CS 240 Homework 1

TOTAL POINTS

7 / 7

QUESTION 1

1 1 / 1

✓ **- 0 pts Correct**

- 0.1 pts One inversion

- 0.2 pts Two inversion

- 0.3 pts Three inversion

- 0.4 pts Four inversion

- 0.5 pts Five inversion

- 0.6 pts Six inversion

- 0.7 pts Seven inversion

- 0.8 pts Eight inversion

QUESTION 2

2 1 / 1

✓ **- 0 pts Correct**

- 0.2 pts Not providing complexity of O(nlgn)

- 0.4 pts Without proof of optimality or totally wrong proof

- 0.4 pts Wrong Algorithm

- 0.2 pts Incomplete or unclear proof, or proof with mistake

- 1 pts Wrong Answer

- 0.2 pts Not optimal algorithm

QUESTION 3

3 1 / 1

✓ **- 0 pts Correct**

- 0.3 pts Unclear proof

- 0.2 pts Expected sizes sort

- 0.2 pts Proof less of case

- 0.8 pts Wrong

- 0.5 pts No proof

- 1 pts No answer

- 0.3 pts Sizes of cake sort

- 0.3 pts Complex of algorithm

QUESTION 4

4 1 / 1

✓ **+ 1 pts Correct**

+ 0.75 pts Correct main idea with minor mistake

+ 0.5 pts Partial correct main idea.

+ 0.25 pts Big mistake with some interesting finding.

+ 0 pts Incorrect

QUESTION 5

5 1 / 1

✓ **- 0 pts Correct**

- 0.5 pts (b)wrong

- 0.5 pts (a) wrong

- 0.2 pts (a)lack of details

- 0.3 pts (a)need more details

- 0.1 pts (b) K is not a constant

- 0.3 pts (b) wrong equation

- 0.2 pts wrong statement or misunderstand

- 1 pts can not find your solution

QUESTION 6

6 1 / 1

✓ **+ 1 pts Correct**

+ 0.5 pts Partial Correct.

+ 0.5 pts Partial correct recursive formula

+ 0 pts Incorrect

QUESTION 7

7 1 / 1

✓ **+ 1 pts Correct**

+ 0.5 pts Get the right DP function: s[I][j]=max{s[I][j-1], s[I+2^{j-1}][j-1]}. The preprocess is correct.

+ 0.5 pts Correctly transfer a query to two access of the table. The query part is correct.

- 0.2 pts little mistake

+ **0 pts** wrong or empty

Problem 1

Solution

Because:

$$\lim_{x\to\infty}(\frac{f_4(n)}{f_2(n)}) = \lim_{x\to\infty}(\frac{10^{10^{10^{10}}}}{n^2 logn}) = 0$$

So, $f_4(n)$ is $O(f_2(n))$.
Then,

$$\lim_{x\to\infty}(\frac{f_2(n)}{f_6(n)}) = \lim_{x\to\infty}(\frac{n^2 logn}{2^{\frac{5}{2}log_2 n}})$$
$$= \lim_{x\to\infty}(\frac{n^2 logn}{n^{\frac{5}{2}}})$$
$$= \lim_{x\to\infty}(\frac{logn}{n^{\frac{1}{2}}})$$
$$= 0$$

So, $f_2(n)$ is $O(f_6(n))$.
Then,

$$\lim_{x\to\infty}(\frac{log(f_6(n))}{log(f_1(n))}) = \lim_{x\to\infty}(\frac{log(2^{\frac{5}{2}log_2 n})}{log(n^{log_2 n})})$$
$$= \lim_{x\to\infty}(\frac{\frac{5}{2}log_2 n \cdot log2}{log_2 n \cdot logn})$$
$$= \lim_{x\to\infty}(\frac{\frac{5}{2}log2}{logn})$$
$$= 0$$

