Problem 1. (10 points)
We're building a new software project out of existing pieces. Each of these pieces offers a set of functionalities if it's included in the final product and we want to ensure that our final product has all functionalities our pieces have to offer (i.e., at least one piece that has the functionality is part of our final product).

Formally, we are given a non-empty set of pieces P and a collection of sets (each set representing a different functionality we want) $F = \{F_1, ...F_m\}$, such that every $F_i \subseteq P$ and for every piece $p \in P$, there exists a F_i that contains it. We need to find that smallest number of pieces such that we have at least one piece in each F_i.

For example, if $P = \{1,2,3\}$ and $F = \{\{1\},\{2,3\}\}$, returning either $\{1,2\}$ or $\{1,3\}$ would be an optimal solution. Returning $\{1,2,3\}$ would not be optimal, as this isn't the smallest number of pieces that we could return. Returning $\{1\}$ is incorrect, as we don't return any piece of the second set in F.

Construct a counterexample for the following algorithm for this problem: Sort the pieces in non-increasing order based on the number of sets a piece occurs in. Process the pieces one at a time and add a piece to the solution if it is part of a set that doesn't have any piece in the solution yet.

Remember to:

a) describe your instance,
b) show what solution the algorithm gives for the instance and briefly explain why, and
c) show what the optimal solution of the instance is.

Solution 1.

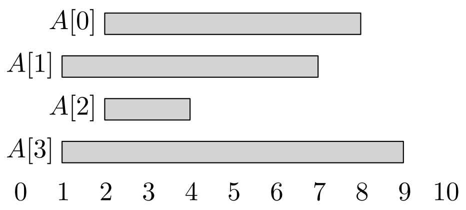
a) We aim to trick the algorithm into adding one piece that occurs in a lot of sets (but isn't actually needed) followed by a number of pieces that add only a single set (but are all needed), which results in more pieces than needed:
$P = \{p_1, p_2, p_3, p_4\}$
$F_1 = \{p_1, p_2\}$
$F_2 = \{p_1, p_3\}$
$F_3 = \{p_1, p_4\}$
$F_4 = \{p_2\}$
$F_5 = \{p_3\}$
$F_6 = \{p_4\}$
b) The algorithm would return $\{p_1, p_2, p_3, p_4\}$, since it starts by adding p_1 (which isn't needed, but occurs in most sets) and all of p_2, p_3, and p_4 (which are needed for F_4, F_5, and F_6).
c) The optimal solution needs only $\{p_2, p_3, p_4\}$ to cover all functionalities.

Problem 2. (25 points)
We're helping out with a scientific experiment where we need to take samples of a given set of specimens. These specimens are generally given as 2-dimensional areas on for example Petri dishes. Unfortunately for us, your clumsy lecturer André dropped the petri dishes, and some of the Petri dishes were broken and a number specimens have gotten mixed. However, this does present us with a unique opportunity to take samples of all specimens faster than taking these samples one at a time: when we take a sample at some location p, our sample includes a little bit of all specimens that got mixed at this location. Your task is to design an algorithm that returns (the locations of) the minimum number of samples needed to sample all specimens.

Since this problem is a bit difficult to solve in 2D, we'll look at the slightly simplified 1D version instead: You're given an array A of length n consisting of 1-dimensional line segments along the x-axis (i.e., tuples of left and right endpoints) and you can assume that each specimen $A[i] \in A$ is given as $[l_i, r_i]$. Your task is to compute (the locations of) the minimum number of samples we need to take to ensure that our samples cover all specimens.

Example:
$A = [[2, 8], [1, 7], [2, 4], [1, 9]]$

In other words, it looks like this:



In this example, the smallest set of samples is $\{3\}$, as taking a single sample there would ensure we cover all four specimens. The set $\{1, 4\}$ would also cover all specimens, but this isn't the smallest such set. The set $\{1\}$ doesn't cover all specimens, so it isn't a valid solution.

Design a greedy algorithm that returns the locations of the minimum number of samples needed to sample all specimens. For full marks, your algorithm should run in $O(n \log n)$ time. Remember to:

a) describe your algorithm in plain English,

b) argue its correctness, and

c) analyze its time complexity.

