Problem 1. (?%)

By adding a source that links to all boys and a drain that linked by all girls, we can easily realize that the size of max flow of the modified graph is equivalent to the maximum pairs in original question. So this question can be solved using Ford-Fulkerson algorithm taught in class.

For the running time part, For-Fulkerson's time complexity is $O(E * |max - flow|)$. But we can see that in fact $|max - flow| = O(v)$. So the running time of Ford Fulkerson is $O(VE)$.

Problem 2. Prove the closeness of P (16%)

Show that the class P, viewed as a set of languages, is closed under the following operations. Assume M_1 is the machine that decides L_1 in polynomial time, M_2 is the machine that decides L_2 in polynomial time.

(1) (4%) $L_1 - L_2 \in$ P

We can construct a M_3 that can decide $L_1 - L_2$ in polynomial time.

M_3 (s)

if $M_1(s) == 1$ and $M_2(s) == 0$ **then**

return 1

else

return 0

end if

<u>Common Mistake</u>

$L_1 - L_2 \subseteq L_1$

$L_1 \in P, \therefore L_1 - L_2 \in P$

This is wrong.

$L \in P,\ L' \subseteq L$ doesn't mean that $L' \in P$

For example, let L be Σ^*, $L \in P$, however, there are many L' subset L but not in P.

(2) (4%) $L_1 L_2 \in$ P

We can construct a M_3 that can decide $L_1 L_2$ in polynomial time.

M_3 (s)

if $M_1(\epsilon) == 1$ and $M_2(s) == 1$ **then**

return 1

end if

if $M_2(\epsilon) == 1$ and $M_1(s) == 1$ **then**

return 1

end if

for i=1:len(s)-1 **do**

if $M_1(s[1:i]) == 1$ and $M_2(s[i+1:len(s)]) == 1$ **then**

return 1

end if

end for

return 0

Because M_3 only needs to try $n+1$ cases, so it still runs in polynomial time.

(3) (4%) $\bar{L_1} \in$ P

We can construct a M_3 that can decide $\bar{L_1}$ in polynomial time.

M_3 (s)

if $M_1(s) == 1$ **then**

return 0

else

return 1

end if

(4) (4%) $L_1^* \in$ P.

$L_1^2 = L_1 L_1$, by (2), $\therefore L_1^2 \in P$

$L_1^3 = L_1^2 L_1$, by (2), $\therefore L_1^3 \in P$

By the same rule, we can derive that $L_1^k \in P$ for any $k \in N$.

That is, for any $k \in N$, there is a polynomial time machine M_ONE_k that decides L_1^k.

Also, there exists a polynomial time machine M_ONE_0 that can decide the language $\{\epsilon\}$.

$L_1^* = \{\epsilon\} \cup L_1 \cup L_1^2 \cup L_1^3 ...$

For a given string s, to decide if $s \in L_1^*$, we need only to decide if $s \in \{\epsilon\} \cup L_1 \cup L_1^2 ... \cup L_1^{len(s)}$.

Thus, we can construct a M_3 that can decide L_1^* in polynomial time.

M_3 (s)

for i=0:len(s) **do**

if $M\_ONE_i(s) == 1$ **then**

return 1

end if

end for

return 0

Problem 3. Counter (20%)

1. (8%) Increment-2^i operation

   (a)
   function INCREMENT-2^i(A, i)
   while $i < k$ and A[i] == 1 **do**
   Flip A[i].
   $i = i + 1$
   end while
   if $i < k$ **then**
   Flip A[i].
   end if
   end function

   (b) Similar to the basic increment operation, If $v = (0\overbrace{1\dots1}^{k-1})$, conducting an increment-2^0 operation requires flipping k bits. Thus, the worst-case running time is $O(k)$.