Thus, $f_6(n)$ is $O(f_1(n))$

$$\lim_{x\to\infty}(\frac{log(f_1(n))}{log(f_5(n)}) = \lim_{x\to\infty}(\frac{log(n^{log_2 n})}{log(2^{\sqrt{n}})})$$
$$= \lim_{x\to\infty}(\frac{log_2(n) \cdot log(n)}{log2 \cdot \sqrt{n}})$$
$$For\ any\ k > 1, we\ have$$
$$\lim_{x\to\infty}(\frac{\log^k(n)}{\sqrt{n}}) = 0$$
$$So,\ \lim_{x\to\infty}(\frac{log_2(n) \cdot log(n)}{log2 \cdot \sqrt{n}}) = 0$$

Hence, $f_1(n)$ is $O(f_5(n))$
Finally, we prove $f_5(n)$ is $O(f_3(n))$

$$\lim_{x\to\infty}(\frac{log(f_5(n))}{log(f_3(n))}) = \lim_{x\to\infty}(\frac{log(2^{\sqrt{n}})}{log(2^{2^n})})$$
$$\lim_{x\to\infty}(\frac{log2 \cdot \sqrt{n}}{2^n})$$
$$= 0$$

So, we've proved $f_5(n)$ is $O(f_3(n))$.
By the limits shown above, we have the sequence:

$$f_4(n),\ f_2(n),\ f_6(n),\ f_1(n),\ f_5(n),\ f_3(n)$$

1 1 / 1

✓ **- 0 pts Correct**

- 0.1 pts One inversion

- 0.2 pts Two inversion

- 0.3 pts Three inversion

- 0.4 pts Four inversion

- 0.5 pts Five inversion

- 0.6 pts Six inversion

- 0.7 pts Seven inversion

- 0.8 pts Eight inversion

Problem 2

Algorithm

Use greedy algorithm.

First, sort the given stones in ascending order by their weights w_i. (i.e. $s_0 < s_1 \ldots < s_n$)

Then, we initialize $sum = 0$, after i^{th} pick, we update sum to $\sum_{j=1}^{i} s_j$.

For each pick, pick the left-most stone(i.e. the stone with smallest index in sorted array) which satisfies $s_{pick_i} > \sum_{j=1}^{i} s_j$

The python code can be writen as:

```
def pick_stones(weights: list[int])->list[int]:
    weights.sort()
    res = []
    sum_ = 0
    for w in weights:
        if w <= sum_:
            continue
        sum_ += w
        res.append(w)
    return res
```

Proof

The correctness of this algorithm can be proved by contradiction.

Let's assume there exists a optimal solution different from the solution given by this algorithm. Let the optimal solution be $a_1, a_2, \ldots, a_i, \ldots, a_n$, and the solution given by this algorithm be $b_1, b_2, \ldots, b_i, \ldots, b_m$, where $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$, for the largest possible i. But $a_i \neq b_i$. By requirement of a "tower", $s_{b_i} > \sum_{j=1}^{i} s_{b_j}$, and $s_{a_i} > \sum_{j=1}^{i} s_{a_j}$. Then, by our assumption, $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$, we know $s_{a_i} > \sum_{j=1}^{i} s_{b_j}$. Because b_i is the minimum value which satisfies $s_{pick_i} > \sum_{j=1}^{i} s_j$, then we can get $a_i \geq b_i$. Then, if we replace a_i with b_i, the total height would not be smaller, for a_{i+1} to a_n is able to stay the same. Thus we've proved $a_i = b_i$, so, by induction, b_i is the optimal solution.

Complexity Analysis

Let n be the size of input,

Then, sorting (pre-processing) will take $O(nlogn)$ time.

Find which stone to pick will take $O(n)$ time, so $O(nlogn + n) = O(nlogn)$ in total. Hence, the time complexity of this algorithm is $O(nlogn)$.

For space complexity, sorting will take $O(1)$ space, and get the correct answer will take $O(m)$ space, where m is the size of output. If we output the result immediately, then the space complexity will be $O(1)$.

Problem 3

Algorithm

Use greedy algorithm.

First, sort the children's expected sizes and sizes of cakes, both in ascending order.

Then, use two pointers, for each children, from left to right, give each children the minimum size of cake which can satisfy him if possible.

The python code is just like this:

```
def give_cakes(expected: list[int], cakes: list[int])->int:
    """Returns the maximum number of children that can be satisfied"""
    expected.sort()
    cakes.sort()
```

Problem 3 continued on next page...

