Problem 1

Binomial Numbers

Since the problem requires an $O(n^2)$ solution, we'll go for the recursive formula. For all integers n, k, when $1 \leq k \leq n-1$,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \tag{1}$$

with boudary values

$$\binom{n}{0} = \binom{n}{n} = 1 \tag{2}$$

for all integers $n \geq 0$.

Algorithm 1: BINOMIAL(n, m) finds the m combinations among n items.

Input: Two non-negative integers n and m, which satisfy $0 \leq m \leq n \leq N$.
Output: The binomial coefficient of n-choose-m.

```
if m = 0 then
    return 1
else
    return BINOMIAL(n − 1, m − 1) + BINOMIAL(n − 1, m)
```

Catalan Numbers

Catalan numbers can be derived as

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \tag{3}$$

for all $n \geq 0$. If we now write the Catalan numbers into recurrence relation,

$$\begin{cases} C_0 = 1 \\ C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \end{cases} \tag{4}$$

for all $n \geq 0$ as well.

From the recurrence relation, without the use of dynamic programming, the time complexity of it is

$$T(n) = \sum_{i=0}^{n} T(i)T(n-i) \tag{5}$$

which is clearly in exponential form. Due to the fact that the recursive implementation generates lots of repeated work, there are overlapping subproblems.

Problem 1 continued on next page...

Algorithm 2: CATALAN(n) finds the nth Catalan number.

Input: A non-negative integer n.
Output: The nth Catalan number C_n, meaning that there are C_n ways to go from $(0, 0)$ to (n, n).

```
storage[n+1] initialization
storage[0] ← 0
storage[1] ← 0
for i ← 2 to n do
    storage[i] ← 0
    for j ← 0 to i − 1 do
        storage[i] ← storage[j] · storage[i-j-1]
return storage[n]
```

The algorithm *roughly* shows us that both loops require n iterations, which leads to the time complexity of $O(n^2)$.

Partition Numbers

The original P_n can operate on numerous numbers, meaning that P_n is the number of different sequences $\{a_1, a_2, \cdots, a_k\}$, such that $k \geq 1$, $a_1 \geq a_2 \geq \cdots \geq a_k$ and $a_1 + a_2 + \cdots + a_k = n$.

We take the first hint, but write $f(n, m)$ for the numbers of ways to partition n into exactly m groups, instead of groups of size at most m.

$$P_m = f(n, m) + f(n, m-1) + \cdots + f(n, 1) + f(n, 0) \tag{6}$$

By observing the P_5 sequence, we can find out that

$$f(n, m) = f(n-1, m-1) + f(n-m, m) \tag{7}$$

because of that fact that when we are partitioning n identical objects into m parts, two scenarios come up:

1. There is a group with one item only, therefore, only $n-1$ items left, and we still need to partition them into $m-1$ groups. Denotes as $f(n-1, m-1)$.
2. Not a group contains single item, so if we *every* group has 1 item first, *then* partition rest of the items, which means that we takes out m items first, leaving $n-m$ items to partition. Denotes as $f(n-m, m)$.

We also have the initial condition that $\forall n, f(n, n) = f(n, 1) = 1$, and $f(n, m) = 0$ if $m < 1$ or $m > n$. Which indicates that we can calculate the partition number using recurrence relations.

Problem 1 continued on next page...

Algorithm 3: PARTITION(n) finds P_n ways to partition n *identical* things into groups.

Input: A non-negative integer n.
Output: The partition number P_n.

```
1  storage[n, n] initialization
2  storage[n, n] ← 1
3  storage[n, 1] ← 1
4  for i ← 1 to n do
5      for j ← 1 to n do
6          if j >= i then
7              storage[i, j] ← 0
8          else
9              storage[i, j] ← storage[i-1, j-1] + storage[i-j, j]
10 return storage[n]
```

To calculate P_n, the algorithm proposed above takes $O(n^2)$ to finish it. But this is slightly different than the P_n defined in the problem – it needs to sum up all the P_i, for all $1 \leq i \leq n$. In this case, when $n = N$, it means that it has $O(N^3)$ (each case is an $O(n^2)$ while there are N calculations to repeat).

Problem 2

The naive way to find the hidden HH-codes is to permute all of them, which leads to $O(2^n)$.

Hidden HH-Code

In order to complete the search in $O(n^2)$, we can try to utilise all the *found* HH-code, meaning we can append to previously found HH-code, in order to generate a new one.

Algorithm 4: HH-CODE(H, N) finds the number of different hidden HH-code.

