Problem A: Special Buffet

Solution:

C++	https://ideone.com/FEVqBE
Java	https://ideone.com/ZUyY3Y
Python	https://ideone.com/Khxf8r

Tóm tắt đề:

Nhà hàng có n loại thức ăn (đánh số 1 đến n), mỗi loại có k phần. Có m vị trí để đặt món ăn. nên số lượng loại thức ăn sẽ ít nhất bằng số lượng vị trí trưng bày, nên các món ăn trưng bày đó sẽ không bao giờ trùng nhau.

Buổi tiệc sẽ bắt đầu khi các vị trí trưng bày các món ăn đã được lắp đầy bởi các loại thức ăn. Và có một điều đặc biệt trong buổi tiệc này, vì mọi người đều dễ thương và tinh tế nên khi chọn thức ăn thì các bạn không chọn theo sở thích của mình mà chọn món đã **được trưng bày lâu nhất** để tránh để thức ăn bị nguội hoặc mất ngon. Để đảm bảo các vị trí đặt món ăn luôn luôn được lắp đầy thì nhân viên sẽ đem ngay món khác thay thế khi một món tại vị trí nào đó được lấy đi.

Biết rằng các loại thức ăn sẽ được đem ra theo thứ tự của nó và xoay vòng cho đến khi không còn nữa (1 → 2 → 3 → … → n → 1 → 2 → …). Và các món ăn được lấy đi từ vị trí trưng bày là lần lượt.

Input:

Dòng đầu tiên chứa T ($1 \le T \le 1000$) là số lượng bộ dữ liệu.

T dòng tiếp theo, mỗi dòng chứa thông tin về một bộ dữ liệu gồm ba số nguyên n, m, k ($1 \le m \le n \le 500$, $1 \le k \le 10^5$) – lần lượt là số loại thức ăn có trong buổi tiệc, số lượng vị trí đặt thức ăn và số lượng mỗi loại thức ăn.

Output:

Gồm T dòng, mỗi dòng là một số nguyên duy nhất, số lần thay thế đĩa thức ăn này bằng một đĩa thức ăn khác trong các vị trí trưng bày.

Ví dụ:

1 4 3 2	5

Giải thích ví dụ:

Chúng ta có 4 loại thức ăn, có 3 vị trí để trưng bày thức ăn và mỗi loại thức ăn gồm 2 phần.

Ban đầu, thức ăn còn đang trong quá trình chế biến nên chưa có món nào được trưng bày.

Các món được đem ra theo thứ tự: 1 → 2 → 3 → 4 → 1 → 2 → 3 → 4.

- Đầu tiên, món 1 được đem ra và đặt vào một vị trí trống.
- Tiếp theo, món 2 được đem ra và đặt vào một vị trí trống.
- Sau đó, món 3 được đem ra và đặt vào một vị trí trống.

Lúc này các vị trí trưng bày đã được lấp đầy và buổi tiệc bắt đầu.

- Tiếp theo món 4 được đem ra và thay cho món 1, vì món 1 là món được trưng bày lâu nhất lúc này (tính 1 lần thay thế thức ăn)
- Tiếp theo món 1 được đem ra và thay cho món 2, vì món 2 là món được trưng bày lâu nhất lúc này (tính 2 lần thay thế thức ăn)
- Tương tự như vậy các món 2, 3, 4 được đem ra và cũng lần lượt được thay thế cho các món 3, 4, 1.

⇨ Như vậy tổng cộng chúng ta sẽ có 5 lần thay thế đĩa thức ăn

Hướng dẫn giải:

Đối với bài toán này chúng ta có thể suy luận ra một công thức giải quyết như sau:

Ta có tất cả n * k món được đem ra tất cả. Theo ví dụ trên thì ta sẽ có tổng cộng: 4 * 2 = 8 (8 lần thức ăn được đem ra). Và chỉ có m đĩa đầu tiên đem ra là không cần phải thay thế. Nên để tính số lần thay thế đĩa thức ăn ta có công thức như sau:

⇨ Công thức của mỗi truy vấn sẽ là (**n * k – m).** Thế số trong ví dụ 1 vào ta có (4 * 2 – 3) = 5.