2 1 / 1

✓ **- 0 pts Correct**

- **0.2 pts** Not providing complexity of O(nlgn)
- **0.4 pts** Without proof of optimality or totally wrong proof
- **0.4 pts** Wrong Algorithm
- **0.2 pts** Incomplete or unclear proof, or proof with mistake
- **1 pts** Wrong Answer
- **0.2 pts** Not optimal algorithm

Problem 2

Algorithm

Use greedy algorithm.

First, sort the given stones in ascending order by their weights w_i. (i.e. $s_0 < s_1 \ldots < s_n$)

Then, we initialize $sum = 0$, after i^{th} pick, we update sum to $\sum_{j=1}^{i} s_j$.

For each pick, pick the left-most stone(i.e. the stone with smallest index in sorted array) which satisfies $s_{pick_i} > \sum_{j=1}^{i} s_j$

The python code can be writen as:

```
def pick_stones(weights: list[int])->list[int]:
    weights.sort()
    res = []
    sum_ = 0
    for w in weights:
        if w <= sum_:
            continue
        sum_ += w
        res.append(w)
    return res
```

Proof

The correctness of this algorithm can be proved by contradiction.

Let's assume there exists a optimal solution different from the solution given by this algorithm. Let the optimal solution be $a_1, a_2, \ldots, a_i, \ldots, a_n$, and the solution given by this algorithm be $b_1, b_2, \ldots, b_i, \ldots, b_m$, where $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$, for the largest possible i. But $a_i \neq b_i$. By requirement of a "tower", $s_{b_i} > \sum_{j=1}^{i} s_{b_j}$, and $s_{a_i} > \sum_{j=1}^{i} s_{a_j}$. Then, by our assumption, $a_1 = b_1, \ldots, a_{i-1} = b_{i-1}$, we know $s_{a_i} > \sum_{j=1}^{i} s_{b_j}$. Because b_i is the minimum value which satisfies $s_{pick_i} > \sum_{j=1}^{i} s_j$, then we can get $a_i \geq b_i$. Then, if we replace a_i with b_i, the total height would not be smaller, for a_{i+1} to a_n is able to stay the same. Thus we've proved $a_i = b_i$, so, by induction, b_i is the optimal solution.

Complexity Analysis

Let n be the size of input,

Then, sorting (pre-processing) will take $O(nlogn)$ time.

Find which stone to pick will take $O(n)$ time, so $O(nlogn + n) = O(nlogn)$ in total. Hence, the time complexity of this algorithm is $O(nlogn)$.

For space complexity, sorting will take $O(1)$ space, and get the correct answer will take $O(m)$ space, where m is the size of output. If we output the result immediately, then the space complexity will be $O(1)$.

Problem 3

Algorithm

Use greedy algorithm.

First, sort the children's expected sizes and sizes of cakes, both in ascending order.

Then, use two pointers, for each children, from left to right, give each children the minimum size of cake which can satisfy him if possible.

The python code is just like this:

```
def give_cakes(expected: list[int], cakes: list[int])->int:
    """Returns the maximum number of children that can be satisfied"""
    expected.sort()
    cakes.sort()
```

Problem 3 continued on next page...

```
        res = 0
        j = 0
        for i in range(len(expected)):
            # Go through the cakes until find one that satisfies this children
            while (j < len(cakes)) and (cakes[j] < expected[i]):
                j += 1
            # Break here, for no more children can be satisfied
            if j == len(cakes):
                break
            # The ith child is satisfied
            res += 1
        return res
```

Proof

Let the sorted matches between children and cakes by this algorithm are $S = [< e_1, c_1 >, < e_2, c_2 >, \ldots, < e_m, c_m >]$. But the sorted optimal matches are $S^* = [< e'_1, c'_1 >, < e'_2, c'_2 >, \ldots, < e'_{opt}, c'_{opt} >]$, where $opt > m$, for the optimal solution can match more children. Here, the "sorted" means $e_1 < e_2 < \ldots < e_k$, for all k.

