Jon and Orbs

Bài này yêu cầu ta cần tìm ngày đầu tiên mà xác suất sinh ra được đủ k loại là ít nhất $\frac{p_i-\epsilon}{2000}$.

Chú ý điều kiện $p_i \leq 1000$, từ điều kiện này thấy được ta chỉ cần quan tâm đến một số lượng ngày đầu tiên cho đến khi xác suất có đủ k loại vượt quá 0.5.

Xác suất 0.5 thực ra không quá khó để đạt được. Trên thực tế, khi $k = 1000$, ta chỉ cần 7274 ngày là đạt được xác suất này. Vì vậy ta có thể thực hiện tính ra với tất cả các ngày trên thì xác suất để có đủ k loại là bao nhiêu rồi sau đó có thể tìm kiếm nhị phân (thậm chí là tìm kiếm tuần tự) để tính ra ngày đầu tiên thỏa mãn cho mỗi giá trị p_i.

Để tính xem xác suất có k loại ở mỗi ngày là bao nhiêu, ta có thể tính thẳng luôn là ở ngày thứ i, xác suất có j loại khác nhau là bao nhiêu bằng quy hoạnh động.

Cụ thể, gọi $f[i][j]$ là xác suất mà sau ngày thứ i có j loại khác nhau.

Vậy thì ta có công thức quy nạp:

$$f[i][j] = f[i-1][j] \times \frac{j}{k} + f[i-1][j-1] \times \frac{k-j+1}{k}$$

Trong đó:

- $f[i-1][j] \times \frac{j}{k}$ là xác suất đến ngày $i-1$, ta có j loại khác nhau ($f[i-1][j]$), đến ngày i, một trong j loại đó được sinh ra ngẫu nhiên $\left(\frac{j}{k}\right)$.
- $f[i-1][j-1] \times \frac{k-j+1}{k}$ là xác suất đến ngày $i-1$, có $j-1$ loại khác nhau ($f[i-1][j-1]$), đến ngày i, một trong số $k-(j-1)$ loại chưa được sinh ra trước đó được sinh ra, tăng tổng số lượng loại khác nhau đã có lên j $\left(\frac{k-j+1}{k}\right)$.

Độ phức tạp tính toán là khoảng $O(7000 \times k)$.

Sau khi tính được là đến ngày i thì xác suất có k loại khác nhau là bao nhiêu, khi truy vấn có thể tìm kiếm nhị phân hoặc đơn giản là duyệt lại, độ phức tạp cũng sẽ không bị kém hơn.

Chú ý khi cài đặt, nên sử dụng một mảng lưu mình chiều $[j]$ thay vì lưu lại cả hai chiều $[i][j]$. Điều này không chỉ để giảm bộ nhớ mà thực tế còn giảm thời gian chạy.

- Cụ thể, ta duyệt ngược lại từ $i = k$ và cập nhật:

  ```
  f[i]=(f[i-1]*(k-i+1)+f[i]*i)/k;
  ```

- Vì ta chỉ thực hiện thao tác vào hai phần tử ở cạnh nhau trên mảng, đầu đọc/ghi sẽ không phải di chuyển nhiều như khi thao tác từ mảng $f[i]$ sang mảng $f[i-1]$ (di chuyển bằng độ dài mảng).
- Trong trường hợp máy có hỗ trợ cache, điều trên thực tế không giúp gì nhiều trong bài này vì tính toán không quá lớn (7×10^6). Trong các bài cần tính toán và truy cập mảng nhiều hơn (lên đến 10^8), tối ưu cache thực sự sẽ giúp cải thiện đáng kể thời gian chạy.

https://codeforces.com/contest/768/submission/61675714

Nhìn chung trong bài này, việc nhìn ra tốn không quá nhiều ngày để có xác suất ≥ 0.5 là mấu chốt để giải được. Có một câu hay dùng để nói về các bài như thế này:

> “Số lượng trạng thái bé.”
>
> *-Nhiều người từng nói, không biết ai nói đầu tiên.*

Với những bài này thì đôi khi có cách để lí giải/chứng minh ra được là số lượng trạng thái bé và có thể làm được bằng quy hoạch động / duyệt nhưng chủ yếu là phải code rồi thử chạy xem có được không (hoặc ít nhất là code để tính số trạng thái). Nếu đủ nhanh được thì tốt, nếu không được thì phải tìm cách khác. Trong khi thi VOI, ta không (chưa) được chấm ngay mà phải nộp offline rồi sau này mới chấm nên việc kiểm tra kĩ lại là rất cần thiết. Nếu như cảm giác thấy một bài là bài có số trạng thái bé mà dễ code được, các bạn cứ mạnh dạng code thử. Nếu được thì quá tốt, nếu không được thì các bạn cũng ăn được thêm một ít, thêm nữa nếu sau này có nghĩ ra thuật khác thì cũng có code để mà so sánh kết quả với thuật khác.

