13 (100 PTS.) That's so Hanoi-ing.

Consider the following variants of the Towers of Hanoi. For each of variant, describe an algorithm to solve it in as few moves as possible. **Prove** that your algorithm is correct. Initially, all the n disks are on peg 1, and you need to move the disks to peg 2. In all the following variants, you are not allowed to put a bigger disk on top of a smaller disk.

13.A. (30 PTS.) Hanoi 0: Suppose you are forbidden to move any disk directly between peg 1 and peg 2, and every move must involve (the third peg) 0. Exactly (i.e., not asymptotically) how many moves does your algorithm make as a function of n?

Solution:

The following recursive algorithm moves the top n disks from the source peg s (either 1 or 2) to the destination peg d (either 1 or 2), where every move uses peg 0. (The forbidden peg never changes, so we can hard-code it into the algorithm.)

```
Hanoi_a(n, s, d):
  if n = 0 then return

  Hanoi_a(n − 1, s, d)
  move disk n from peg s to peg 0
  Hanoi_a(n − 1, d, s)
  move disk n from peg 0 to peg d
  Hanoi_a(n − 1, s, d)
```

The initial call is $Hanoi_a(n, 1, 2)$.

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

As for proof of correctness, the trick is to be concise.

Proof: The procedure is clearly correct for $n = 1$. Assume it works correctly for $n < k$, and consider the case $n = k$. The first recursive calls succeeds by induction, and we only have to worry about the moves being legal. Moving disk n to from s to 0 succeeds, since all the first $n-1$ disks are on peg $d \neq 0$. Similar argumentation implies that the other recursive call are successful, and the move from peg 0 to peg d of n is valid (as all the first $n-1$ disks are on peg s at this point in time. As such, by induction, all the moves performed by the algorithm are legal, and clearly the first n disks end up on peg d, which implies the claim. ■

13.B. (30 PTS.) Hanoi 2: Suppose you are only allowed to move disks from peg 0 to peg 1, from peg 1 to peg 2, or from peg 2 to peg 0.

Provide an upper bound, as tight as possible, on the number of moves that your algorithm uses.

(One can derive the exact upper bound by solving the recurrence, but this is too tedious and not required here.)

Solution:

Here $[\![n]\!] = \{1, 2, \ldots, n\}$. We can arrange the pegs in a circle. In order to move the disks one peg clockwise, you need to move the top $n-1$ discs two pegs clockwise, move the n^{th} disc one peg clockwise, then move the $n-1$ discs two pegs clockwise again. Hence, we can formulate the following two-part algorithm:

```
Move+1(n, s):
    if n = 0 then return
    Move+2(n − 1, s) // Disks [[n − 1]] are on peg s + 2
    move disk n from peg s to peg (s + 1) % 3      (L1)
    Move+2(n-1, (s + 2) % 3)

Move+2(n, s):
    if n = 0 then return
    Move+2(n − 1, s) // Disks [[n − 1]] are on peg s + 2
    move disk n from peg s to peg (s + 1) % 3
    Move+1(n − 1, (s + 2) % 3) // Disks [[n − 1]] are on peg s
    move disk n from peg (s + 1) % 3 to peg (s + 2)%3
    Move+2(n − 1, s) // Disks [[n]] are on peg (s + 2) % 3
```

The initial call is Move+1$(n, 1)$ which will move the first n disks from peg 1 to peg 2.

Proof: (of correctness) We prove by induction on n that all the moves performed by both procedures are legal. The claim is immediate for $n = 1$. So assume both procedures works correctly for $n < k$, and consider the case $n = k$.

Arguing as above, when Move+1(n, s) is being called, when reaching (L1), all the first $n-1$ disks are on peg $(s+2)$ % 3, as all moves performed by the call Move+2$(n-1, s)$ are valid by induction. As such, one can move disk n from peg s to peg $(s+1)$ % 3 safely. The rest of moves in the second recursive calls are also valid by induction. Which implies that Move+1(n, s) performs only legal moves.

A similar inductive argument shows the correctness of Move+2(n, s).