First, we prove $e'_1, e'_2, \ldots, e'_{opt}$ are also opt^{th} smallest expectations. Beacuse if e'_k is not the k^{th} smallest, we can just replace it with the k^{th} smallest, for $c'_k \geq e'_k > k^{th}$ *smallest*, without carrying any effects to other matches.

Here, we define a inversion: $e_x > e_y \geq c_x > c_y$, but the match is $< e_x, c_y >, < e_y, c_x >$. If we match it as $< e_x, c_x >, < e_y, c_y >$, it will not decrease the number of matched pairs.

Obviously, this algorithm produces no inversions. But if their exists an optimal solution different from this solution, there must exist an inversion in it. For $e_1 = e'_1, e_2 = e'_2, \ldots$. Then we can just adjust the inversion, without decreasing matches.
Hence, we've proved $S = S^*$ for we proved $e_1 = e'_1, e_2 = e'_2, \ldots$, and $c_1 = c'_1, c_2 = c'_2, \ldots$.

Complexity Analysis

Assume there are m children and n cakes, The two sorting will take $O(nlogn + mlogm)$ time. Then find the optimal matches is a one-pass process, which takes $O(m + n)$ time. So, the total time complexity is $O(nlogn + mlogm)$, which can also be represent as $O(\max\{m, n\} \cdot log(\max\{m, n\}))$

And the space complexity is $O(1)$ for no additional spaces are used.

Problem 4

Algorithm

First, sort the students by x in ascending order (i.e. $x_0 < x_1 < \ldots < x_n$). Then, use divide and conquer algorithm to count the number of inverse pairs about y by descending order (for some i, j satisfies $i < j$ but $arr[i].y < arr[j].y$). This count is the number of friends.
The algorithm that count the number of inverse pairs is a extension of merge sort. When merge the two sub-arrays, just named arr_L and arr_R, We count the inverse pairs between arr_L and arr_R (i.e. all (i, j) pairs satisfiy $arr_L[i].y < arr_R[j].y$). This can be counted in $O(n)$ time, for when we merge from arr_R, count should be updated to $sum += len(arr_L) - i$. Then, find the inverse pairs in arr_L and arr_R recursively (by divide arr_L and arr_R). The python code of the overall process is shown below:

Problem 4 continued on next page...

3 **1 / 1**

✓ **- 0 pts Correct**

- 0.3 pts Unclear proof

- 0.2 pts Expected sizes sort

- 0.2 pts Proof less of case

- 0.8 pts Wrong

- 0.5 pts No proof

- 1 pts No answer

- 0.3 pts Sizes of cake sort

- 0.3 pts Complex of algorithm

```
        res = 0
        j = 0
        for i in range(len(expected)):
            # Go through the cakes until find one that satisfies this children
            while (j < len(cakes)) and (cakes[j] < expected[i]):
                j += 1
            # Break here, for no more children can be satisfied
            if j == len(cakes):
                break
            # The ith child is satisfied
            res += 1
        return res
```

Proof

Let the sorted matches between children and cakes by this algorithm are $S = [< e_1, c_1 >, < e_2, c_2 >, \ldots, < e_m, c_m >]$. But the sorted optimal matches are $S^* = [< e'_1, c'_1 >, < e'_2, c'_2 >, \ldots, < e'_{opt}, c'_{opt} >]$, where $opt > m$, for the optimal solution can match more children. Here, the "sorted" means $e_1 < e_2 < \ldots < e_k$, for all k.

First, we prove $e'_1, e'_2, \ldots, e'_{opt}$ are also opt^{th} smallest expectations. Beacuse if e'_k is not the k^{th} smallest, we can just replace it with the k^{th} smallest, for $c'_k \geq e'_k > k^{th}$ *smallest*, without carrying any effects to other matches.

Here, we define a inversion: $e_x > e_y \geq c_x > c_y$, but the match is $< e_x, c_y >, < e_y, c_x >$. If we match it as $< e_x, c_x >, < e_y, c_y >$, it will not decrease the number of matched pairs.

Obviously, this algorithm produces no inversions. But if their exists an optimal solution different from this solution, there must exist an inversion in it. For $e_1 = e'_1, e_2 = e'_2, \ldots$. Then we can just adjust the inversion, without decreasing matches.
Hence, we've proved $S = S^*$ for we proved $e_1 = e'_1, e_2 = e'_2, \ldots$, and $c_1 = c'_1, c_2 = c'_2, \ldots$.