Chứng minh rằng bài này không có quá nhiều trạng thái thì có thể làm như sau:

- Gọi P_n là xác suất để đến ngày thứ n thì có đủ k loại.
- Ta có thể tính P_n bằng bao hàm loại trừ thay vì tính bằng quy hoạch động như trong giải thuật.
- Vậy thì ta tính số cách mà có tối đa k loại khác nhau trừ đi số cách mà có tối đa $k-1$ loại khác nhau.
- Ta có được $P_n \geq \frac{k^n-(k-1)^n\times k}{k^n}$ vì:
    - k^n là số cách mà dùng tối đa k loại.
    - $(k-1)^n \times k$ lớn hơn hoặc bằng số cách mà dùng tối đa $k-1$ loại vì ý tưởng là ta sẽ cấm không cho dùng một loại nào đó trong k loại tuy nhiên sẽ các trường hợp dùng ít hơn hẳn $k-1$ loại bị tính nhiều lần.
    - Vì vậy $k^n-(k-1)^n \times k$ bé hơn hoặc bằng số lượng dùng đúng k loại nên $P_n \geq \frac{k^n-(k-1)^n\times k}{k^n} = 1-(k-1)\left(\frac{k-1}{k}\right)^n$
- $1-(k-1)\left(\frac{k-1}{k}\right)^n \geq 0.5 \Leftrightarrow (k-1)\left(\frac{k-1}{k}\right)^n \leq 0.5$
- Vì $\frac{k-1}{k} > \frac{k'-1}{k'}$ khi $k > k' > 0$ (cơ bản) và $k-1 > k'-1$ khi $k > k' > 0$, k càng lớn thì giá trị $(k-1)\left(\frac{k-1}{k}\right)^n$ càng lớn.
- Vậy thì ta chỉ cần tìm n trong trường hợp k lớn nhất $(k = 1000)$
- Giải ra được $n = 7597$, không quá tồi so với giá trị chính xác 7274.
- Vậy thì ta cần không quá 7597 ngày để xác suất cho tất cả các trường hợp của k lớn hơn 0.5

Chú ý rằng khi chứng minh thuật toán, ta không cần quá cầu kì về giá trị cụ thể mà có thể dùng bất đẳng thức và ước lượng để làm nhanh hơn. Cũng chú ý luôn rằng trong các phương pháp tính toán như bao hàm loại trừ hay là khai triển Euler, không cần có quá nhiều phần tử để đạt được độ chính xác đủ dùng, trong khi chứng minh sẽ rất tiện lợi.

Sereja and Two Sequences

Bài này cho ta hai loại thao tác, một loại có giá cố định là e, loại thứ hai có giá bằng số lượng số ta bỏ đi ở các thao tác trước đó.

Loại thứ nhất có thể dùng nhiều lần nhưng loại thứ hai chỉ được dùng một lần, vì sau khi dùng xong thì trò chơi kết thúc vì không còn số nào để chơi nữa.

Nhận xét luôn là phải dùng loại thứ hai thì mới có tiền, còn không sẽ không được xu nào, nên luôn phải dùng loại thứ hai.

Ý tưởng đơn giản nhất cho bài này là ta sẽ quy hoạch động $f[i][j][k][l]$ là có cách nào thỏa mãn: trên dãy thứ nhất thì ta đã xóa hết i số đầu, trên dãy thứ hai đã xóa hết j số đầu, ta đã có được k xu (chưa nhận) và còn lại l năng lượng từ s năng lượng ban đầu.

Kết quả là $\max(k)$ trong mọi trạng thái $f[i][j][k][l]$ thỏa mãn $l \geq i + j$ vì ta cần phải thực hiện thao tác thứ hai để lấy kết quả.

Thực tế, vì giá trị của mỗi thao tác thứ nhất bằng nhau và bằng e, ta không cần phải lưu l vì dĩ nhiên $l = s - ke$.

Vậy thì ta rút gọn được về hàm $f[i][j][k]$.

Nhận thấy là cái hàm $f[i][j][k]$ là hàm theo hiểu "có trường hợp này không" mà bài này là một bài toán tối ưu, vậy nên ta có thể có khá năng để chuyển nó thành một hàm theo kiểu "trong các trạng thát thỏa mãn trường hợp này, trạng thái nào là tốt nhất."