Solution 2.

a) We sort A by the r_i values in increasing order. We now scan through the specimens from left to right and take a sample when we're about to pass the right endpoint of a specimen that we haven't sampled yet. To be able to check this efficiently, we keep track of the location at which we sampled last and check if this is between the left and right endpoint of the current specimen.

```
function TakeSamples($A$)
    $result \leftarrow [\,]$
    $lastSample \leftarrow -\infty$
    Sort $A$ by the $r_i$ values in increasing order and renumber such that $|r_0| \leq |r_2| \leq ... \leq |r_{n-1}|$
    for each $A[i]$ in this order do
        if not $l_i \leq lastSample \leq r_i$ then
            $result \leftarrow result \cup r_i$
            $lastSample \leftarrow r_i$
    return $result$
```

b) We first briefly argue that *lastSample* is indeed all we need to take into account to determine if a specimen has been sampled before. Say we have a specimen $A[i]$ that is sampled before *lastSample*. This implies that $l_i \leq lastSample$. Furthermore, since we are considering $A[i]$ after we took a sample at *lastSample*, we also know that $lastSample \leq r_i$. Hence, if $A[i]$ was sampled before *lastSample*, it's also sampled by *lastSample*. We also note that if *lastSample* doesn't sample $A[i]$, this implies that $lastSample < l_i$ and hence no earlier sample could've sampled it either. Thus we can use *lastSample* for testing purposes. Combining this with adding a new sample when *lastSample* doesn't sample the specimen, ensures that we have covered all specimens before $A[i]$ when considering $A[i]$.
Next, we use an exchange argument to show that any optimal solution can be converted into our greedy solution. We consider both solutions in increasing sorted order of the coordinate of their samples. Let OPT be the optimal solution and let j be the index of the sample with the smallest coordinate in OPT that differs from *result*. This means that all specimens sampled by the first $j-1$ samples are covered in both solutions. Let $A[i]$ be the specimen that caused our greedy algorithm to add sample j. As we argued above, *lastSample* ensures that all specimens before $A[i]$ were correctly sampled. We added this new sample at the last possible moment before we would miss sampling $A[i]$. Thus, the sample used in the optimal solution needs to be to the left of our sample, as otherwise OPT wouldn't sample $A[i]$. This implies that if we replace the j-th sample in OPT by the j-th sample from our *result*, we still sample all specimens that are sampled by OPT using the first j samples. (We also note that OPT can't have another sample before our j-th sample, as this sample wouldn't be necessary to cover all specimens, thus contradicting the optimality of OPT.) By repeating this process for every sample in OPT, we can convert it into our greedy solution, showing that our algorithm is optimal.

c) The running time is dominated by the sorting step, which takes $O(n \log n)$ time using for example merge sort. The for loop scans through n specimens and performs a constant number of $O(1)$ time operations on them (if we use a linked list to keep track of *result*), so that takes $O(n)$ time in total. The initialization takes constant time and returning the result is either constant or linear time (depending on whether we explicitly return the set or only a pointer to it). Regardless, the total running time is $O(n \log n)$ as required.

Problem 3. (25 points)
For the Winter Olympics of 2086, a new extreme sport will be introduced: snowboarding with rocket-boosted snowboards! To make this event a success we're looking for a good place in the Snowy Mountains to organise this. The location we use should meet a number of conditions, but for security reasons we aren't told what exactly we're looking for yet. For example, the event may need to start from a location that has a downward slope, or we may want to finish on a flat part so the competitors can be interviewed. Another condition may be that we

want to include at least one ramp for the competitors to perform a nice jump for the cameras. While we don't know what the organisers will ask for yet, we do know that there will be exactly three things that the organisation is looking for.

After scouting the Snowy Mountains, we've been provided with a string A of length n encoding what each location can be used for. We're also given a 3-character string S, which encodes what the organisation is looking for. You can treat the strings as arrays containing a single character at each position. We need to determine the number of different ways to extract S (in order) from A to give the organisation a good idea of the number of options they have to consider. The characters of S do **not** have to be consecutive in A, as long as they occur in the correct order.

Example:
$A = acabdcb$
$S = acb$
In this example, we can extract S in four ways: 1. $A[0], A[1], A[3]$, 2. $A[0], A[1], A[6]$, 3. $A[0], A[5], A[6]$, and 4. $A[2], A[5], A[6]$. Thus, your algorithm should return 4.

