text{maxCount} \leftarrow 0 \\
&\text{for } j \leftarrow 1 \text{ to } p \\
&\quad \text{count} \leftarrow 0 \\
&\quad \text{for } i \leftarrow 1 \text{ to } n \\
&\quad \quad \text{if } \text{Party}[i] \neq 0 \\
&\quad \quad \quad \langle \langle \text{do nothing} \rangle \rangle \\
&\quad \quad \text{else if } \text{Count}[j] = 0 \\
&\quad \quad \quad \text{Party}[i] \leftarrow j \\
&\quad \quad \quad \text{rep} \leftarrow i \\
&\quad \quad \quad \text{count} \leftarrow \text{count} + 1 \\
&\quad \quad \text{else if } \text{SAMEPARTY}(i, \text{rep}) \\
&\quad \quad \quad \text{Party}[i] \leftarrow j \\
&\quad \quad \quad \text{count} \leftarrow \text{count} + 1 \\
&\quad \quad \text{if } \text{maxCount} < \text{count} \\
&\quad \quad \quad \text{maxCount} \leftarrow \text{count} \\
&\quad \quad \quad \text{maxRep} \leftarrow \text{rep} \\
&\quad \text{return } \text{maxRep}
\end{align*}
\]

We can find all members of the plurality party in $O(n)$ additional time, by introducing the plurality representative to everyone else (or by more careful bookkeeping).

Rubric: 3 points = 2 for algorithm + 1 for running time. This is not the only correct solution, but it is the best possible running time. In particular, when $p = n - 1$, we must introduce every pair of delegates in the worst case to find the single pair that is in the same party.