Now, since all the disk moves performed by the algorithm are valid, it easy to verify that Move+1$(n, 1)$ indeed moves the n disks from peg 1 to peg 2, as desired. ∎

Bounding the number of moves. Let $T_1(n)$ and $T_2(n)$ be the number of moves performed by Move+1$(n, \cdot)$ and Move+2$(n, \cdot)$, respectively.

We have the following two recurrence relations:

$$T_1(n) = 2T_2(n-1) + 1$$
$$T_2(n) = 2T_2(n-1) + T_1(n-1) + 2$$

Substituting the first equation into the second yields:

$$T_2(n) = 2T_2(n-1) + 2T_2(n-2) + 3.$$

The solution to this recurrence is

$$T_2(n) = \frac{(1+\sqrt{3})^{n+2} - (1-\sqrt{3})^{n+2}}{4\sqrt{3}} - 1 \leq (1+\sqrt{3})^n \leq 2.733^n.$$

This means $T_1(n) = \frac{(1+\sqrt{3})^{n+1} - (1-\sqrt{3})^{n+1}}{2\sqrt{3}} \leq 2.733^n$.

13.C. (40 PTS.) Hanoi Bye Bye: Finally consider the disappearing Tower of Hanoi puzzle where the largest remaining disk will disappear if there is nothing on top of it. The goal here is to get all the disks to disappear and be left with three empty pegs (in as few moves as possible). Provide an upper bound, as tight as possible, on the number of moves your algorithm uses.

Solution:

The intuition for this puzzle is to move $n-2$ discs to another peg and then move the $n-1$ disk to the third empty peg. At this point the two largest disks will be on separate pegs without any disks on top of them and thus, disappear. You can use the classic Hanoi algorithm to move the first $n-2$ disks and use the algorithm below to complete the puzzle:

```
Hanoi_c(n, s, d):
  if n ≤ 1 then
    return
  t ← peg that is not s or d
  HanoiReg(n − 2, s, t)
  Move disk n − 1 from s to d
  //disks n and n − 1 disappear
  Hanoi_c(n − 2, t, d)
```

The initial call is $Hanoi_c(n, 1, 2)$. Here $HanoiReg(n, \cdot, \cdot)$ is the regular Hanoi algorithm which takes $2^n - 1$ moves.

Proof: (of correctness) Follows readily by using the same argumentation as above. ■

Knowing that the standard Hanoi function takes $2^n - 1$ moves, the recurrence for the number of moves performed by the algorithm is

$$T(n) = 2^{n-2} - 1 + T(n-2).$$

With $T(1) = 0$, $T(2) = 1$ and $T(3) = 2$. By repeated opening, we have that

$$T(n) = 2^{n-2} + 2^{n-4} - 2 + T(n-4) = \sum_{j=1}^{i} 2^{n-2j} - i + T(n-2i).$$

If $n = 2k+1$, then

$$T(n) = \sum_{j=1}^{k} 2^{n-2j} - k + T(n-2k) = 2(1+4+\cdots 4^{(k-1)}) - k + T(1) = \frac{2(4^k-1)}{3} - k$$
$$= \frac{2(4^{\lfloor n/2 \rfloor}-1)}{3} - \lfloor n/2 \rfloor \leq \frac{2^n}{3}.$$

If $n = 2k$, then

$$\begin{aligned} T(n) &= \sum_{j=1}^{k-1} 2^{n-2j} - k + 1 + T(n - 2(k-1)) \\ &= 4 + 4^2 + \cdots + 4^{(k-1)} - k + 1 + T(2) = \frac{4(4^{k-1} - 1)}{3} - k + 2 \\ &= \frac{4^{\lfloor n/2 \rfloor} - 4}{3} - \lfloor n/2 \rfloor + 2 \leq \frac{2^n}{3}. \end{aligned}$$

14 (100 PTS.) Divide and Merger

Suppose you are given k sorted arrays $A_1, A_2, \ldots, A_k$ (potentially of different sizes). Let $n_i > 0$ be the size of the ith array A_i, for $i = 1, \ldots, k$, with $\sum_{i=1}^{k} n_i = n$. Assume that all the numbers in all the arrays are distinct. You would like to merge them into a single sorted array A of n elements.

