NTUCSIE ADA 2015
Evil Homework 2
B03902027 王冠鈞

Problem 1

(0) References:

i. *"Catalan number"*, on Wikipedia.
https://en.wikipedia.org/wiki/Catalan_number

ii. *"Partition (number theory)"*, on Wikipedia.
https://en.wikipedia.org/wiki/Partition_%28number_theory%29

(1) Binomial Numbers

Solution:

Apply Pascal's rule, saying that:

$$\binom{\mathrm{n}-1}{\mathrm{k}}+\binom{\mathrm{n}-1}{\mathrm{k}-1}=\binom{n}{k}, for\ 1 \leq k \leq n,$$

and the famous Pascal's triangle is the result.

The Pascal's triangle can be regarded as an triangular array $\mathrm{T[i][j]}\ (0 \leq \mathrm{j} \leq \mathrm{i})$; for all $\mathrm{T}[i][0]$ and $\mathrm{T[i][i]}$, which represent $\binom{\mathrm{i}}{0}, \binom{\mathrm{i}}{\mathrm{i}}$ respectively, are 0. The other indices can be calculated via Pascal's rule by a single addition, taking $\mathrm{O}(1)$ time. Thus, the total time will be:

$$1+2+3+\cdots+(\mathrm{n}+1)=\frac{(n+2)n}{2}=\mathrm{O}(\mathrm{n}^2)$$

(2) Catalan Numbers

Solution:

The Catalan numbers satisfy the following recurrence:

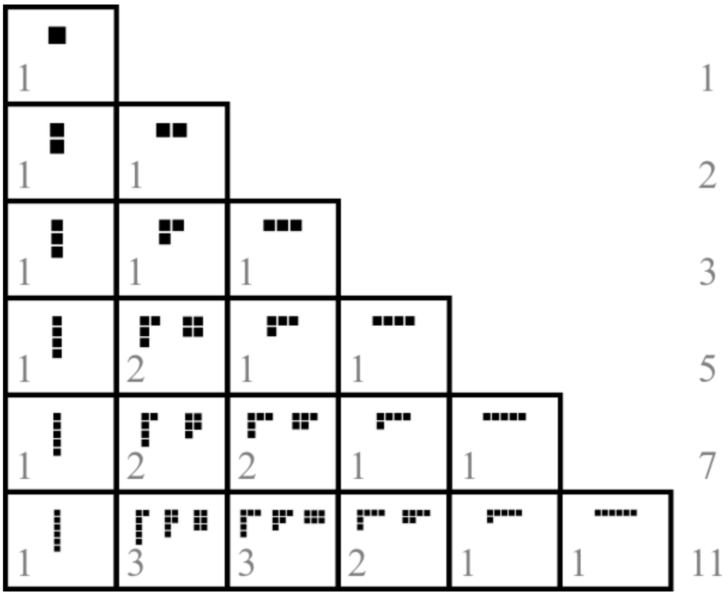
$$\mathrm{C}_0=1, C_{n+1}=\sum_{i=0}^{n} C_i\, C_{n-i}$$

, according to web references. Thus, we can apply such formula to calculate all Catalan numbers in $\mathrm{O}(\mathrm{n}^2)$ time. That is, we need $\mathrm{O}(\mathrm{n})$ time to calculate every Catalan number, and it's obvious that why we need such time to calculate every number by observing the recurrence relation above.

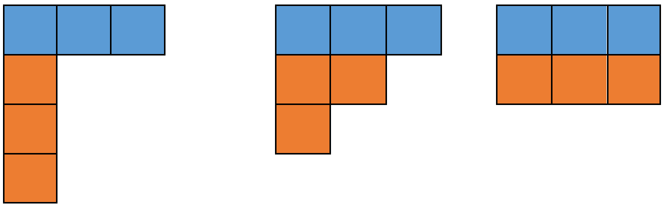
(3) Partition Numbers

Solution:

The hints don't arise me with any inspirations, but the following figure from the web resources does:



The rightmost numbers are partition numbers $P_1 \sim P_6$, and every cell illustrates the partition graphs with a maximum addend. By observing every cell, we can find out that if we remove the upmost row of every graph, the remaining graphs are somehow identical to some of the original graphs in the cells above. For instance, if we remove the upmost rows of the graphs in the 5th, 4th, 3rd cells of P_6, the remaining graphs are identical to all the graphs of P_1, P_2, P_3.



When we remove the blue parts of the graph, the remaining red parts constitute P_3.

