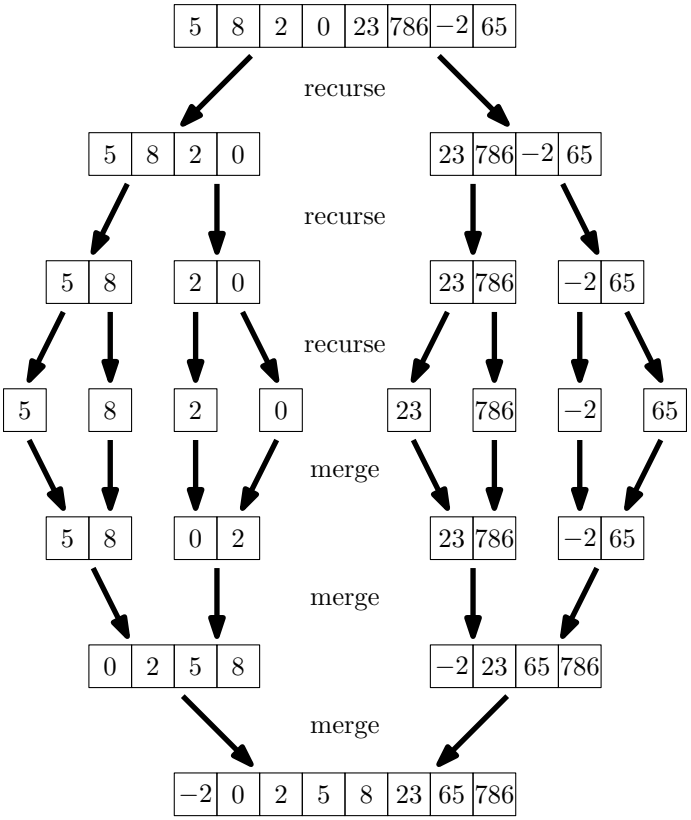
Warm-up

Problem 1. Sort the following array using merge-sort: $A = [5, 8, 2, 0, 23, 786, -2, 65]$. Give all arrays on which recursive calls are made and show how they are merged back together.

Solution 1. Below all arrays are shown. Until all arrays have size 1, all splits represent recursive calls. Since arrays of size 1 are sorted, after this the arrays are merged back together in order to create the final sorted array.



Problem 2. Consider the following algorithm.

```
1: function REVERSE(A)
2:     if |A| = 1 then
3:         return A
4:     else
5:         B ← first half of A
6:         C ← second half of A
7:         return concatenate REVERSE(C) with REVERSE(B)
```

Let $T(n)$ be the running time of the algorithm on an instance of size n. Write down the recurrence relation for $T(n)$ and solve it by unrolling it.

Solution 2. The divide step takes $O(n)$ time when we explicitly copy it. For the recursion, we recurse on two parts of size $n/2$ each. The conquer step concatenates the two arrays, which involves copying over at least one of the two, so this takes $O(n)$ time. Solving a base case when $n = 1$ takes constant time, since we just return the single element array.

Hence, the recursion is:

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n)$$

Problem solving

Problem 3. Given an array A holding n objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than $n/2$ positions of A.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

a) Show that if x is a majority element in the array then x is a majority element in the first half of the array or the second half of the array

b) Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.

c) Put these observation together to design a divide an conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$

d) Solve the recurrence by unrolling it.

Solution 3.

a) If x is a majority element in the original array then, by the pigeon-hole principle, x must be a majority element in at least one of the two halves. Suppose that n is even. If x is a majority element at least $n/2 + 1$ entries equal x. By the pigeon hole principle either the first half or the second half must contain $n/4 + 1$ copies of x, which makes x a majority element within that half.

b) We scan the array counting how many entries equal x. If the count is more than $n/2$ we declare x to be a majority element. The algorithm scans the array and spends $O(1)$ time per element, so $O(n)$ time overall.

c) If the array has a single element, we return that element as the majority element. Otherwise, we break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element".
To argue the correctness, we see that if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call returns, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive calls will identify x (by part a)) and the subsequent check will lead the algorithm to return x.
Regarding time complexity, breaking the main problem into the two subproblems and testing the two candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

d)

$$T(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n).$$

Problem 4. Let A be an array with n distinct numbers. We say that two indices $0 \leq i < j < n$ form an inversion if $A[i] > A[j]$. Modify merge sort so that it computes the number of inversions of A.

Solution 4. We augment the standard merge sort algorithm so that in addition to returning the sorted input array, it also returns the number of inversions. If our array consists of a single element, there can't be any inversions, so we return the array and 0.

Let L be the sorted left half of the input array and R be the sorted right half of the input array. We can count on the recursive calls to count the inversions within L and within R, but we need to modify the merge procedure to count the number of inversions (i, j) where i is in the first half and j is in the second half.