Hãy xem xét các cách chuyển:

- Nếu ta đổi thành $f[i][j]$ là k tối đa là bao nhiêu thì cũng làm được. Dĩ nhiên rất nhiều khi ta sẽ không thực hiện tối đa mà vẫn đủ năng lượng nên sau này ta sẽ phải đi tìm giá trị tối đa $x \leq f[i][j]$ sao cho $x \times e + i + j \leq s$. Hiểu đơn giản là ta bỏ qua không thực hiện phép thứ nhất một số lần để có đủ năng lượng.
    - Nhìn chung cách này cũng khá là hay, tuy nhiên ta phải lưu và tính $n \times m = 10^{10}$ phần từ như vậy (trong trường hợp xấu nhất), không đủ thời gian/bộ nhớ.

    - Nếu $n \times m$ bé hơn thì cách này không phụ thuộc vào giá trị của s và e.
- Nhận thấy là sau này ta sẽ phải trả chi phí là $i + j$ cho thao tác thứ hai, nên nếu cố định i, k thì j càng bé thì càng tốt. Vậy thì ta có thể thực hiện quy hoạch động là $f[i][k]$ là giá trị j bé nhất có thể sau cho thực hiện thao tác thứ nhất được k lần.
    - Nhận thấy kết quả không quá $\frac{s}{e}$ nên ta chỉ cần lưu k lên đến $\frac{s}{e}$ là đủ.
    - Điều kiện cho biết $s \leq 3 \times 10^5, 10^3 \leq e \leq 10^4$ nên thực tế $\frac{s}{e} \leq 300$.
    - Ta có thể tính $f[i][k] = next_b\big(a[i], \min(f[i-1][k-1])\big)$. $next_b(x, i)$ là tìm vị trí đầu tiên của x sau vị trí thứ i của x trên b. Đơn giản là nếu ta muốn tối thiểu hóa $f[i][k]$ thì ta nên tìm vị trí đầu tiên mà a_i xuất hiện trong b sau vị trí bé nhất mà ta có thể tiếp tục từ một trong các đoạn trước đó đã có được $k-1$ (đề bài yêu cầu khi xóa thì hai phần tử cuối cùng ở hai dãy phải bằng nhau).
    - Nhận thấy là ta có thể tính được các giá trị trên với độ phức tạp là $O\big(n \times \max(k) \times O(next_b)\big)$ vì $\min(f[i-1][k-1])$ có thể tính trước được bằng một mảng.
    - $O(next_b)$ là độ phức tạp để tính hàm $next_b$. Độ phức tạp này thực tế khó làm tốt hơn được $O(\log(m))$ nhưng như vậy là đủ trong bài này. Để làm được $O(\log(m))$, tạo một vector vị trí cho từng giá trị và thực hiện tìm kiếm.
    - Vậy độ phức tạp là $O(n \times \max(k) \times \log(m)) = n \times 300 \times \log(m)$ tuy nhiên với một số cải tiến sẽ chạy đủ nhanh.
    - Cải tiến có thể là kiểm tra xem có chắc chắn có giá trị đó không rồi mới tìm kiếm nhị phân thì sẽ giảm được thời gian chạy tổng cộng. Có thể chỉ duyệt k đến $\frac{s-i}{e}$, …
    - Nhận thấy thêm là $f[i][k]$ thực tế không quan trọng, nên ta có duyệt i tăng dần và chỉ lưu là $f'[k]$ là $\min(f[i'][k]), i' \leq i$. Lưu như thế này thì ta sẽ mất thông tin về i để tính kết quả, nên phải vừa duyệt i vừa tính kết quả. Cách này giảm bộ nhớ và tối ưu cache, …
    - https://codeforces.com/contest/425/submission/61897664

Karen and Supermarket

Bài này thực tế là cho một cái cây gốc ở 1, mỗi nút i có hai loại giá, một là c_i hai là $c_i - d_i$. Nếu ta dùng loại lá thứ hai cho một nút thì ta buộc phải dùng loại thứ hai cho cha của nó, và cứ thế nên thực ra ta sẽ phải mua dùng loại thứ hai cho tất cả các nước tổ tiên của nút này.

Vậy thì ta có thể quy hoạch động $f[i][x][j]$ là số tiền bé nhất đã dùng nếu ta mua được j nút trong nút con của i, $x = 0$ thì ta không bị bắt mua các nút tổ tiên của i, nếu $x = 1$ thì ta bắt buộc phải mua các nút tổ tiên của i.

Kết quả là giá trị j tối đa sao cho $f[1][0][j] \leq b$ hoặc $f[1][1][j] \leq b$.

Kiểu quy hoạch động trên cây này thì ta sẽ duyệt từng nút con, tính nút con rồi dùng để cập nhật ngay vào nút cha.