14.A. (30 PTS.) Use a divide and conquer strategy to derive an algorithm that sorts the given sorted arrays in $O(n \log k)$ time, into one big happy sorted array.

Solution:

We will divide the problem of merging k sorted arrays $A_1, \ldots, A_k$, of total size n, as follows.

- Merge $\lfloor k/2 \rfloor$ sorted arrays $A_1, \ldots A_{\lfloor k/2 \rfloor}$ into a single sorted array B_1.
- Merge $\lceil k/2 \rceil$ sorted arrays $A_{\lfloor k/2 \rfloor + 1}, \ldots, A_k$ into a single sorted array B_2.

We can recursively solve the above two problems and merge B_1 and B_2 into a single sorted array using the **merge** subroutine from **MergeSort**.

```
MergeMultipleArrays( A_1, ..., A_k ):
  if k = 1 then
    return A_1
  B_1 ← MergeMultipleArrays(A_1, ..., A_⌊k/2⌋ )
  B_2 ← MergeMultipleArrays(A_⌊k/2⌋+1, ..., A_k )
  return merge(B_1, B_2)
```

Running time. It is easy to verify that the running time of the algorithm is proportional to the time spend inside the **merge** procedure, so we bound this quantity. The depth of the recursion of this algorithm is $O(\log k)$. Each element participates in $O(\log k)$ merges, and it pays for its participation. As such, the overall running time is $O(n \log k)$.

Correctness. We use induction on k. Let k be an arbitrary integer ≥ 1. Let $A_1, \ldots, A_k$ be k arbitrary sorted arrays (with the assumption that all numbers in the arrays are distinct), each of size n. We wish to show that **MergeMultipleArrays**, on input $A_1, \ldots, A_k$, merges them into a single sorted array A of kn elements.

For the base case, we have $k = 1$. In this case A_1 is already sorted and **MergeMultipleArrays** simply returns the single array A_1.

For the inductive step, assume that **MergeMultipleArrays** correctly merges ℓ sorted arrays, for every $\ell < k$, into a single sorted array of size ℓn. From the inductive hypothesis,

it follows that B_1 is a sorted array of the first $\lfloor k/2 \rfloor$ input arrays, and B_2 is a sorted array of the last $\lceil k/2 \rceil$ input arrays. Since MERGE correctly merges the two arrays into a single sorted array, we conclude that **MergeMultipleArrays** correctly merges the k sorted arrays into a single sorted array.

14.B. (30 PTS.) In **MergeSort** we split the array of size N into two arrays each of size $N/2$, recursively sort them and merge the two sorted arrays. Suppose we instead split the array of size N into k arrays of size N/k each and use the merging algorithm in the preceding step to combine them into a sorted array. Describe the algorithm formally and analyze its running time via a recurrence. You do not need to prove the correctness of the recursive algorithm.

Solution:

he algorithm is as given below. We split the array of size N into k arrays of size $\lceil N/k \rceil$. Note that the kth array is dealt outside the for loop since $k \cdot \lceil \frac{N}{k} \rceil$ can be larger than N. Note also that each array B_i is of size $\lceil \frac{N}{K} \rceil$, except B_k. This can be easily fixed by appending large numbers at the end of B_k. We have skipped over this detail to keep the algorithm brief.

```
NewMergeSort ( A[1,...,N] ):
  if N = 1
     return A
  for i ← 1 to k − 1
     j ← (i − 1) · ⌈N/K⌉
     B_i ← NewMergeSort (A[j + 1,...,j + ⌈N/K⌉])
  B_k ← NewMergeSort (A[(k − 1) · ⌈N/K⌉ + 1,...,N])
  return MergeMultipleArrays (B_1,...,B_k)
```