Input: H is the HH-code of interest, while N is the length of H.
Output: Total number of different hidden HH-code.

```
1  sp ← 1
2  cnt ← 1
3  buf[cnt] ← first character in H
4  foreach h in H do
5      skip if h is the first character
6      end ← cnt
7      tuck h in buf if it's the first 1 or 0
8      if h == 1 then
9          start ← sp
10         sp ← cnt + 1
11     else
12         start ← 1
13     for i ← start to end do
14         cnt ← cnt + 1
15         buf[cnt] ← buf[i] appends h
16 return cnt
```

The *sp* in the algorithm holds the last position of the first element generated by 1, while *cnt* holds the total amount of hidden HH-code found. The following description is the step-by-step explanation of the algorithm, using $H = 1011$ as an example:

Read 1
$sp : 1, cnt : 1, buf : [1]$

Read 0
$sp : 1, cnt : 2, buf : [1, 0]$
$sp : 1, cnt : 3, buf : [1, 0, 10]$

Read 1

Problem 2 continued on next page...

$sp : 1, cnt : 6, buf : [\mathbf{1}, 0, 10, 11, 01, 101]$
$sp : 4, cnt : 6, buf : [1, 0, 10, \mathbf{11}, 01, 101]$

Read 1
$sp : 4, cnt : 9, buf : [1, 0, 10, \mathbf{11}, 01, 101, 111, 011, 1011]$
$sp : 7, cnt : 9, buf : [1, 0, 10, 11, 01, 101, \mathbf{111}, 011, 1011]$

End, Return 9

In each cycle of the **for-each** clause, there is at most one loop running, and total cycle needed for the inner-loop is at most N. The outer-loop undoubtedly has to loop for each character, which is N cycle as well. Therefore, the time complexity is $O(N^2)$.

Hidden HH-Code Fast

The hint tells us to utilise the problem with partial sum, if we think of the 1s and 0s in the HH-code as an instruction of the increment, it is possible to count the result without going through the sequence generation itself.

Algorithm 5: HH-Code-Hidden(H, N) finds different hidden HH-code in $O(N)$.

Input: H is the HH-code of interest, while N is the length of H.
Output: Total number of different hidden HH-code.

```
1  sp ← 1
2  cnt ← 1
3  foreach h in H do
4      skip if h is the first character
5      end ← cnt
6      tuck h in buf if it's the first 1 or 0
7      if h == 1 then
8          start ← sp
9          sp ← cnt + 1
10     else
11         start ← 1
12     cnt ← cnt + (end − start + 1)
13 return cnt
```

Since we eliminate the **for-loop** clause, we quickly reduce the time complexity from $O(N^2)$ to $O(N)$, but it comes with the price that we won't be able to know *what* are the hidden HH-codes.

Hidden HH-Code with length

Problem 2 continued on next page...

In order to find the hidden HH-codes, we perform a series of iterative moves. Instead of cycling through all the possible combinations, we cycle till our desired length K, therefore the complexity can be reduced from $O(N^2)$ to $O(KN)$.

Algorithm 6: HH-CODE-HIDDEN(H, N, K) finds different hidden HH-code with specific length K.

Input: H is the HH-code of interest, while N is the length of H.
Output: Total number of different hidden HH-code.

```
1  sp ← 1
2  cnt ← 1
3  buf[cnt] ← first character in H
4  foreach h in H do
5      skip if h is the first character
6      end ← cnt
7      tuck h in buf if it's the first 1 or 0
8      if h == 1 then
9          start ← sp
10         sp ← cnt + 1
11     else
12         start ← 1
13     for i ← start to end ∩ buf[cnt].length ≤ K do
14         cnt ← cnt + 1
15         buf[cnt] ← buf[i] appends h
16 return cnt
```

Problem 3

Fast matrix exponentiation

1. Consider A^2 for an $m \times m$ matrix A, it needs m multiplications and $m-1$ additions. Since there are m^2 elements in the matrix, the entire computation costs m^3 operations.

   When we try to computer A^n, the computation needs at most $log_2 n$ operations, since we can multiply the intermediate result together, like the following illustration for 2^4:

   $$2 \times 2 = 4$$
   $$4 \times 4 = (2 \times 2) \times (2 \times 2) = 16$$

   instead of 4 operations, only 2 are needed. Therefore, it's easy to see that it can be calculated in $O(m^2 log_2 n)$ time. □

   As for the $\sum_{i=0}^{n} A^i$, we take the hint from the problem statement, when multiplying matrices of the form $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}$,

   $$\begin{aligned} M^3 = \begin{bmatrix} A & I \\ 0 & I \end{bmatrix}^3 &= \begin{bmatrix} A^2 & A \\ 0 & I \end{bmatrix} \begin{bmatrix} A & I \\ 0 & I \end{bmatrix} \\ &= \begin{bmatrix} A^3 & A^2 + A \\ 0 & I \end{bmatrix} \end{aligned} \tag{8}$$

   when the matrix M is multiplied by $n+1$ times, it can calculate $\sum_{i=0}^{n} A^i$, the result is located at M_{12}^{n+1}.