Design a divide and conquer algorithm for the above problem. For full marks, your algorithm should run in $O(n)$ time. Remember to:

a) describe your algorithm in plain English,

b) argue its correctness, and

c) analyze its time complexity. (If you use the Master Theorem, don't forget to specify the parameters you get when you apply, so we can see you understand how this works.)

Solution 3.

a) We build an algorithm that computes not only the number of occurrences of the full string $S = S_{1,3}$, but also those of the substrings consisting of only the first character $S_{1,1}$, the first two characters $S_{1,2}$, only the middle character $S_{2,2}$, the last two characters $S_{2,3}$, and only the last character $S_{3,3}$. If our input is small enough, say $n < 2$, then we can compute all of these counts explicitly (i.e., all variables indicating a substring of length 2 or more have count 0 and a variable for a substring of length 1 is 1 if the character matches, and 0 otherwise). If our input is larger ($n \geq 2$), we split the input and recursively call ourselves on both parts. Any split works (even splitting into 1 and $n-1$ elements), though two halves L and R is the most obvious choice, so this solution uses that. These subarrays are passed to recursive calls using the indices, to avoid having to perform a costly copying operation. The recursive calls will solve the smaller subproblems and thus we need to focus only on combining the results of the subproblems to solve the original problem. When computing the results of the original input, we update the counts as follows ($L(X)$ indicates using the count of X from L):
$S_{1,1} = L(S_{1,1}) + R(S_{1,1})$,
$S_{1,2} = L(S_{1,2}) + R(S_{1,2}) + L(S_{1,1}) \cdot R(S_{2,2})$,
$S_{2,2} = L(S_{2,2}) + R(S_{2,2})$,
$S_{2,3} = L(S_{2,3}) + R(S_{2,3}) + L(S_{2,2}) \cdot R(S_{3,3})$,
$S_{3,3} = L(S_{3,3}) + R(S_{3,3})$,
$S_{1,3} = L(S_{1,3}) + R(S_{1,3}) + L(S_{1,1}) \cdot R(S_{2,3}) + L(S_{1,2}) \cdot R(S_{3,3})$.
In the end, we return only $S_{1,3}$ of the original array A.

b) We prove the correctness of the counts we maintain by induction on the size of the subproblems. In the base case, we explicitly computed these counts, so those are indeed correct.
For the inductive step, we assume that the counts we get out of the subproblems are correct and thus we need to argue that we correctly combine them to get the counts for the original problem. We start with the single characters: Since $S_{1,1}$, $S_{2,2}$, and $S_{3,3}$ store the frequencies of a single characters, it's easy to see that the frequency of a character in the original array is simply the sum of the occurrences in the two parts.
For the longer substrings we also need to take the combinations into account. Hence, for $S_{1,2}$ and $S_{2,3}$ we need to count both the full occurrences in the two parts, as well as the combinations having the first character in L and the second character in R. Since this combination keeps the order of the characters intact (all characters in L occur before the characters in R in A), every occurrence of the first character in L can be combined with every occurrence of the second character in R and thus multiplying these two gives the correct number of ordered combinations.
Similarly, to compute the number of occurrences of the full substring $S_{1,3}$, we need to add the number of occurrences in L, the occurrences in R, and the number of combinations of L and R. These combinations can now consist of either the first two characters being in L and the last being in R, or the first being in L and the last two being in R. Again, since L comes before R in A, summing up the products of these combinations gives the correct number of ordered combinations. Hence, we conclude that we maintain the correct counts throughout the execution, and thus by returning $S_{1,3}$ at the end, we return the correct number of options, as required.

c) Splitting the array into two equal-sized parts and combining and returning the counts takes $O(1)$ time, when we simply specify the indices to recurse on in the recursive calls. Thus the recurrence is: $T(n) = 2T(n/2) + O(1)$. Using the Master Theorem, we see that this solves to $O(n)$ time ($a = 2$, $b = 2$, case 1). Unrolling gives the same result, as we sum $O(1)$ for n, two times $O(1)$ for $n/2$, four times $O(1)$ for $n/4$, and so on, we sum $2n$ times $O(1)$ in total, which is $O(n)$.