Complexity Analysis

Assume there are m children and n cakes, The two sorting will take $O(nlogn + mlogm)$ time. Then find the optimal matches is a one-pass process, which takes $O(m + n)$ time. So, the total time complexity is $O(nlogn + mlogm)$, which can also be represent as $O(\max\{m, n\} \cdot log(\max\{m, n\}))$

And the space complexity is $O(1)$ for no additional spaces are used.

Problem 4

Algorithm

First, sort the students by x in ascending order (i.e. $x_0 < x_1 < \ldots < x_n$). Then, use divide and conquer algorithm to count the number of inverse pairs about y by descending order (for some i, j satisfies $i < j$ but $arr[i].y < arr[j].y$). This count is the number of friends.
The algorithm that count the number of inverse pairs is a extension of merge sort. When merge the two sub-arrays, just named arr_L and arr_R, We count the inverse pairs between arr_L and arr_R (i.e. all (i, j) pairs satisfiy $arr_L[i].y < arr_R[j].y$). This can be counted in $O(n)$ time, for when we merge from arr_R, count should be updated to $sum += len(arr_L) - i$. Then, find the inverse pairs in arr_L and arr_R recursively (by divide arr_L and arr_R). The python code of the overall process is shown below:

Problem 4 continued on next page...

```
class Stuent:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def compare(student):
    return student.x

def count_friends(students : list[Student]):
    students.sort(key=compare) # sort students by 'x' in ascending order
    friend_count = 0

    def merge(array, start, mid, end, temp):
        nonlocal friend_count
        i, j = start, mid + 1
        while i <= mid and j <= end:
            if array[i].y >= array[j].y: # merge by descending order by 'y'
                temp.append(array[i]); i += 1
            else:
                # This is the most important step to count the friends
                frient_count += mid - i + 1
                temp.append(array[j]); j += 1
        while i <= mid:
            temp.append(array[i]); i += 1
        while j <= end:
            temp.append(array[j]); j += 1
        for i in range(len(temp)): array[start + i] = temp[i]
        temp = []

    def merge_sort(array, start, end, temp):
        if start >= end: return
        mid = (start + end)//2
        merge_sort(array, start, mid, temp)
        merge_sort(array, mid + 1, end, temp)
        merge(array, start, mid,  end, temp)
    merge_sort(students, 0, len(students) - 1, [])
    return frient_count
```

Problem 5

Solution

- a)
  Lemma
  If $0 \leq a_i < 2^K$ for all i, then,

$$0 \leq \min\{\max_{1 \leq i \leq n}(a_i \oplus X)\} < 2^K$$

  and

$$0 \leq X < 2^K$$

  1. Proof of lemma
     First of all, the XOR operation only change the bits, cannot generate negative numbers, and it's defined on non-negative numbers, so $0 \leq \min_{i \leq X \leq n}(a_i \oplus X)$ and $0 \leq X$.
     Notice that:

$$\forall j \geq K, \forall a_i \in A, bin(a_i)_j = 0$$

4 1 / 1

✓ **+ 1 pts** **Correct**

+ **0.75 pts** Correct main idea with minor mistake

+ **0.5 pts** Partial correct main idea.

+ **0.25 pts** Big mistake with some interesting finding.

+ **0 pts** Incorrect

```
class Stuent:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def compare(student):
    return student.x

def count_friends(students : list[Student]):
    students.sort(key=compare) # sort students by 'x' in ascending order
    friend_count = 0

    def merge(array, start, mid, end, temp):
        nonlocal friend_count
        i, j = start, mid + 1
        while i <= mid and j <= end:
            if array[i].y >= array[j].y: # merge by descending order by 'y'
                temp.append(array[i]); i += 1
            else:
                # This is the most important step to count the friends
                frient_count += mid - i + 1
                temp.append(array[j]); j += 1
        while i <= mid:
            temp.append(array[i]); i += 1
        while j <= end:
            temp.append(array[j]); j += 1
        for i in range(len(temp)): array[start + i] = temp[i]
        temp = []

    def merge_sort(array, start, end, temp):
        if start >= end: return
        mid = (start + end)//2
        merge_sort(array, start, mid, temp)
        merge_sort(array, mid + 1, end, temp)
        merge(array, start, mid,  end, temp)
    merge_sort(students, 0, len(students) - 1, [])
    return frient_count
```