Độ phức tạp: O(T) với T là số lần truy vấn.

Problem B: Wanderer

Solution:

C++	https://ideone.com/sht9Kv
Java	https://ideone.com/Uk01M9
Python	https://ideone.com/i81CKo

Trước khi giải quyết bài toán, ta nhận thấy: trường hợp duy nhất xảy ra để Aki có thể di chuyển được trong thời gian vô hạn là khi tồn tại một chu trình (đơn giản nhất là một đường vòng tròn) mà tất cả các đoạn đường nằm trên chu trình đó đều có chi phí bằng 0.

Để kiểm tra điều kiện này, ta có thể thực hiện tìm đường đi ngắn nhất giữa mọi cặp vị trí thông qua thuật toán Floyd-Warshall: điều kiện ở trên sẽ thoả mãn nếu có 1 vị trí bất kỳ mà tổng chi phí để đi đến chính nó bằng 0. Độ phức tạp cho quá trình này là $O(n^3)$, với n là số vị trí.

Tới đây, ta sẽ chỉ phân tích trường hợp bài toán có đáp số hữu hạn (không có chu trình nào mà tổng chi phí đúng bằng 0).

Ta nhận thấy: đường đi rẻ nhất giữa hai vị trí bất kỳ mà đi qua đúng k đường sẽ luôn ngắn hơn đường đi rẻ nhất giữa hai vị trí bất kỳ đi qua đúng k+1 đường. Dễ dàng chứng minh điều này nhờ phản chứng: nếu có một đường đi rẻ nhất giữa hai vị trí bất kỳ mà đi qua đúng k đường mà có chi phí a và một đường đi rẻ nhất giữa hai vị trí bất kỳ mà đi qua đúng k+1 đường mà có chi phí b, trong đó $a \geq b$, vậy nếu ta loại bỏ một đoạn đường có chi phí d ($d \geq 0$) nằm ở một trong hai đầu đường đi gồm k+1 đoạn thì ta được một đường đi gồm k đoạn có chi phí là b - d. Hiển nhiên $a \geq b - d$, mâu thuẫn với giả định đường đi rẻ nhất ở trên, từ đây giả thiết phản chứng sai.

Do đó, ta có thể tìm kết quả đúng thông qua tìm kiếm nhị phân. Ở đây, ta cần lưu ý, đáp án tối đa của bài toán có thể là nx + (n - 1), xảy ra khi cả n đỉnh tạo nên một chu trình trong đó có đúng 1 đường có chi phí 1, còn lại là các đường đi có chi phí 0 (Ta xuất phát từ một trong hai đầu đoạn đường chi phí 1 kể trên, đi đủ x vòng, sau đó đi vòng thứ x-1 đi trên các đoạn đường miễn phí cho tới khi đến đầu bên kia).

Với mỗi lần tìm kiếm nhị phân trên giá trị z, ta cần biết đường đi rẻ nhất đi qua đúng z đoạn có vượt quá chi phí x hay không - nếu có thì đáp án phải nhỏ hơn x, hoặc ngược lại.

Để kiểm tra điều này, ta dựa vào bản chất của thuật toán Floyd-Warshall và thực hiện thuật toán đó x lần trên đồ thị đã cho. Dĩ nhiên, vì x rất lớn nên quá trình này không thể thực hiện tuyến tính trong thời gian cho phép.

Ta có thể thực hiện quá trình trên thông qua tư duy chia để trị: việc biến đổi ma trận chi phí thông qua thuật toán Floyd-Warshall có đặc điểm khá tương đồng với nhân ma trận khi có tính kết hợp, do vậy thay vì tuần tự thực hiện lần lượt (tìm đường đi rẻ nhất độ dài k thông qua đường đi rẻ nhất độ dài k-1, rồi thông qua k-2, …), ta có thể chia nhỏ con số dưới hàm luỹ thừa (tìm đường đi rẻ nhất độ dài k thông qua đường đi rẻ nhất độ dài floor(k/2), rồi thông qua floor(k/4) [ở đây ta coi floor là phép lấy phần nguyên], …, liên tục cho tới trường hợp đơn vị là đường đi rẻ nhất độ dài 1).