At the base case, we have $T(N) = O(1)$ for $N = 1$. At each step, it takes $O(1)$ to set up each recurrence There are a total of k recurrences, so it takes a total of $O(k)$ time to set them all up[1]. Finally, it takes $O(N \log k)$ time to run the **MergeMultipleArrays** routine (since $n = N/k$). This eclipses the $O(k)$ time taken to set up the recurrences (since $N > k$). Thus, the recurrence is given by

$$T(N) = \begin{cases} O(1) & \text{if } N = 1, \\ kT(\frac{N}{k}) + O(N \log k) & \text{otherwise.} \end{cases}$$

To solve the recurrence relation, note that at level i in the recurrence tree there are a total of k^i nodes. Each node represents a problem of size N/k^i. So the total work done at level i of the recurrence tree is $O(k^i \frac{N}{k^i} \cdot \log k) = O(N \log k)$. Since there are $\log_k N$ levels, the total work done (at the non-leaf nodes) is given by

$$\begin{aligned} \sum_{i=0}^{\log_k N - 1} O(N \cdot \log k) &= O(N \cdot \log_k N \cdot \log k) \\ &= O(N \cdot \frac{\log N}{\log k} \cdot \log k) \\ &= O(N \log N). \end{aligned}$$

Since there are a total of $O(k^{\log_k N}) = O(N)$ leaves, the total work done at leaves is $O(N)$. Thus, we conclude that the NEWMERGESORT algorithm runs in $O(N \log N)$ time, which is no better (asymptotically) than the regular merge sort.

To show the correctness of the algorithm, we will use induction on N. Let N be an arbitrary integer ≥ 1. We wish to show that **NewMergeSort**, on input an unsorted array A, sorts A.

For the base case, we have $N = 1$. In this case A is already sorted and **NewMergeSort** simply returns A.

For the inductive step, assume that **NewMergeSort** correctly sorts any arbitrary input array of size $\ell < N$. From the inductive hypothesis, it follows that each B_i, for $i \in [1, k]$, is a sorted array of size $\lceil N/k \rceil$. Since **MergeMultipleArrays** correctly merges the k sorted arrays (from the previous part), we conclude that NEWMERGESORT correctly sorts A.

14.C. (40 PTS.) Describe an algorithm (not necessarily divide and conquer) for the settings of **(14.A.)** that works in $O(\sum_{i=1}^{k} n_i \log \frac{n}{n_i})$ time. Prove the correctness of your algorithm, Note that this is potentially an improved algorithm if the n_is are non-uniform. For example, if $n_i = n/2^i$, for $i = 1, \ldots, k$, then the overall running time is linear. One can verify (but you do not need to do it – it is not immediate) that $O(\sum_{i=1}^{k} n_i \log \frac{n}{n_i}) = O(n \log k)$. This implies that this algorithm is a strict improvement over **(14.A.)**.

Solution:

For an array of size t, set its rank $r(t) = \lfloor \log_2 t \rfloor$. The algorithm would repeatedly merge the two sorted arrays of lowest rank into a new sorted array (resolving ties arbitrarily), until remaining with a single array.

Consider the rank of an element as the rank of the array it s currently in. If a rank of an element does not change during a merge, than it was in the second smallest array in the current collection, merged with an array with a strictly smaller rank. The next merge for element, would involve another array that would be of either the same rank, or bigger rank. In either case, the rank of the element increases. Namely, at least every two times an element participates in a merge, its rank increases. A rank that stars with rank $r(n_i)$, an rise to rank at most $\log_2 n$. Namely, an element of rank $r(n_i)$ can participate in at most $O(\log n - \log n_i) = O(\log(n/n_i))$ merges. Thus, $O(\sum_i n_i \log(n/n_i))$ bounds the total running time of the algorithm.

To implement things efficiently, we precompute an array of size $O(\log n)$, where the ith entry contains a linked list of all current arrays of rank r_i. It is easy to verify that finding the two elements of smallest rank, and moving them to the new rank, can be done in $O(1)$ time (observing that the ranks are monotonically increasing). As such, this management of the arrays costs $O(k)$ time overall, which is dominated by the other terms.

