Assignment5

Student number: 490051481

October 29, 2024

Problem 1

(a)

Consider $F = \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}$ and $P = \{1,2,3,4\}$. After Sort the pieces P in non-increasing order based on the number of sets a piece occurs in, we can get $P = \{1,2,3,4\}$ or $\{1,3,2,4\}$ or $\{1,4,2,3\}$ or some other order (since there is a tie).

(b)

Say we have $P = \{1,2,3,4\}$ (one possible sorted P from (a)). By using the provided algorithm, firstly, we add piece 1 into the solution list because set $\{1,2\}$ does not have any piece in the solution list. Secondly, we add piece 2 to the solution because set $\{2,3\}$ does not have any piece in the solution. Lastly, we add piece 3 into the solution because set $\{3,4\}$ does not have any piece in the solution. Notice that we just process the sorted pieces one at a time from left to right. You can also use other possible sorted P to get similar result (i.e. have 3 pieces in the solution), although not all sorted P could lead to above answer, such as $\{2,4,1,3\}$, but as long as there is one could lead to non-optimal solution, we can use that one as an counter example.

(c)

As you may already notice that one of the optimal solution for $P = \{1,2,3,4\}$ is $\{2,4\}$. We can only use these 2 pieces to include all functionalities instead of by using 3 pieces in (b). Hence we successfully get a counter example.

Problem 2

(a)

There are 3 steps

- Step 1: sort the array A according to the r_i of the specimen in non-decreasing order, namely former specimen's r is larger or equal than the later specimen's r.
- Step 2: initializing an empty solution set *result*,
- Step 3: traverse the sorted array from left to right one-by-one. That is
  - firstly, we set the *current_check_point* as the first specimen's r and also add it to *result* set.
  - secondly, we need to do a for-loop to traverse all specimens in the array. Specifically, comparing current specimen's l with *current_check_point* (current specimen will change to next specimen of current specimen after each iteration), and there are 2 possible conditions:
    - (1): current specimen's l larger than *current_check_point*. In this condition, we need to set *current_check_point* as current specimen's r and add the new *current_check_point* to *result*, otherwise, current specimen will not be covered by existing check point in the *result* set. We can say this because all later specimens can never have smaller r than former specimens and other check points inside *result* (include *current_check_point*) can only be former specimen's r. So if later specimen's l is larger than *current_check_point*, then it will not be covered by all check points inside *result* set.
    - (2): current specimen's l less than or equal to *current_check_point*. In this condition, we do nothing, since *current_check_point* already cover the current specimen. The reason is current specimen's l $<=$ *current_check_point* $<$ current specimen's r.
- after doing above steps, the *result* set contains the locations of the minimum number of samples needed to sample all specimens.

(b)

I think I already explain a little bit about my algorithm in (a). Below paragraph is a supplementary explanation.

After sorted the array, it guarantee the left specimen must have smaller r than right specimen's r and its own l never gonna exceed its own r. Base on this fact, we must pick one check point between $[l, r]$ of the left most specimen as the first check point, otherwise we will not cover this specimen. This is the initialization step in above Step 3.

In addition, my algorithm is greedy because it set the check point as the tail of current specimen which is the closest point to the rest specimens and this point (comparing with other points within the range of current specimen) cover the most specimens. I come to this

conclusion because the lower bound of the l of other right specimens that overlap with the current specimen is the r of the current specimen. That means the check point is the local optimum.

Lastly, if there is a new specimen's l larger than *current_check_point*, then we need to set the new *current_check_point* as the new specimen's r and repeat above steps. Refer to step 3 condition 2.

(c)

Step 1 takes $O(nlogn)$ by using Mrge sort from the lecture.

Step 2 takes $O(1)$.

Step 3 takes $O(n) * O(1) = O(n)$ since the for-loop only traverse the array once, and inside the for-loop, we only do simple assignment and comparison.

In total, we find step 1 is the dominating term which lead to the overall complexity of this algorithm is $O(nlogn)$

Problem 3

(a)

Assume $S = abc$, where a, b, c are random character. We split the array A into 2 subspaces, call them $left$ and $right$. Notice that, the base case is reached when subspace size is less than or equal to 3, namely, whenever subspace length is less than or equal to 3, then we stop dividing and do the conquer thing.

During recursion, in $left$ subspace, we will count number of below terms (both side can have below terms, for example $right$ side can also have TERM 3, but we just ignore them on that side):

- TERM 1: $k_1(a)k_2(b)k_3(c)k_4$, where k_i is random character, they can also be null.
- TERM 2: $k_1(a)k_2(b)k_3$, where k_i is random character, they can also be null.
- TERM 3: $k_1(a)k_2$, where k_i is random character, they can also be null.
- TERM X: $k_1(b)k_2$, where k_i is random character that is not inside S, they can also be null.

During recursion, in $right$ subspace, we will count number of below terms(both side can have below terms, for example $left$ side can also have TERM 5):

- TERM 4: $k_1(a)k_2(b)k_3(c)k_4$, where k_i is random character, they can also be null.
- TERM 5: $k_1(b)k_2(c)k_3$, where k_i is random character, they can also be null.
- TERM 6: $k_1(c)k_2$, where k_i is random character, they can also be null.
- TERM Y: $k_1(b)k_2$, where k_i is random character that is not inside S, they can also be null.

Notice that, we should also use TERM X (from $left$) and TERM 6 (from $right$) to generate TERM 5 for their parent array, or TERM 3 (from $left$) and TERM Y (from $right$) to generate TERM 2 for their parent array (If possible). If we do not do this, we gonna miss some information.

Finally, the number of S we can extract from A is equal to number of TERM 1 + TERM 4 + TERM2*TERM6 + TERM3*TERM5

(b)

It is correct. In the base case, there are only 2 sub-array with maximum length equal to 3. And it is quiet easy for you to check that above formula can catch all S in parent array. Then go back to top level, You can also get the number of all possible ways to extract S from A because there is not another combination between $left$ and $right$ for A to get S. Besides, I also consider about the formation of TERM 2 and TERM 5, since a and b may not always appear in the same side, so I need to combine them.

(c)

$$T(n) = \begin{cases} 2T(n/2) + O(1), & \text{if } n \geq 1, \\ O(1), & \text{if } n = 1. \end{cases}$$

Each iteration, we split the array into half until we reach the base case. Besides, each parent class only do one simple calculation and few assignments by using return value from children classes, so the conquer step should be constant. In the base case, we just count the number of above TERMs which is also cost constant time. By using mater theorem, we have $a = 2, b = 2, f(n) = 1$, so we have $log_b a = 1$, that is case 1, hence we have time complexity $O(n)$.