Cách giải Bài 1. ĐẾM ĐOẠN

Thuật toán quá đơn giản nên càng trình bày càng khó hiểu, mời các bạn xem code (ở đây các ký tự trong xâu S được đánh số từ 0)

```
int n = s.length();
int R = s.find_last_of(s[0]);
int L = s.find_first_of(s[n - 1]);
Chọn cách tốt hơn trong 2 cách sau
1/ s[0 … R – 1] và s[1 … R]
2/ s[L … n – 2] và s[L + 1 … n - 1]
```

Cách giải Bài 2. CHIA ĐỘI

Đây có lẽ là bài toán khó nhất trong đề nếu đòi hỏi thuật toán $O(n)$, thuật toán $O(n \log n)$ dễ hơn nhiều và cũng đủ để có trọn vẹn số điểm, ta sẽ trình bày thuật toán $O(n)$, bạn có thể tự thiết kế thuật toán $O(n \log n)$ để so sánh.

Bằng một phép sắp xếp, ta có thể coi dãy $a_{1\dots n}$ được sắp giảm dần:

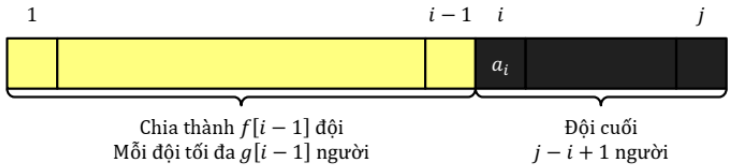
$$a_1 \geq a_2 \geq \cdots \geq a_n$$

Chú ý rằng các giá trị trong mảng $a[1 \dots n]$ đều là số nguyên $\in \{1 \dots n\}$ nên để sắp xếp ta có thể sử dụng thuật toán đếm phân phối mất thời gian $O(n)$.

Nhận xét 1: Trong các phương án chia đội tối ưu, tồn tại phương án mà mỗi nhóm gồm những thí sinh liên tiếp. Thật vậy, với một cách chia đội bất kỳ, xét đội chứa thí sinh 1, nếu đội này có thí sinh $i+1$ mà không có thí sinh i, ta có thể yêu cầu hai thí sinh i và $i+1$ đổi đội, khi ấy thí sinh i được đưa vào đội có ít nhất a_1 người còn thí sinh $i+1$ được đưa vào đội có ít nhất a_i người. Do $a_1 \geq a_i$ và $a_i \geq a_{i+1}$ nên yêu cầu của mọi thí sinh vẫn được thỏa mãn. Cứ làm như vậy cho tới khi đội có thí sinh 1 gồm các thí sinh liên tiếp (chẳng hạn từ 1 tới k), ta tiếp tục phương pháp tương tự với đội có thí sinh $k+1$... Như vậy ta chỉ cần quan tâm tới phương án chia đội mà mỗi đội gồm những thí sinh liên tiếp.

Xét dãy thí sinh từ 1 tới j, gọi $f[j]$ là số đội cực đại có thể thành lập từ những thí sinh này và $g[j]$ là giới hạn cực tiểu sao cho có thể thành lập $f[j]$ đội mà số thí sinh trong mỗi đội không vượt quá $g[j]$ (Nếu không có cách chia đội cho các thí sinh này coi như $f[j] = -\infty$ và $g[j]$ vô nghĩa). Ta cần xây dựng công thức truy hồi tính các $f[j]$ và $g[j]$ với $\forall j: a_1 < j \leq n$.

Nếu ta biết được trong phương án tối ưu, đội cuối cùng chứa các thí sinh $i \dots j$ thì vấn đề trở thành chia tối ưu trên dãy các thí sinh từ 1 tới $i-1$.



Tức là:

$$\begin{cases} f[j] = f[i-1] + 1 \\ g[j] = \max(g[i-1], j-i+1) \end{cases}$$

Vậy để tính $f[j]$, ta xét mọi chỉ số i thỏa mãn $1 \leq i \leq j$ và $a[i] \leq j-i+1$ (gọi chỉ số i này là điểm cắt). Với mỗi điểm cắt i thì $f[j]$ được cực đại hóa theo $f[i-1]+1$. Để tính $g[j]$, ta cũng xét mọi điểm cắt $i \leq j$ như trên, bổ sung thêm điều kiện $f[j] = f[i-1]+1$ rồi cực tiểu hóa $g[j]$ theo $\max(g[i-1], j-i+1)$.

Cơ sở: $f[0] = 0$; $g[0] = 0$; $f[a_1] = 1$; $g[a_1] = a_1$ còn $\forall j: 0 < j < a_1$ thì $f[j] = -\infty$. Công thức truy hồi được dùng để tính lần lượt các $f[j]$ và $g[j]$ với $j \in \{a_1 + 1 \dots n\}$. Thuật toán $O(n^2)$ không có gì khó khăn.

Thuật toán $O(n)$ đòi hỏi có thêm những nhận xét tinh tế. Ta sẽ tách riêng hai công đoạn tính hàm mục tiêu f và hàm mục tiêu g.

Tính f:

Nhận xét 2: với $\forall j > a_1$ thì:

$$f[j-1] \leq f[j] \leq f[j-1] + 1$$

Tức là nếu bổ sung thêm thí sinh j vào tập thí sinh $\{1,2, \dots, j-1\}$, số đội trong phương án tối ưu không giảm đi nhưng cùng lắm chỉ tăng lên 1.

Thật vậy, với một phương án chia đội cho những thí sinh từ 1 tới $j-1$, ta chỉ cần bổ sung thí sinh j vào đội cuối là được một phương án chia đội cho những thí sinh từ 1 tới j, suy ra $f[j] \geq f[j-1]$.