As for the last claim (which you did not have to do), let $p_i = n_i/n$. We have that

$$O\Big(\sum_{i=1}^{k} n_i \log \frac{n}{n_i}\Big) = O\Big(n\big(-\sum_{i=1}^{k} p_i \lg p_i,\big)\Big).$$

where $\lg = \log_2$. Here $p_i \in (0, 1)$, and $\sum_i p_i = 1$. The summation $-\sum_{i=1}^{k} p_i \lg_2 p_i$ is known as the entropy function, and it known to be maximized when $p_i = 1/k$, which

readily implies the claim. Proving the later claim requires some tedious but straightforward calculus, which we omit.

15 (100 PTS.) Fowl business.

You were given a kettle of n birds, which look all the same to you. To decide if two birds are of the same species, you perform the following experiment – you put the two of them in a cage together. If they are friendly to each other, then they are of the same species. Otherwise, you separate them quickly before survival of the fittest kicks in.

15.A. (60 PTS.) Suppose that strictly more than half of the birds belong to the same species. Describe and analyze an efficient algorithm that identifies every bird among the n birds that belong to this dominant species.

Solution:

(The solution that was expected.)

I'll describe a recursive algorithm to identify *one* member of the majority species. After this algorithm runs, we can identify *all* members of the majority species in $O(n)$ additional time, by introducing the chosen representative to everyone else.

(A) Split the birds roughly in half. Send one half to the left side of the room, the other half to the right side of the room.
(B) Recursively find a member of the majority species for the birds on the left. Call her Elle.
(C) Recursively find a member of the majority species for the birds on the right. Call him Ari.
(D) Introduce Elle and Ari to everyone (including each other) in $O(n)$ time.
(E) Whoever gets more friendly interactions is a member of the majority species.

Here's the same algorithm in pseudocode:

```
OneInMajority ( D[1,...,n] ):
  if n == 1
    return D[1]
  l ← OneInMajority ( D[1,...,n/2] )
  r ← OneInMajority ( D[n/2+1,...,n] )
  l_count ← 0
  r_count ← 0
  for i ← 0 to n
    if SameSpecies (l, D[i])
      l_count ← l_count + 1
    if SameSpecies (r, D[i])
      r_count ← r_count + 1
  if l_count > r_count
    return l
  else
    return r
```

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

Lemma 5.1. 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 $D[1 \,..\, \lceil n/2 \rceil]$ and r is an element of $D[\lceil n/2 \rceil + 1 \,..\, n]$. ■

Lemma 5.2. ONEINMAJORITY *returns a member of the majority species* ***if there is one****.*

Proof: Assume there is a majority species. That species must contain more than half of the first $\lceil n/2 \rceil$ birds, or more than half of the last $\lfloor n/2 \rfloor$ birds, or possibly both. Thus, the inductive hypothesis implies that either ℓ or r is a member of the majority species. The previous lemma implies that both ℓ and r are birds, even if their half of the birds has no majority species, so their parties are always well-defined. The for-loop counts how many people are in ℓ's species and in r's species and returns whichever species is larger. ■

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

Solution:

Here is a simpler but cleverer solution, that requires only $O(n)$ experiments.

If the number of birds is odd, then take one of the birds, compare it to all the other birds. If it is the majority species (i.e., the count of birds of the same type is $> \lfloor n/2 \rfloor$), then we are done. Otherwise, throw this bird away (we now have an event number of birds).

Now, pair the birds up in pairs, and compare their species. If two birds are different, then throw both of them away (bye bye birdies). If two birds are the same, keep one of them and release the other. do this to all the pairs. Now repeat the algorithm recursively on the remaining birds. Till you are left with a single bird. This bird is in the majority species.

If you want to identify the majority species, you can now scan all the birds, and compare it to this bird.

The number of experiments this process performs is

$$T(n) \leq n - 1 + \lfloor n/2 \rfloor + T(n/2),$$

and the solution to this recurrence is $T(n) \leq 3n$, since opening the recurrence we get a geometric series $(3/2)(n+n/2+n/4+\cdots) \leq 3n$. You need to take the final bird and scan all the birds, So overall this would perform $\leq 4n$ experiments.