Let $F(n, m)$ $(n \geq 1, 1 \leq m \leq n)$ be the number of the graphs in P_n with at least one maximum addend m. Then according to the observations above, we can induct the relations of $F(n, m)$:

$$F(n, m) = \begin{cases} 1, if\ n = m \\ \sum_{i=1}^{\min(m, n-m)} F(n-m, i) \end{cases}$$

, and we can calculate P_n by adding all components, that is:

$$P_n = \sum_{i=1}^{n} F(n, i)$$

We need a loop to calculate every $F(i, j)$, a loop for P_i, and a loop for P_1 to P_n. Thus, in total, we need roughly $O(n^3)$ time to finish this task.

Problem 2

(0) References

i. *"how to find the number of distinct subsequences of a string?"*, on stackoverflow.com. http://stackoverflow.com/questions/5151483/how-to-find-the-number-of-distinct-subsequences-of-a-string

ii. Taught by B03902028

(1) Hidden HH-Code

I've got an algorithm of $O(n)$ time, so this sub-problem will be omitted.

(2) Hidden HH-Code Fast

Solution:

#This algorithm is derived from reference *i.*

We define something first.

a. $\mathrm{dp(i)} = \text{number of distinct subsequences ending with a[i]}$.

b. $\mathrm{sum(i)} = \sum_{j=0}^{i} dp(i)$

c. $\mathrm{last(0), last(1)} = last\ position\ of\ 0/1\ in\ the\ string$.

And define $\mathrm{dp(0)} = 1$, and $\mathrm{sum(0)} = 1; \text{if } \mathrm{k} < 0, \mathrm{sum(k)} = 0$

The basic idea of this algorithm is that for any $\mathrm{dp(i)}$, it can be regarded as appending the current code to all the strings in $\mathrm{sum(i-1)}$; however, there's a problem that after this process the subsequences might no longer be distinct again. That's the reason that we need to record the last position that a character appeared. The pseudocode is below:

```
Algorithm (string a)
n = strlen(a)
for i = 1 to n
set last(0), last(1) to 0
    dp[i] = sum[i - 1] - sum[last(a[i]) - 1]
    sum[i] = sum[i - 1] + dp[i]
    last(a[i]) = i
return sum[n] - 1 //exclude null string
```

We can build a table to make the process look more clearly.

	dp	sum	last0	last1
0(null)	1	1	0	0
1 (1)	1	2	0	0
2 (0)	2	4	0	1
3 (1)	3	7	2	1
4 (1)	3	10	2	3

(3) Hidden HH-Code with length

This algorithm is derived from B03902028 (or maybe others).

We need another "Hidden HH-Code Fast" algorithm first. Now we use three memories to store ans, #0, #1; the meaning of #a is that the potential number of strings that will be added to ans if the next character is a. The idea of the algorithm is that for every step, we can append 0 or 1 to all the newly-generated subsequences. For instance, we can append 0 or 1 to a null string only in initial state, because we haven't generated any new subsequences; if we newly generated 0, 10, we can append 0 or 1 on them and become 00, 100 or 01, 101. The example of 1011 is written below:

	null	1	0	1	1
Ans	0	1	3	6	9
		"1"	add: "0", "10"	add: "11", "01", "101"	add: "110", "010", "1010"
#0	1	2	2	5	8
	denotes: "0"	"0", from left "10", newly generated	"00", "100"	"00", "100", "110", "010", "1010"	"00", "100", "110", "010", "1010", "1100", "0100", "10100"
#1	1	1	3	3	3
	"1"	"11", newly generated	"11", "01", "101"	"110", "010", "1010"	"1101", "0101", "10101"

Pseudocode:

```
Algorithm (string a)
ans = 0, #1 = 1, #0 = 1
for i = 1 to strlen(a)
    if a[i] = 1
        ans = ans + #1
        #0 = #0 + #1
    else
        ans = ans + #0
        #1 = #1 + #0
return ans
```

And for certain length K, we need to modify the algorithm by making the storage from a single memory into an array, and for n^{th} index of the array, we store the numbers or length n. Take the example of 1011 again:

#(a, b, c, d) = (subseq length 4, length 3, length 2, length 1)

		null	1	0	1	1
Ans		0=(0, 0, 0, 0)	1=(0, 0, 0, 1)	3=(0, 0, 1, 2)	6=(0, 1, 3, 2)	9=(1, 3, 3, 2)
			"1"	add: "0", "10"	add: "11", "01", "101"	add: "110", "010", "1010"
#0		1=(0, 0, 0, 1)	2=(0, 0, 1, 1)	2=(0, 1, 1, 0)	5=(1, 3, 1, 0)	8=((1,) 3, 3, 1, 0)
		denotes: "0"	"0", from left "10", newly generated	"00", "100"	"00", "100", "110", "010", "1010"	"00", "100", "110", "010", "1010", "1100", "0100", "10100"
#1		1=(0, 0, 0, 1)	1=(0, 0, 1, 0)	3=(0, 1, 2, 0)	3=(1, 2, 0, 0)	3=((1,) 2, 0, 0, 0)
		denotes: "1"	"11", newly generated	"11", "01", "101"	"110", "010", "1010"	"1101", "0101", "10101"