Problem 5

Solution

- a)
  Lemma
  If $0 \leq a_i < 2^K$ for all i, then,

$$0 \leq \min\{\max_{1 \leq i \leq n}(a_i \oplus X)\} < 2^K$$

  and

$$0 \leq X < 2^K$$

  1. Proof of lemma
     First of all, the XOR operation only change the bits, cannot generate negative numbers, and it's defined on non-negative numbers, so $0 \leq \min_{i \leq X \leq n}(a_i \oplus X)$ and $0 \leq X$.
     Notice that:

$$\forall j \geq K, \forall a_i \in A, bin(a_i)_j = 0$$

Here, $bin(x)_j$ donates the j^{th} number in the binary representation of x, the right-most bit is 0^{th} bit.
Then, because we want to minimize $\max_{1\le i\le n}(a_i \oplus X)$, then K^{th} bit and the left hand side of K^{th} bit of X and $result$ should be all 0s, to make the result less than 2^K. Otherwise the result must greater or equal than 2^K. Hence, $\min\{\max_{1\le i\le n}(a_i \oplus X)\} < 2^K$, and $X < 2^K$.

2. Proof of the base case
If there are only one bit in a_i's binary expression(i.e. $K = 1$), then there are 3 possible conditions, $A = \{0\}$, $A = \{1\}$ and $A = \{0, 1\}$. For $A = \{0\}$ or $A = \{1\}$, notice that $x \oplus x = 0$, , and all other numbers will make the value greater than 0, so the minimum value is 0. If $A = \{0, 1\}$, by the lemma, X can only be 0, 1, and the value is 1 for both case. So, the base case of the pesudo code is correct.
3. Proof of transition
In the pesudo code, we divide the array to two parts, one part (B) satisfies $\forall x \in B, bin(x)_{K-1} = 1$, and the other part (C) satisfies $\forall x \in C, bin(x)_{K-1} = 0$. If B is empty, just set $bin(X)_{K-1} = 0$, then $bin(result)_{K-1} = 0$, is minimized. Else if C is empty, just set $bin(X)_{K-1} = 1$, then $bin(result)_{K-1} = 0$, is also minimized. If B and C are all non-empty, then $bin(result)_{K-1}$ can not be 0 at all. The result should plus 2^{K-1}. After determining the $(K-1)^{th}$ of answer, just find $(K-2)^{th}$ recursively. Hence, we proved the transition function.

Thus, we have proved the base case and transition function, by mathmatical induction, the overall result is correct.

- b)
We can find the recursive function of $T(k, n)$, where n is the length of the array K:

$$T(k, n) = n + T(k-1, x) + T(k-1, n-x)$$

where x is the number of items in B (i.e. number of items that greater or equal than 2^{k-1}). Notice that

$$\begin{aligned}
&T(k, n) = n + T(k-1, x) + T(k-1, n-x)\\
&= n + x + T(k-2, x_1) + T(k-2, x-x_1) + n - x + T(k-2, x_2) + T(k-2, n-x-x_2)\\
&= 2n + T(k-2, x_1) + T(k-2, x-x_1) + T(k-2, x_2) + T(k-2, n-x-x_2)\\
&= \dots\\
&= n \cdot k
\end{aligned}$$