   (c) First, we simplify the problem. Because the basic increment operation can be considered as an increment-2^i operation by setting i as 0, we only need to deal with n increment-2^i operations. In the following, we use the accounting method to analyze increment-2^i operations. The idea is as same as that of the basic increment operation. We assign 2 coins to the amortized cost of each increment-2^i operation. From the pseudo code, it is clear that there is only one bit toggled from 0 to 1 in each increment-2^i operation. We spend one coin on toggling the bit from 0 to 1, and the other coin is prepaid for togging the bit from 1 to 0. By this setting, all the costs of toggling bits from 1 to 0 are prepaid. Thus, the total amortized cost is always an upper bound of the total actual cost, and the amortized cost of each operation is $O(1)$.

2. (8%) Shifting-right operation

   (a)
   function SHIFTING-RIGHT(A)
   for $i = 0$ to $k - 2$ **do**
   if A[i] $\neq$ A[$i + 1$] **then**
   Flip A[i].
   end if
   end for
   if A[$k - 1$] == 1 **then**
   Flip A[$k - 1$].
   end if
   end function

   (b) If $v = (\overbrace{10101010\dots}^{k})$, conducting a shifting-right operation requires flipping k bits. Thus, the worst-case running time is $O(k)$.

(c) We also use the accounting method to analyze shifting-right and increment operations. Assign 3 and 0 coins to the amortized costs of increment and shifting-right operations, respectively. Considering an increment operation, one coin is paid for toggling the bit from 0 to 1, another coin is prepaid for togging the bit from 1 to 0, and the other coin is prepaid for shifting-right operations. In the lecture, it has been shown that 2 coins is an eligible amortized cost of each increment operation. Thus, we only focus on shifting-right operations here. Denote the value of the counter just after the j-th operation as v_j. Assume the $(j+1)$-th operation is a shifting-right operation. If $v_j = 0$, there are no bits being toggled in the $(j+1)$-th operation. If $v_j > 0$, the maximum number of bits being toggled in the $(j+1)$-th operation is no more than $1 + \lfloor \log_2 v_j \rfloor$. By the definition of the shifting-right operation, $v_{j+1} = \left\lfloor \frac{v_j}{2} \right\rfloor$, so the difference between v_j and v_{j+1} is $v_j - \left\lfloor \frac{v_j}{2} \right\rfloor = \left\lceil \frac{v_j}{2} \right\rceil$. The coins which can be used for the $(j+1)$-th operation are from the coins released from the difference between v_j and v_{j+1} and one coin prepaid for toggling the highest bit from 1 to 0. Because $1 + \lfloor \log_2 v_j \rfloor \leq 1 + \left\lceil \frac{v_j}{2} \right\rceil, \forall v_j > 0$, the maximum number of bits being toggled is no more than the prepaid coins. Therefore, the total amortized cost is always an upper bound of the total actual cost, and the amortized cost of each operation is $O(1)$.

3. (4%) The amortized cost of each operation cannot be $O(1)$. Here is a counterexample. Conduct about $k/2$ increment-2^i operations to make $v = (\overbrace{10101010\ldots}^{k})$, then run two shifting-right and one increment-2^{k-1} operations repeatly until conducting n operations.

$$\overbrace{10101010\ldots}^{k} \rightarrow \overbrace{010101010\ldots}^{k} \rightarrow 0\overbrace{010101010\ldots}^{k-1} \rightarrow \overbrace{10101010\ldots}^{k}$$

There are $2k$ bits being flipped in each cycle consisting of two shifting-right and one increment-2^{k-1} operations. Considering $n >> k$, the total cost of flipping bits is about $\frac{n}{3} \times 2k$, so the average cost is about $\frac{2k}{3}$, not $O(1)$.

Problem 4. (16%)

1. (8%)

```
init stack {value, pos}
stack.push(array[0], 0)
maximum = 0
for i in 1 to n-1 do
    if array[i] >= stack.top then
        stack.push(array[i], i)
    else
        while stack != empty and stack.top > array[i]
            (value, position) = stack.pop()
            if maximum < (i-position)*value then
                maximum = (i-position)*value
                start = position and end = i-1
        stack.push(array[i], position)
push 0 in stack to pop all element and compute maximum like above
maximum and array[start~end] is what we want
```