Pseudocode:

```
Algorithm (string a, int k)
n = strlen(a)
ans[n] = {0}
#0[1] = 1
#1[1] = 1
for i = 1 to n
    if a[i] = 1
        add indices from #1 to ans
        move all #1[j] to #1[j + 1]
        add indices from #1 to #0
    else
        add indices from #0 to ans
        move all #0[j] to #0[j + 1]
        add indices from #0 to #1
return ans[k]
```

Problem 3

(0) References

(1) Fast Matrix Exponentiation

1. We have a $\mathrm{m} \times \mathrm{m}$ matrix, so it's equivalent to multiplying $\mathrm{m}^2\ \mathrm{m} - \mathrm{tuple}\ \times \mathrm{m} - \mathrm{tuple}$. It takes $\mathrm{O}(\mathrm{m}^3)$ time. That means we need to do the exponentiation and addition of matrices in logarithmic time.
   For the first one, A^{n}, we can do it by fast exponentiation using recursion.
   The pseudocode is in below.
   ```
   Algorithm: Exp(Matrix A, n)
   if n = 0
    return I
   else if n = 1
    return A
   else if n is even
    return Exp(A * A, n / 2)
   else
    return A * Exp(A * A, (n - 1) / 2)
   ```
   The second one, $\sum_{i=0}^{n} A^i$, can be calculated in $\mathrm{O}(\mathrm{m}^3 \log_2 n)$ time by using the hint:, because $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}^n = \begin{bmatrix} A^n & \sum_{i=0}^{n} A^i \\ 0 & I \end{bmatrix}$. Then we can use the algorithm above by just replacing A to $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}$, and get the result of index $(1, 2)$.
2. For properties (a) and (b), we can use a $\mathrm{K} \times \mathrm{K}$ matrix A, and for term $\mathrm{a}_{\mathrm{i,j}}$, it represents appending nucleotide $\mathrm{n_i}$ to $\mathrm{n_j}$: if such action is available, $\mathrm{a}_{\mathrm{i,j}} = 1$; if $\mathrm{n_i}$ cannot be adjacent to $\mathrm{n_j}$ due to the restrictions of (b), $\mathrm{a}_{\mathrm{i,j}} = 0$.
   For the first matrix A_1, since the first nucleotide doesn't append to any other nucleotide, all the terms of A_1 will be 1.
   For the $\mathrm{a_j}\ th$ matrix $\mathrm{A}_{\mathrm{a_j}}$, due to property (c), only terms in column j, (that is, term $\mathrm{a}_{\mathrm{b_j},i}, 1 \leq \mathrm{i} \leq \mathrm{k})$ will contain 1s (and of course, some 0s due to (b)).
   Next, we can use a m by 1 matrix $\mathrm{B_i}$, and for every term $\mathrm{b}_{\mathrm{a},1}$, it represents the possible number of DNA sequences with length i and with nucleotide a appended to the last, and we define it to be $\mathrm{dp}(\mathrm{a}, \mathrm{i})$ here.
   Then we can get:

$$\begin{pmatrix} dp(n_1, l) \\ dp(n_2, l) \\ \vdots \\ dp(n_{K-1}, l) \\ dp(n_K, l) \end{pmatrix} = A_l \dots A^{a_P} \dots A^{a_2} A^{a_2 - a_1 - 1} A_{a_1} A^{a_1 - 2} A_1 \begin{pmatrix} dp(n_1, 0) \\ dp(n_2, 0) \\ \vdots \\ dp(n_{K-1}, 0) \\ dp(n_K, 0) \end{pmatrix}$$

$$\text{Ans} = \sum_{\lambda=L}^{R} \sum_{i=1}^{K} dp(n_i, \lambda)$$

We can use $\mathrm{O}(\mathrm{K}^3 \mathrm{P} \log_2 R)$ time to calculate $\sum_{i=1}^{l} A_i$ by using fast matrix exponentiation in fragments of those same, sequential matrices. And for counting from L to R, we just use the property proven in the previous sub-problem to calculate something like $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}^R - \begin{bmatrix} A & I \\ 0 & I \end{bmatrix}^{L-1}$, and we can get all the terms we want in the time complexity aforementioned.

(2) Convex Hull Optimization

1. trivial.

$$\mathrm{f_j}(\bar{x}) = a_j \bar{x} + b_j = f_k(\bar{x}) = a_k \bar{x} + b_k$$
$$\Longrightarrow \bar{x} = -\frac{b_j - b_k}{a_j - a_k}$$

2.
3.