For each element $e \in R$ let $L(e)$ be the number of elements in L that are greater than e. Note that the number of inversions (i, j) where i is in the first half of the input array and j is in the second half of the input array equals $\sum_{e \in R} L(e)$. Furthermore, $L(e)$ can be computed on the fly when e is added to the merged array since elements that remain in L at that time are precisely those elements of L that are greater than e. The pseudocode of this algorithm is given below.

```
1: function MERGE-AND-COUNT(L, R)
2:     result ← new array of length |L| + |R|
```

```
    count ← 0
    l, r ← 0, 0
    while l + r < |result| do
        index ← l + r
        if r ≥ |R| or (l < |L| and L[l] < R[r]) then
            result[index] ← L[l]
            l ← l + 1
        else
            result[index] ← R[r]
            count ← count + |L| − l
            r ← r + 1
    return result, count
```

```
function COUNT-INVERSIONS(A)
    if |A| = 1 then
        return A, 0
    else
        B ← first half of A
        C ← second half of A
        sortedB, countB ← COUNT-INVERSIONS(B)
        sortedC, countC ← COUNT-INVERSIONS(C)
        sortedBC, countBC ← MERGE-AND-COUNT(sortedB, sortedC)
        return sortedBC, countB + countC + countBC
```

We focus on the correctness of computing the inversions, since the correctness of sorting was argued during the lecture. The correctness can be proven using a short inductive argument on n, the size of A: Our base case occurs when $n = 1$, in which case no inversions can occur and thus we indeed return 0.

For our inductive hypothesis, we assume that for all arrays of size k the algorithm computes the number of inversions correctly. We now show that the algorithm computes the number of inversions for $k + 1$ correctly as follows: it splits the array in two halves, each of size at most k, so by our induction hypothesis the number of inversions in them is computed correctly. Thus, if we can show that we correctly compute the number of inversions having one element in L and one in R, the correctness of our algorithm follows for $k + 1$. Fortunately, this is indeed the case, as the number of such inversions is exactly the number of unprocessed element in L (i.e., $|L| - l$) when we put an element of R in the sorted array.

The running time analysis is the same as that for merge sort. We get the following recursion

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to

$$T(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n).$$

Problem 5. Given a sorted array A containing distinct non-negative integers, find the smallest non-negative integer that isn't stored in the array. For simplicity, you can assume there is such an integer, i.e., $A[n-1] > n-1$

Example: $A = [0, 1, 3, 5, 7]$, result: 2

Solution 5. Our approach is very similar to binary search: we consider the middle element, determine which half contains a missing element and recurse on that. So the main question is how to determine whether a half contains a missing element. This can be done by checking if the integer stored in $A[i] = i$. If $A[i] = i$, we know that there can be no missing values in the first i elements, since all integers are distinct and non-negative. If $A[i] > i$, this means that there is some value missing and hence, we can recurse on the first half. Note that because the integers are distinct, $A[i]$ can't be less than i. Our base case is when we have an array of size 1 left, in which case we can output our smallest missing integer.

```
1: function FIND-MISSING-IN-RANGE(A, low, high)
2:     if low = high then
3:         return low
4:     else
5:         mid ← ⌊(low + high)/2⌋
6:         if A[mid] = mid then
7:             return FIND-MISSING-IN-RANGE(A, mid + 1, high)
8:         else
9:             return FIND-MISSING-IN-RANGE(A, low, mid)
```

```
1: function FIND-MISSING(A)
2:     return FIND-MISSING-IN-RANGE(A, 0, n − 1)
```

Correctness follows directly from the fact that the integers are sorted and distinct, as well as that it's given that there actually is a missing integer. To argue that we indeed recurse on the correct half of the array, we note that we maintain the invariant that we recurse on the half that is missing the smallest element (implied by our check above as to whether $A[i] = i$). Since the size of the array that we recurse on decreases with each iteration, this implies that we eventually reach an array of size 1, which thus must contain the smallest missing element.

The running time analysis is the same as that for binary search. The recursion is

$$T(n) = \begin{cases} T(n) = T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to $O(\log n)$.

Problem 6. Design an $O(n)$ time algorithm for the majority problem.

Solution 6. (Sketch) Our base case for a single element stays the same as before. In order to get a linear time algorithm we need a different strategy from the one in the earlier question. Assume for now that n is an even number. First, we pair up

arbitrarily objects and compare the objects in each pair. For each pair, if the objects are different discard both; however, if the objects are the same, keep only one copy. If n happens to be odd, we pick an arbitrary object, test whether it is the majority element; if not, we eliminate it and run the even case routine.

Either way, at most half the elements remain after this filtering step. We keep doing this until only one element remains, at which point we can test in linear time if it is indeed a majority element.

The correctness rests on the fact that if an element is a majority element in the original array, it remains a majority element in the filtered array. (Why?)

Since the filtering can be carried out in linear time and the size of the problem halves after each filtering step, we get the recurrence you get is

$$T(n) = \begin{cases} T(n) = T(n/2) + O(n) & \text{for } n > 2 \\ O(1) & \text{for } n \leq 2 \end{cases}$$

which solves to $T(n) = c \cdot n + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + ... = O(n)$.