   Observing the square of the matrix $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}$, it takes a constant operation counts of 5 (because multiply by 0 can be ignored). Since the elements of this matrix are matrices, there are $log_2 n + 1 \approx log_2 n$ matrix operations need to perform. We know that each matrix multiplication needs m^3 operations, while a simple addition needs m^2, the time complexity is can be written as $O(m^3 log_2 n) + O(m^2 log_2 n) + O(m^2) = O(m^3 log_2 n)$. □

   Notes: The $O(m^2)$ at the end is due to the fact that multiplying $\begin{bmatrix} A & I \\ 0 & I \end{bmatrix}$ gives us $\sum_{i=1}^{n} A^i$, but the problem requires $\sum_{i=\mathbf{0}}^{n} A^i$.

2. We can think of it as a variant of Markov chain, the relationship between *current* nucleotide and the consecutive nucleotide can form a transition matrix. When the length of the DNA sequence λ satisfies $max(a_i) \leq L \leq \lambda \leq R$, it means that there are at

most R nucleotides.

From the deduction above, we can know that, there exists at most $R-1$ times of multiplications between transition matrices. Using the result from (1), and knowing that there are K different nucleotides, the possible combinations of the sequence can be calculated in $O(K^3 log_2 R)$.

The problem states that there are P immutable positions, which means that we have to take them into consideration *exclusively*, hence, we have to multiple P during our calculations, since each possible immutable position can create a certain category of the DNA sequence. The time complexity now denotes as $O(K^3 P log_2 R)$.

Algorithm 7: DNA-Sequence(Γ_T, λ) calculates the possible outcomes.

Input: Γ_T is the transition matrix of the nucleotides.
Input: λ is the length of the sequence.
Output: Formulated transition matrix after λ nucleotides.

```
1  x1 ← Γ_T
2  x2 ← Γ_T^2
3  for i ← λ − 2 to 0 do
4      if n_i = 0 then
5          x2 ← x1 x2
6          x1 ← x1^2
7      else
8          x1 ← x1 x2
9          x2 ← x2^2
10 return x1
```

Note: This is basically the Montgomery's ladder technique, referenced from the Wikipedia. We can simply formulate the possible combinations from the transition matrix by solving the transient state of it.

When the length of the DNA sequence is located between L and R, we can first calculate DNA-Sequence(Γ_T, R), then subtract it with DNA-Sequence(Γ_T, L). The time complexity of the operation is $O(K^2)$.

The different immutable positions of P, can generates at most P transition matrix, denotes as Γ_{Ti}, to calculate all of them simply have to call DNA-Sequence(Γ_T, λ) P times.

Problem 3 continued on next page...

Formulate the above result, the algorithm itself is $O(K^3 log_2 \lambda)$, consider the worst case scenario, which is of length R, it turns into $O(K^3 log_2 R)$. Though we have to subtract the lower boundary, but the **Big-O** won't change. At last, we consider the immutable positions, which further increase the time complexity to $O(K^3 P log_2 R)$. □

Convex hull optimisation

1. Basically, $\bar{x}(j,k)$ represents the intersection between $f_j(x)$ and $f_k(x)$. Therefore,

$$f_j(x) = a_j x + b_j = a_k x + b_k = f_k(x) \tag{9}$$

by solving this equation, we can find out that

$$\bar{x}(j,k) = x = -\frac{b_k - b_j}{a_k - a_j} \tag{10}$$

2. According to the notation, $f_{ji}(x) = a_{ji}x + b_{ji}$, a_{ji} and x we can then implement the dynamic programming by filling up the array, updating the cost function on the run and reach the result at last.

Algorithm 8: DP(N) calculates $dp(i) = \min_{j<i} a_j x_i + b_j$.

Input: Γ_T is the transition matrix of the nucleotides.
Input: λ is the length of the sequence.
Output: Formulated transition matrix after λ nucleotides.

```
1 E initialisation
2 cost[0] ← 0
3 Add y = mx + b to E, where m = a[i] and b = cost[0]
4 for idx ← 1 to N do
5 |   cost[i] ← E[x[i]]
6 |   if i < N then
7 |   |   Add y = mx + b to E, where m = a[i] and b = cost[i]
8 return cost[N]
```

The array E mentioned in the algorithm, should have the newly added line sorted by it's slope m. Since this is done during each insertion, the step that update the cost according to previous cost through b won't have to perform linear search, which is the ultimate reason that it can calculate the table in $O(n)$.

Note: Algorithm credits to the case study of **USACO MAR08 acquire** *enlisted on the PEGWiki, part of the website under Woburn C.I. Programming Enrichment Group.*

Problem 3 continued on next page...

3. The main issue of this problem, is enlisting the cost function. In this case

$$f_i(x) = a_i(x) + b_i \tag{11}$$

with f_i being the cost of the specific setting of the fish, while x indicates the position of interest (on the line $0 \leq x \leq N$), a_i represents how many fish at that specific point x, and b_i is actually the c_i mentioned in the problem (the physical power required for Manaka).

We know that we can formulate the required physical power for both Manaka and the fish, and we know that we want the required physical power to be minimal, then the algorithm described above is in our favour. Simply filling out the table can get the job done, the maximum size of the ocean is n, meaning it takes $O(n)$ to finish the task.

Quadrangle inequality optimisation
Divide and conquer
I can barely finish the required problems... sorry :(