The correctness is somewhat more fun.

Lemma 5.3. *The above algorithm output is correct.*

Proof: If n is odd, then the algorithm continues "recursively" to $n - 1$ birds, only if this bird was not in the majority species. Throwing it away just makes the majority relatively bigger, which means that the majority species had not changed.

Now consider the pairing stage. If a pair is of different species, then if one of them belong to the majority, and the other to the minority, than removing both of them keep the majority

species in the majority. So it is a valid move. If both of the are in the minority, then throwing both of them away is also a valid move, as the majority just increases.

In the end of this throwing away mixed pairs, we remain only with pure pairs that are of the same species, and importantly, because the above throwing away was valid, the majority species defines the majority of the pure pairs. As such, taking a bird from each such pair, reduces the number of birds by a factor of two, but the new group of birds have the same majority species, and as such, assuming the recursive call returns the right answer, this algorithm would return the majority species. ■

Solution:

There is an even simpler and even more clever solution.

Algorithm. We start with a counter $x = 0$, and no active bird. We now handle the birds one by one $b_1, b_2, \ldots, b_n$. In the ith iteration, we have the ith bird. There are several possibilities:

- If there is no active bird, then we set the active bird to be b_i, and set $x = 1$. We continue to the next iteration.
- If there is an active bird, then we compare its species to b_i.
  If they are the same species, then we increase x by one.
  If they are of different species, then we decrease x by one. If $x = 0$ then we no longer have an active bird.
  The algorithm now continues to the next iteration (keeping the same active bird, if it has one).

Surprisingly, the active bird in the end of the execution of the algorithm belongs to the majority species. Clearly this algorithm performs $n-1$ experiments. If you want to identify all the birds that belong to this majority species, you need another $n-2$ comparisons. So overall $2n-2$ comparisons.

Lemma 5.4. *If there is a species with strictly more than $\lfloor n/2 \rfloor$ birds, then the above algorithm would identify it.*

Proof: Whenever we increase the counter, think about it as pushing the bird into a stack. Whenever we decrease the counter, we pop the bird from the stack, and pair it with the bird that caused this decrease in the counter. In the end of the execution of the algorithm, we would have x birds in the stack, and all other birds would be matched in mixed pairs. As such, it must be that the majority species are the x birds in the stack.

Lets do the last argument more explicitly. Assume $n = 2m$, and we have a majority species that has $m + 1$ birds. Assume the above interpretation of the algorithm created t pairs, and there are x birds on the stack. We know that $2t + x = n = 2m$. The t pairs are mixed, so at most t of them belong to the majority species. Since $t \leq m$, and there $m + 1$ birds in the majority species, it follows that there must be $n + 1 - t \geq 1$ birds of the majority species that are unpaired. Namely – these birds are on the stack. A similar argumentation works if n is odd.

As such, the unique species of the birds in the stack, is just the species of the active bird, which is the majority species. ■

This pretty algorithm is the simplest case of an algorithm for computing a heavy hitter in a stream of elements. One can extend it to handle (B), but we omit the details here.

15.B. (40 PTS.) Now suppose that there are exactly p species present in your kettle of n birds. and one species has a plurality: more birds belong to that species than to any other species. Present a procedure to pick out the birds from the plurality species as efficiently as possible (i.e., minimize the number of experiments you have to do as a function of n and p). Do *not* assume that $p = O(1)$.

Solution:

The following algorithm finds one member of the plurality species, in $O(np)$ time, by brute force. We repeatedly find a bird that is not already assigned to a species, and introduce them to everyone that is left. After p times this happens, we had classified correctly all the birds.

```
OneInPlurality(D[1,...,n]):
  S[1...n] ← 0, c[1...p] ← 0
  k = 0
  for i = 1,...,n do
    if S[i] = 0 then
      k ← k + 1
      S[i] ← k
      for j = i + 1,...,n do
        if SameSpecies(S[i], S[j]) then
          S[j] ← k
          c[k] ← c[k] + 1

  return (max_j c[j], arg max_j c[j])
```