Cụ thể khi duyệt nút u:

- Đầu tiên ta gán các giá trị $f[u][0][0] = 0, f[u][0][1] = c[u]$ vì ta có thể không mua nút con nào trong u hoặc mua mình nút u.
- Ta cũng gán các giá trị $f[u][1][0] = b + 1$ và $f[u][1][1] = c[u] - d[u]$ Nếu ta bắt buộc phải mua u và các tổ tiên của nó thì ta cũng phải mua u, vì thế thì ta phải mua ít nhất một nút nào đó nên trạng thái $f[u][1][0]$ thực tế không đặt được. Vì ta đang tối thiểu hóa số tiền, ta cần gán $f[u][1][0] = \infty$. Ta gán $f[u][1][0] = b + 1$ là đủ vì ta chỉ cần quan tâm đến các giá trị không quá b. $f[u][1][1] = c[u] - d[u]$ đơn giản là trường hợp mà ta sử dụng coupon ở nút u.
- Tất cả các giá $f[u][0][i]$ và $f[u][1][i]$ $(i > 1)$ gán hết bằng $b + 1$ vì ta chưa có cách để đạt được các trạng thái này.
- Sau đó ta sẽ đi duyệt một nút con v của u.
    - Đệ quy vào tính v.
    - Tính được v thì ta có thể thực hiện tối ưu hóa là các giá trị $f[u][0][i]$ và $f[u][1][i]$ như sau:
        - $f[u][0][i + j] = \min(f[u][0][i + j], f[u][0][i] + f[v][0][j])$ (Nếu ta không mua u giảm giá thì dĩ nhiên không được mua v giảm giá.)
        - $f[u][1][i + j] = \min(f[u][1][i + j], f[u][1][i] + f[v][0][j])$
        - $f[u][1][i + j] = \min(f[u][1][i + j], f[u][1][i] + f[v][1][j])$\

(Nếu ta đã mua u mà giảm giá thì ta muốn mua v kiểu nào cũng được.)

    - Để không bị dùng $f[v][x][j]$ tính cho $f[u][x][i+j]$ nhiều lần thì ta có thể duyệt i giảm dần rồi mới duyệt j.
- Nếu mỗi lần mà ta duyệt lại i và j lên đến n thì độ phức tạp là $O(n^3)$ không đủ nhanh.
- Nếu ta chỉ duyệt các giá trị i và j cần thiết thì độ phức tạp thực tế $O(n^2)$.
- Cụ thể:
    - Giả sử u có các nút con lần lượt là $v_1, v_2, v_3, \dots, v_k$.
    - Ta gọi $size(a)$ là độ lớn cây con nút a.
    - Khi duyệt các giá trị j của mỗi v_i, rõ ràng ta không mua quá $size(v_i)$ nút được nên chỉ duyệt j đến $size(v_i)$ là đủ.
    - Nếu ta đã duyệt xong các nút $v_1, v_2, \dots, v_{t-1}$ thì ta chỉ cần duyệt i đến $1 + size(v_1) + size(v_2) + \cdots + size(v_{t-1})$ là đủ vì đó là số lượng tối đa mà ta đã mua được.
    - Vậy thì độ phức tạp ở nút con của u là:

$$sum\left(\left(sum\left(size(v_y)\right)+1\right)\times size(v_x)\right)$$

(Với $1 \le y < x \le k$).)

    - Có thể biến đổi giá trị trên bằng nhiều cách để đạt được giới hạn $O(n^2)$, tuy nhiên có một cách lí giải hay như sau:
        - Tưởng tượng ban đầu có một cây có một đỉnh là u.
        - Mỗi lần cập nhật v_i, thực tế là ta nối hai cây làm một, độ phức tạp là tích số nút của hai cây.
        - Sau khi cập nhật hết thì cả cây con gốc u là một cây và không có chuyện ta nối hai hậu duệ của u nữa.
        - Vì mỗi nút chỉ được nối với mỗi nút khác một lần duy nhất trong cả quá trình tính toán, tổng độ phức tạp là $O(n^2)$

https://codeforces.com/contest/815/submission/61607780

Nhìn chung bài này không có gì đặc biệt, nếu các bạn đã biết kĩ thuật quy hoạch động trên cây $O(n^2)$ này thì nó là bài đơn giản.

Kĩ thuật này được sử dụng đến trong nhiều bài quy hoạch động trên cây, các bạn nên nắm vững.

Ngoài ra kĩ thuật này cũng cho thấy rằng, đôi khi, cách làm mà tối thiểu hóa các hành động không cần thiết sẽ tốt hơn các bạn tưởng tượng. Nếu các bạn thấy là cách đó nhìn qua không được nhưng có cách để tối ưu hóa được thì có khi cũng nên mạo hiểm cài đặt thử, hoặc là nháp ra xem nếu tối ưu như vậy thì độ phức tạp có ổn không.