B03902062 資工二 董文捷

Problem 1

1.

To calculate C_m^n, we can use Pascal's triangle. For the n-th item, we can choose it (C_{m-1}^{n-1}) or don't choose it (C_m^{n-1}), so $C_m^n = C_{m-1}^{n-1} + C_m^{n-1}$ Build a two dimensional array to store all C_m^n by this equation. To build the array, for n = 0 ~ N, find the value of C_m^n for all m ≦ n by C_{m-1}^{n-1} and C_m^{n-1}, notice for m = 0 and m = n, the value should be set to 1.

2.

n > m

f(n, m) = f(n, m – 1) + f(n – 1, m)

n = m

f(n, m) = f(n, m – 1)

n < m

f(n, m) = 0

Use the recurrence relation to build a two dimensional array to store all f(n, m), and f(n, n) is the C_n we want. To build the array, for n = 0 ~ N, find the value of f(n, m) for all m ≦ n by f(n, m – 1) and f(n – 1, m) (since value for n < m is not used, we need not to deal with them), notice for m = 0, the value should be set to 1, and for m = n, the value should be set to f(n, m – 1)

3.

f(n, m) = $\boldsymbol{\sum_{k=0}^{k\_max} f(n - k * m,\ \ m - 1)}$ where k_max is the max k satisfying $n - k * m \geqq 0$. For f(n, m), there can be 0 ~ k_max group of size m, we can get different partitions by increasing the number of size m group and partition n – k * m into groups of size at most m – 1.

To build up a two dimensional array to store f(n, m), initialize all f(0, x) with 1 (x ≤ N), then for n = 1 ~ N, set f(n, 0) to 0, choose m from 1 to N, get f(n, m) by the above equation. f(n, n) is the P_n we want.

Time complexity:

To get every value in the table, we need to find at most k_max values to sum up, while k_max ≦ n_max / m_min = N

So T(N) ≦ (N * N) * N = O(N^3)

Problem 2

1. The same as 2.2. (O(n) is also $O(n^2)$)

2. To calculate the number, I use two variables **end_0** and **end_1** to record the distinct subsequences number end with 0 and 1. Set end_0 and end_1 to 0 initially. Run a for loop to increase length from 1 to N. Each time a new character is added, for example a new character 1 is added to 101 and new string is 1011. **end_0** will not be adjust since the new added character is '1', **end_1** will be adjust to **end_0 + end_1 + 1**. The meaning of end_0 + end_1 is to add new '1' to every subsequence, since sequences in end_0 and end_1 are distinct, after adding '1', they will be distinct, too. The meaning of 1 is '1'. We need not to consider subsequences end with 1 and do not add the new '1' because if S'1' is a subsequence, S is also a subsequence, we can add the new '1' to let it become S'1', so actually it is already counted in case adding '1'. (101 is a subsequence of 101, we can also form it by adding '1' to 10, which is also a subsequence of 101) The only exception is '1' (single character), since empty is not a subsequence, '1' is not counted in case adding new '1'.

```
end_0 = 0
end_1 = 0
for i = 0 ~ ( N – 1 )
    If string[ i ] = '1'
        end_1 = end_0 + end_1 + 1
    else
        end_0 = end_0 + end_1 + 1
ans = end_0 + end_1
```

Time complexity **: O(n)**

3. Use two array end_0[K + 1] and end_1[K + 1] to record number of distinct subsequence end with 0 and 1 of length 1 ~ K. Like 2.2, if a new '1' is added, we can get end_1[length + 1] by add the new '1' to end_0[length] and end_1[length] (end_1[length + 1] = end_0[length] + end_1[length]), and don't need to adjust end_0 array. '1' is also a special case, so end_1[1] should be set to 1. Similarly, if '0' is added, we can get end_0[length + 1] by add the new '0' to end_0[length] and end_1[length], and don't need to adjust end_1 array. end_0[1] should be set to 1. By running through the string from 0 to (N – 1), we can find out end_0[K] and end_1[K], the sum of them is the final answer.

Problem 3

(1)

1. To calculate A^n, we can use divide and conquer to divide $\mathbf{A^n}$ into $\mathbf{A^{n/2}} * \mathbf{A^{n/2}}$, $T(n) = T(n / 2) + m^3$ (matrix multiplying), so $T(n) = O(m^3\log_2 n)$

By mathematical induction, we can find that if $M = \begin{bmatrix} A & I \\ 0 & I \end{bmatrix}$

$\mathbf{M^n} = \begin{bmatrix} A^n & \sum_{i=0}^{n-1} A^i \\ 0 & I \end{bmatrix}$, to calculate $\sum_{i=0}^{n} A^i$, we only need to calculate $\mathbf{M^{n+1}}$ and can use divide and conquer, too. $T(n) = T(n / 2) + 8\ m^3$

$T(n) = O(m^3\log_2 n)$

2.