Độ phức tạp của quá trình kiểm tra chu trình miễn phí sẽ là **$O(n^3)$.**

Độ phức tạp của quá trình tính toán các trường hợp hữu hạn **là $O(n^3 * \log^2 x)$** (n^3 do quá trình biến đổi, $\log^2 x$ do quá trình tìm kiếm nhị phân và quá trình chia để trị).

Problem C: Good Subsequence

Solution:

C++	https://ideone.com/8T5acO
Java	https://ideone.com/8uuvSJ
Python	https://ideone.com/4HP62h

Tóm tắt đề:

Với một dãy p bất kì, ta định nghĩa $sum(p)$ là tổng các phần tử trong dãy p. khi đó, dãy p được gọi là "tốt" nếu với mọi x từ 1 đến $sum(p)$, luôn tồn tại một dãy con (có thể không liên tiếp) của p sao cho tổng các phần tử của dãy con này bằng x.

Ví dụ:

- $[2, 3, 1]$, $[1, 1, 1, 4]$, $[1]$ là các dãy tốt
- $[4, 2, 3]$, $[5, 1, 2]$, $[100]$ không phải là các dãy tốt
    - Với dãy $[4, 2, 3]$ và $[100]$, không tồn tại dãy con có tổng bằng 1
    - Với dãy $[5, 1, 2]$, không tồn tại dãy con có tổng bằng 4

Cho dãy a gồm n phần tử. Hãy đếm xem, trong số $2^n - 1$ dãy con khác rỗng của a, có bao nhiêu dãy tốt. Vì kết quả có thể rất lớn nên hãy in kết quả sau khi chia lấy dư cho $10^9 + 7$.

Input:

Dòng đầu tiên gồm hai số nguyên n $(1 \le n \le 3000)$ – số phần tử của dãy a.

Dòng tiếp theo gồm n số nguyên $a_1, a_2, \dots, a_n$ $(1 \le a_i \le 3000)$ – mô tả dãy a.

Output:

In ra một số nguyên duy nhất là kết quả bài toán sau khi chia lấy dư cho $10^9 + 7$.

Ví dụ:

4 2 5 1 2	5
5 1 1 1 1 1	31

5 8 5 3 4 2	0

Giải thích ví dụ:

Ở ví dụ thứ nhất, có 5 dãy con của a là dãy tốt:

- $[1]$
- $[1, 2]$
- $[2, 1]$
- $[2, 1, 2]$
- $[2, 5, 1, 2]$

Ở ví dụ thứ hai, dãy a có $2^5 - 1 = 31$ dãy con, và các dãy con này đều là dãy tốt.

Ở ví dụ thứ ba, mọi dãy con của a đều không phải là dãy tốt nên kết quả là 0.

Hướng dẫn giải:

1) Điều kiện để một dãy p bất kì là một dãy tốt.

Đối với một dãy p, ta gọi độ phủ của dãy là số D lớn nhất sao cho với mọi k từ 1 đến D, luôn tồn tại một dãy con của p có tổng bằng k. Ví dụ:

- Dãy $[1, 2, 2, 7]$ có độ phủ là 5.
- Dãy $[5,1,2]$ có độ phủ là 3.
- Dãy $[2,3,4]$ có độ phủ là 0.

Ta kí hiệu $sum(p, i) = p_1 + p_2 + \cdots p_i$ (tổng i phần tử đầu tiên của dãy p). Dãy $p_1, p_2, \ldots, p_k$ là một dãy tốt tương đương với việc độ phủ của dãy p bằng $sum(p, k)$.