Hence, the time complexity $T(n)$ is $O(nk)$

Problem 6

Algorithm
Use the dynamic programming algorithm.
The base case is, any charater(sequence with length=1) $'c'$ and empty sequence $''$ can only be merged to "c". Then, we use a backtrace DP algorithm to solve the problem recursively. We consider whether the left-most i charaters in A and left-most j charaters in B can be merged to the left-most $i + j$ charaters in C. The transition function is:

$$OPT(i,j) = \begin{cases} OPT(i, j-1), & if\ (C[i+j]\ ==\ B[j]\ \&\&\ C[i+j]\ \neq\ A[i]) \\ OPT(i-1, j), & if\ (C[i+j]\ ==\ A[i]\ \&\&\ C[i+j]\ \neq\ B[j]) \\ OPT(i-1, j)\ |\ OPT(i, j-1), & if\ (C[i+j]\ ==\ A[i]\ \&\&\ C[i+j]\ ==\ B[j]) \\ False, & Otherwise \end{cases}$$

5 **1 / 1**

✓ **- 0 pts** **Correct**

- 0.5 pts (b)wrong

- 0.5 pts (a) wrong

- 0.2 pts (a)lack of details

- 0.3 pts (a)need more details

- 0.1 pts (b) K is not a constant

- 0.3 pts (b) wrong equation

- 0.2 pts wrong statement or misunderstand

- 1 pts can not find your solution

Here, $bin(x)_j$ donates the j^{th} number in the binary representation of x, the right-most bit is 0^{th} bit.
Then, because we want to minimize $\max_{1\le i\le n}(a_i \oplus X)$, then K^{th} bit and the left hand side of K^{th} bit of X and $result$ should be all 0s, to make the result less than 2^K. Otherwise the result must greater or equal than 2^K. Hence, $\min\{\max_{1\le i\le n}(a_i \oplus X)\} < 2^K$, and $X < 2^K$.

2. Proof of the base case
If there are only one bit in a_i's binary expression(i.e. $K = 1$), then there are 3 possible conditions, $A = \{0\}$, $A = \{1\}$ and $A = \{0, 1\}$. For $A = \{0\}$ or $A = \{1\}$, notice that $x \oplus x = 0$, , and all other numbers will make the value greater than 0, so the minimum value is 0. If $A = \{0, 1\}$, by the lemma, X can only be 0, 1, and the value is 1 for both case. So, the base case of the pesudo code is correct.
3. Proof of transition
In the pesudo code, we divide the array to two parts, one part (B) satisfies $\forall x \in B, bin(x)_{K-1} = 1$, and the other part (C) satisfies $\forall x \in C, bin(x)_{K-1} = 0$. If B is empty, just set $bin(X)_{K-1} = 0$, then $bin(result)_{K-1} = 0$, is minimized. Else if C is empty, just set $bin(X)_{K-1} = 1$, then $bin(result)_{K-1} = 0$, is also minimized. If B and C are all non-empty, then $bin(result)_{K-1}$ can not be 0 at all. The result should plus 2^{K-1}. After determining the $(K-1)^{th}$ of answer, just find $(K-2)^{th}$ recursively. Hence, we proved the transition function.

Thus, we have proved the base case and transition function, by mathmatical induction, the overall result is correct.

- b)
We can find the recursive function of $T(k, n)$, where n is the length of the array K:

$$T(k, n) = n + T(k-1, x) + T(k-1, n-x)$$

where x is the number of items in B (i.e. number of items that greater or equal than 2^{k-1}). Notice that

$$\begin{aligned}
&T(k, n) = n + T(k-1, x) + T(k-1, n-x)\\
&= n + x + T(k-2, x_1) + T(k-2, x-x_1) + n - x + T(k-2, x_2) + T(k-2, n-x-x_2)\\
&= 2n + T(k-2, x_1) + T(k-2, x-x_1) + T(k-2, x_2) + T(k-2, n-x-x_2)\\
&= \dots\\
&= n \cdot k
\end{aligned}$$

Hence, the time complexity $T(n)$ is $O(nk)$

Problem 6

Algorithm
Use the dynamic programming algorithm.
The base case is, any charater(sequence with length=1) $'c'$ and empty sequence $''$ can only be merged to "c". Then, we use a backtrace DP algorithm to solve the problem recursively. We consider whether the left-most i charaters in A and left-most j charaters in B can be merged to the left-most $i + j$ charaters in C. The transition function is:

$$OPT(i,j) = \begin{cases} OPT(i, j-1), & if\ (C[i+j]\ ==\ B[j]\ \&\&\ C[i+j]\ \neq\ A[i]) \\ OPT(i-1, j), & if\ (C[i+j]\ ==\ A[i]\ \&\&\ C[i+j]\ \neq\ B[j]) \\ OPT(i-1, j)\ |\ OPT(i, j-1), & if\ (C[i+j]\ ==\ A[i]\ \&\&\ C[i+j]\ ==\ B[j]) \\ False, & Otherwise \end{cases}$$

6 **1 / 1**

✓ **+ 1 pts** **Correct**

+ 0.5 pts Partial Correct.