Example.. array = 0 4 2 3 3 3 3 2 1 0

```
(0,0) -> (0,0)
(4,1) -> (0,0) (4,1)
(2,2) -> (0,0) (2,1) maximun = 4*1 = 4
(3,3) -> (0,0) (2,1) (3,3)
(3,4) -> (0,0) (2,1) (3,3) (3,4)
(3,5) -> (0,0) (2,1) (3,3) (3,4) (3,5)
(3,6) -> (0,0) (2,1) (3,3) (3,4) (3,5) (3,6)
(2,7) -> (0,0) (2,1) (2,7) maximum = max(4, 3*4) = 12
(1,8) -> (0,0) (1,1) maximum = max(12, 2*7) = 14
(0,9) -> (0,0) (0,1) maximum = max(14, 1*8) = 14
end -> maximum = max(14, 0*10) = 14
```

From the pesudo code and example above, we can find that in the for loop, every element is pushed in stack one time and pop out one time. we can use counting method, set the amoritized cost for push is 2 and pop is 0, then the total amortized cost = $O(2 * pushtimes)$ = $O(2 * numelements) = O(n)$ is always a upper bound of the total actual cost.

2. (8%)

First transform the matrix by $M_{i,j} = \text{sum}_{0 \le k \le i}(M_{k,j})$, then for each row do the computation like the last problem, then we can find the maximum sub-matrix from the top to each row. Compare the maximum sub-matrix on each row, we can find the maximum sub-rectangle of the entire matrix.

```
matrix[row][column] = the input
map[row][column] = transform matrix
for i in 0 to row-1 do
    for j in 0 to column-1 do
        if matrix[i][j] == 0 then map[i][j] = 0
        else map[i][j] = matrix[i][j] + (i > 0 ? matrix[i-1][j]: 0)
totalMax = 0
for i in 0 to row-1 do
    In map[i] find the maximum of the subarray based on problem 4.1
    totalMax = max(totalMax, maximum)
return totalMax
```

Take the figure of the problem as an example, we can transform the matrix

```
0 1 1 0 1   maximum = 2
0 0 2 1 0   maximum = 2
0 1 3 2 1   maximum = 4
1 0 4 3 0   maximum = 6
```

so the area of maximum seb-rectangle is 6

From the pesudo code above

time complexity = $O(mn(\text{transformation}) + m(\text{each row})n(\text{find sub-array})) = O(mn)$.

Problem 5. Multiple Sorted Arrays(18%)

1. Algorithm: Perform binary search on the sorted array A_i until we find x or we have search every array ($i = 0, 1, 2, 3, ..., k-1$).
   Worst-Case Running Time: We have to search each array once.
   Let A_m be the largest array that is not empty.
   $\sum_{i=0}^{m} \log(2^i) = \sum_{i=0}^{m} i = \frac{m(m-1)}{2} = O(m^2) = O((\log(n))^2)$

2. Algorithm: Insert an array A_0 that contains x.
   If there are two A_i in the array, merge them into A_{i+1}.
   Worst-Case Running Time: If $n = 2^m - 1$ before the insertion, we have to merge all arrays into a single A_m.
   The cost would be $\sum_{i=1}^{m} 2^i = 2^{m+1} - 2 = O(2^m) = O(n)$.
   Amortized Running Time: Consider $m = 2^p$ insertions. A_i would be constructed for at most $\lfloor \frac{2^p}{2^i} \rfloor$ times. The total cost is at most $\sum_{i=0}^{p} (\lfloor \frac{2^p}{2^i} \rfloor) 2^i \leq \sum_{i=0}^{p} 2^p = O(m \log(n))$.
   Therefore, the amortized cost is $O(\log(n))$.

3. Algorithm: To delete $A_i[j]$, we can find the smallest array A_m that is not empty. If $m = i$, delete $A_i[j]$ from A_i split the rest of the array into $A_0, A_1, A_2, A_3, A_4, ..., A_{m-1}$. If $m < i$, split A_m into $A_0, A_1, A_2, A_3, A_4, ..., A_{m-1}$ and an additional A_0. We put the additional A_0 into A_i to replace the deleted element $A_i[j]$ and sort A_i.