Làm thế nào để xác định nhanh độ phủ của một dãy $p_1, p_2, \ldots, p_k$ (giả sử $p_1 \leq p_2 \leq \cdots \leq p_k$)? Ta nhận xét rằng, khi ta thêm một phần tử t vào một dãy có độ phủ D thì:

- Nếu $t > D + 1$, ta không có cách nào để chọn tập con có tổng $D + 1$. Do đó, độ phủ mới của dãy sẽ vẫn là D.
- Nếu $t \leq D + 1$, với mọi k từ $D + 1$ đến $D + t$, ta có thể chuẩn bị một tập bao gồm số t và tập con (của dãy cũ) có tổng $k - t$. Do đó, độ phủ mới của dãy sẽ là $D + t$.

Như vậy, ta có thể xác định độ phủ của dãy p bằng cách xuất phát từ dãy rỗng có độ phủ 0, và lần lượt thêm các phần tử $p_1, p_2, \dots, p_k$. Giả sử như đã có dãy $p_1, p_2, \dots, p_{i-1}$ có độ phủ $sum(p, i-1)$, khi ta thêm phần tử p_i:

- Nếu $p_i > sum(p, i-1) + 1$, độ phủ của dãy $p_1, p_2, \dots, p_i$ vẫn sẽ là $sum(p, i-1)$. Việc thêm các phần tử $p_{i+1}, p_{i+2}, \dots$ cũng sẽ không làm tăng độ phủ của dãy nên ta kết luận độ phủ của dãy cả p cũng là $sum(p, i-1)$.
- Ngược lại, dãy $p_1, p_2, \dots, p_i$ sẽ có độ phủ $sum(p, i-1) + p_i = sum(p, i)$

Từ những nhận xét trên, ta có nhận định sau:

Dãy $p_1, p_2, \dots, p_k$ (giả sử $p_1 \le p_2 \le \dots \le p_k$) là dãy tốt khi và chỉ khi $p_1 = 1$ và $p_i \le sum(p, i-1) + 1$ với $2 \le i \le k$.

2) Bài toán "Đếm số dãy con của a là dãy tốt"

Ta có giải bài toán này bằng quy hoạch động:

Gọi $dp[i][j]$ là số dãy con tốt của dãy $a_1, a_2, \dots, a_i$ mà có tổng các phần tử bằng j.

Từ một dãy con bất kì của dãy $a_1, a_2, \dots, a_i$ mà có tổng các phần tử bằng j:

- Nếu $a_{i+1} \le j + 1$, ta có thể chọn hoặc không chọn phần tử a_{i+1} vào dãy con.
- Nếu $a_{i+1} > j + 1$, ta chỉ có thể không chọn phần tử a_{i+1} vào dãy con (nếu chọn thì dãy con sẽ không còn tốt).

Do đó, từ trạng thái $dp[i][j]$, ta cập nhật các trạng thái khác như sau:

- Tăng $dp[i+1][j]$ thêm $dp[i][j]$
- Nếu $a_{i+1} \le j + 1$ thì tăng $dp[i+1][j + a_{i+1}]$ thêm $dp[i][j]$

Lời giải này có độ phức tạp $O(n^2 * A)$ với A là giới hạn a_i tối đa.

Để cải tiến thuật toán trên, ta nhận xét rằng với mọi $j \ge A$ thì thao tác thứ hai trong quy tắc cập nhật trạng thái QHĐ trên luôn phải được thực hiện, và nó cũng chỉ tác động đến các trạng thái $(i+1, k)$ với $k \ge A$. Nói cách khác, ta có thể gộp các trạng thái (i, j) với $j \ge A$ thành một trạng thái duy nhất (i, A).

Với nhận xét trên, ta chỉ cần xét các trạng thái $dp[i][j]$ với $1 \le j \le A$. Từ trạng thái $dp[i][j]$, ta cập nhật các trạng thái khác như sau:

- Tăng $dp[i+1][j]$ thêm $dp[i][j]$
- Nếu $a_{i+1} \leq j+1$ thì tăng $dp[i+1][\mathbf{min}(\boldsymbol{A}, \boldsymbol{j}+\boldsymbol{a_{i+1}})]$ thêm $dp[i][j]$

Độ phức tạp: O(n * A) với n là số phần tử mảng a, A là giới hạn các phần tử của mảng a.