+ 0.5 pts Partial correct recursive formula

+ 0 pts Incorrect

Hence, we can find $OPT(m, n)$ by backtrace dynamic programming method in $O(mn)$ time.

Problem 7

Algorithm
This is can be solved by constructing the sparse table (ST algorithm), which is a dynamic programming algorithm that takes $O(nlogn)$ time for pre-processing and $O(1)$ time for each query. To break the ties, the subsricpt of the are start from 1 (i.e. ...)

- a) Pre-processing
  (1) Construct a 2D table F with size $n*logn$. Define $F(i, j)$ is the maximum number in $a_i, a_{i+1}, \ldots, a_{i+2^j-1}$
  (2) Fill the $(i, 0)$ entries of table F, the base case is, $F(i, 0) = a_i$, because there is only one number a_i here, obviously, the maximum in $\{a_i\}$ is a_i
  (3) Fill other entries of F. Notice that

  $$\max\{a_i, a_{i+1}, \ldots, a_{i+2^j-1}\} = \max\{\max\{a_i, a_{i+1}, \ldots, a_{i+2^{j-1}-1}\}, \max\{a_{i+2^{j-1}}, a_{i+2^{j-1}+1}, \ldots, a_{i+2^j-1}\}\}$$

  where

  $$\max\{a_i, a_{i+1}, \ldots, a_{i+2^{j-1}-1}\} = F(i, j-1)$$

  and

  $$\max\{a_{i+2^{j-1}}, a_{i+2^{j-1}+1}, \ldots, a_{i+2^j-1}\} = F(i+2^{j-1}, j-1)$$

  Hence, we get the transition function:

  $$F(i, j) = \max\{F(i, j-1), F(i+2^{j-1}, j-1)\}$$

  The pre-processing process is shown above, which will take $O(nlogn)$ time.

- b) Query
  Define the query operation $Q(l, r) = \max_{l<i<r} a_i$, and let $l' = l + 1$ and $r' = r - 1$, then $Q(l, r) = \max_{l' \le i \le r'} a_i$.
  When querying (1) Compute the maximum k that satisfies $2^k \le r' - l' + 1$, obviously the answer is $k = (int)log_2(r' - l' + 1)$ (2) Divide $a_{l'}, a_{l'+1}, \ldots, a_{r'}$ to 2 parts (may have overlaps) and find the maximum in each part.

  $$\max_{l' \le i \le r'} a_i = \max\{\max\{a_{l'}, \ldots, a_{l'+2^k-1}\}, \max\{a_{r'-2^k+1}, \ldots, a_{r'}\}\}$$

  where

  $$\max\{a_{l'}, \ldots, a_{l'+2^k-1}\} = F(l', k)$$
  $$\max\{a_{r'-2^k+1}, \ldots, a_{r'}\} = F(r' - 2^k + 1, k)$$

  are all in our table F.
  Hence, we get the overall process of finding $Q(l, r)$:

  $$let\ i = l + 1;$$
  $$let\ j = r - 1;$$
  $$let\ k = (int)log_2(j - i + 1);$$
  $$Q(l, r) = \max\{F(i, k), F(j - 2^k + 1, k)\}$$

  This step only takes $O(1)$ time.

Hence, we get the complete algorithm, which takes $O(nlogn + q)$ time.

7 1 / 1

✓ **+ 1 pts Correct**

+ 0.5 pts Get the right DP function: s[I][j]=max{s[I][j-1], s[I+2^{j-1}][j-1]}. The preprocess is correct.

+ 0.5 pts Correctly transfer a query to two access of the table. The query part is correct.

- 0.2 pts little mistake

+ 0 pts wrong or empty