Set a K * K matrix A and initialize it value with 1, for Q pairs can not appear consecutively, set both $A(\beta_{i1}, \beta_{i2})$ and $A(\beta_{i2}, \beta_{i1})$ to 0. $\mathbf{dp_T(L)}$ represents number of permutation of length L end with $\mathbf{n_T}$.

$$\begin{bmatrix} dp_1(L) \\ dp_2(L) \\ \vdots \\ dp_{n-1}(L) \\ dp_n(L) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 1 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & 0 & \cdots & 1 & 1 \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} \begin{bmatrix} dp_1(L-1) \\ dp_2(L-1) \\ \vdots \\ dp_{n-1}(L-1) \\ dp_n(L-1) \end{bmatrix}$$

For $\mathbf{dp_1(L)}$, if n_1 can not appear consecutively with n_2, we can add n_1 to $dp_1(L\text{-}1)$ $dp_3(L\text{-}1)$ …… $dp_n(L\text{-}1)$ to get it,
so $dp_1(L) = dp_1(L\text{-}1) + dp_3(L\text{-}1) + \dots\dots + dp_n(L\text{-}1)$

Sort a_j and swap b_j simultaneously, then we can divide the DNA sequence to P + 1 subsequences with the P fixed position, notice that the last sequence length may vary from $L – a_P$ to $R – a_P$

Fill the P subsequence by the order. For the first subsequence, set $dp_1(0)$ $dp_2(0)$ … to 1, and multiply dp column matrix with A^{a1} to get $dp_1(a_1)$ $dp_2(a_1)$…… Since a_1 is set to b_1, only $dp(a_1)$ end with b_1 is remained, other impossible $dp(a_1)$ should be set to 0. For other subsequence, multiply dp column matrix with $A^{an-a(n-1)}$ to get $dp(a_n)$, only keep $dp(a_n)$ which end with b_n and set other impossible $dp(a_n)$ to 0.
Finally, the sequence's length may vary from L to R, so we need to multiply dp column matrix with $(A^{L-ap} + A^{L-ap+1} + \dots\dots + A^{R-ap})$, the final answer is $dp_1(R) + dp_2(R) + \dots\dots + dp_n(R)$

Time complexity:
Set matrix : O(Q)
Sort a : O(PlogP)
Fill all subsequence to get ans:
($O(K^3\log_2 R) + O(K)$) * P ($a_n - a_{n-1} < R$) (The last matrix sum can also be get in $O(K^3\log_2 R)$ by 3.1.1) (O(K) is the time need to set impossible dp to 0)
Total:
$O(K^3 P \log_2 R)$
(2)

1. x(j, k) is the intersect point of $f_j(x)$ and $f_k(x)$, so x(j, k) = $\frac{b_k - b_j}{a_j - a_k}$

2. At first, α_0 is set to $-\infty$ and α_1 is set to ∞, j_1 is set to 0, dp(1) = $f_0(x_0)$. Each time when adding a new line f_n, move ∞ to the rightmost α, find the intersect point of f_n and f j_n , α_n, and j_{n+1} is set to n, push_back them into the deque. If α_n is smaller than α_{n-1}, we need not to keep f j_{n-1} anymore, so we can swap α_n α_{n-1}, j_n j_{n-1}, and then pop_back out α_n j_n (∞ should be kept carefully), update α_{n-1} to be intersect point of f_n and f j_{n-1}. If α_{n-1} is still smaller than α_{n-2}, repeat the process to pop_back out unused line until α become the biggest number. We can find dp(n + 1) by the maintained deque and a variable recording choosing position of dp(n), if $x_{n+1} > \alpha$, $f_n(x_{n+1})$ is the best answer, else the position should be equal or become bigger, find $\alpha_{P-1} \leqq x_{n+1} \leqq \alpha_P$ and dp(n + 1) = $f_P(x_{n+1})$
Time complexity :
Build deque :
A line will be insert and delete at most one time, so the time complexity is O(n)
Find dp(i) :
α and j will be pop out and traverse at most one time, so the time complexity is O(n)
Total :
O(n)
3.

Reference: b03902044 b03902061