Problem D: XOR composition

Solution:

C++	https://ideone.com/ssQ7zT
Java	https://ideone.com/adKann
Python	https://ideone.com/lQIXaQ

Trước tiên, nếu ta có thể tính được tất cả các giá trị của $S(x)$ (x từ 0 đến $2^b - 1$), ta có thể tính $S(x)^M$ trong $O(logM)$ bằng thuật toán binary exponentiation. Ta sẽ tập trung vào cách tìm ra $S(x)$ với tất cả các giá trị của x.

Với mỗi số nguyên từ 0 đến $2^b - 1$, ta định nghĩa "giá trị" của x như sau: - Nếu tồn tại cặp a_i, w_i nào đó trong input sao cho $x = a_i$, thì $value_x = w_i$. - Nếu không tồn tại cặp nào sao cho $x = a_i$, $value_x = 0$.

Theo như trong test ví dụ, thì giá trị của 0 là 2, giá trị của 1 là 3, giá trị của 2 là 4, giá trị của 3 là 0 (vì không có a_i nào bằng 3).

Ta sẽ tiếp cận bài toán từ nhỏ đến lớn.

Gọi $dp(x,k)$ là tổng giá trị của các dãy số có độ dài k, và tổng XOR của dãy số này bằng x, ta có công thức truy hồi:

$$dp(x,k) = \begin{cases} \sum_{i \oplus j = x} d\,p(i,k-1) * value_j & (k > 1) \\ value_x & (k = 1) \end{cases}$$

Đáp án của bài toán là tổng của $S(x)^M = dp(x,k)^M$ với $x = 0$ đến $2^b - 1$. Ta có 2^b giá trị có thể của x, như vậy có $2^b * k$ trạng thái quy hoạch động. Mỗi trạng thái có thể được tính trong $O(2^b)$. Như vậy ta đã có thể giải quyết bài toán trong $O(2^b * 2^b * k)$ hay $O(4^b * k)$. Độ phức tạp này cho phép ta vượt qua 35% số test.

Để cải tiến thuật toán, ta sẽ xem xét lại cách ta di chuyển từ việc tính toán từ trạng thái độ dài k lên độ dài $k + 1$. Xét đoạn mã C++ sau:

```
for (int i = 0; i < (1 << b); i++) {
   for (int j = 0; j < (1 << b); j++) {
      next[i^j] += cur[i] * value[j]
```

```
    }
}
```

Đây thực chất là phép nhân đa thức bằng toán tử XOR, trong đó đa thức $next$ là tích của đa thức cur và $value$. Như vậy, việc chuyển từ trạng thái k lên trạng thái $k+1$ cũng là một phép nhân đa thức.

Ta sẽ đặt một đa thức P có bậc 2^b, trong đó hệ số của lũy thừa x bằng $value_x$ (như ta định nghĩa ban đầu). Khi đó ta có $S(x)$ đúng bằng hệ số của lũy thừa x trong đa thức $Q = P \cdot P \cdot P \cdot \ldots = P^K$ (đa thức P lũy thừa lên K lần). Bằng đoạn mã trên, ta có thể thực hiện phép nhân 2 đa thức bậc D trong $O(D^2)$. Để lũy thừa đa thức lên K lần, ta có thể dùng thuật toán binary exponentiation có độ phức tạp $O(logK)$. Ta có thể tìm tất cả các giá trị $S(x)$ bằng phép nhân đa thức trong $O((2^b)^2 * logK)$ hay $O(4^b * logK)$. Độ phức tạp này cho phép ta vượt qua 75% số test.

Để có thể vượt qua bài toán, ta sẽ sử dụng thuật toán biến đổi nhanh Walsh-Hadamard, cho phép nhân 2 đa thức bậc D bằng phép XOR trong $O(D * logD)$. Ta cải tiến độ phức tạp của thuật toán lên $O(b * 2^b * logK)$.

Độ phức tạp của cả bài toán: $\boldsymbol{O(b * 2^b * logK * logM)}$