Ngược lại, với một phương án chia ≥ 2 đội cho những thí sinh từ 1 tới j, ta chỉ cần nhập 2 đội cuối lại và loại bỏ thí sinh j là được một phương án chia đội cho những thí sinh từ 1 tới $j-1$, suy ra $f[j-1] \geq f[j]-1$.

Bây giờ làm thế nào để kiểm tra $f[j]$ có phải bằng $f[j-1]+1$ hay không?

Xét phương án tối ưu chia đội cho những thí sinh 1 ... j, giả sử đội cuối gồm những thí sinh i ... j, khi đó có điều kiện $a_i \leq j-i+1$. Ta sẽ chỉ ra rằng nếu $f[j] = f[j-1]+1$ thì bắt buộc $a_i = j-i+1$, bởi nếu $a_i < j-i+1$ ta chỉ cần loại bỏ thí sinh j khỏi đội cuối là được phương án chia đội cho $j-1$ thí sinh đầu tiên với số đội bằng $f[j]$ (lớn hơn $f[j-1]$). Mâu thuẫn.

Điều kiện $a_i = j-i+1$ có thể viết thành $a_i + i - 1 = j$. Thuật toán $O(n)$ tính hàm mục tiêu f như sau: Với mỗi chỉ số i, ta đưa nó vào danh sách $L[a_i + i - 1]$. Để tính $f[j]$, trước tiên ta đặt $f[j] = f[j-1]$, sau đó thay vì phải xét mọi điểm cắt i đứng trước chỉ số j, ta chỉ cần xét mọi điểm cắt i trong danh sách $L[j]$ mà thôi. Tổng kích thước các danh sách $L[.]$ là n phần tử và mỗi danh sách chỉ phải duyệt qua 1 lần khi tính hàm mục tiêu f nên độ phức tạp tính toán là $O(n)$.

Tính g:

Dựa vào những lập luận trong thuật toán tính $f[.]$, nếu $f[j] = f[j-1]+1$ thì để tính $f[j]$ ta chỉ cần xét các điểm cắt $i \in L[j]$ có $f[i-1] = f[j]-1$ rồi cực tiểu hóa $g[j]$ theo $\max\left(g[i-1], \underbrace{j-i+1}_{=a_i}\right)$

Trường hợp $f[j] = f[j-1]$, xét phương án tối ưu chia các thí sinh 1 ... j thành $f[j]$ đội (tối ưu theo nghĩa cho $g[j]$ nhỏ nhất). Phương án này có thể thuộc một trong hai loại:

Loại 1: Đội cuối là "chặt", có nghĩa là nếu đội cuối gồm các thí sinh i ... j và ta bỏ thí sinh j ra khỏi đội cuối thì sẽ vi phạm yêu cầu của thí sinh i. Điều này có nghĩa là $a_i = j-i+1$ hay $i \in L[j]$. Như vậy nếu phương án tối ưu rơi vào loại 1, nó có thể được xác định bằng cách xét mọi điểm cắt $i \in L[j]$, chọn ra điểm cắt i thỏa mãn hai điều kiện:

$$\begin{cases} f[i-1] = f[j]-1 \\ \max(g[i-1], a_i) \text{ nhỏ nhất có thể} \end{cases}$$

Loại 2: Đội cuối là không "chặt" tức là nếu đội cuối gồm các thí sinh i ... j và ta bỏ thí sinh j ra khỏi đội cuối thì được một phương án chia đội cho các thí sinh 1 ... $j-1$. Điều này cho thấy $g[j] \geq g[j-1]$.

- Nếu $f[j-1] \times g[j-1] = j-1$, tức là $j-1$ thí sinh đầu tiên được chia tối ưu thành $f[j-1]$ đội với số thành viên bằng nhau và bằng $g[j-1]$. Khi thêm thí sinh j vào, $g[j]$ chắc chắn $> g[j-1]$, vậy ta thêm thí sinh j vào đội cuối và thu được $g[j] = g[j-1]+1$. Vì $g[j] > g[j-1]$ nên $g[j] = g[j-1]+1$ đảm bảo là giá trị $g[j]$ tối ưu.
- Nếu không, tức là $j-1$ người đầu tiên được chia tối ưu thành $f[j-1]$ đội với số thành viên không hoàn toàn giống nhau, trong trường hợp này ta có thể thêm người j vào đội có ít thành viên nhất và làm cho $g[j] = g[j-1]$. Vì $g[j] \geq g[j-1]$ nên $g[j] = g[j-1]$ đảm bảo là giá trị $g[j]$ tối ưu. (Việc phân đội cho thí sinh j thế này có thể khiến cho các đội không gồm các thí sinh liên tiếp nhưng ta đã lập luận rằng sẽ có phương án chia đội gồm các thí sinh liên tiếp tương đương)

Việc tìm $g[j]$ tốt nhất trong các phương án loại 1 mất thời gian $O(|L[j]|)$ còn tìm $g[j]$ tốt nhất trong các phương án loại 2 mất thời gian $O(1)$. Độ phức tạp tính toán của thuật toán tính $g[.]$ là $O(n)$.

```
«Tạo các danh sách L[1…n]»;
//Tính f
f[0] = 0; f[1…a[1] – 1] = -∞; f[a[1]] = 1;
for (j = a[1] + 1; j <= n; ++j)
{
    f[j] = f[j - 1];
    for (i: L[j])
        «Cực đại hóa f[j] theo f[i – 1] + 1»;
}
//Tính g
for (j = a[1] + 1; j <= n; ++j)
{
    g[j] = -∞;
    if (f[j] > f[j - 1])
    {
        for (∀i ∈ L[j]: f[j] == f[i - 1] + 1)
           «Cực tiểu hóa g[j] theo max(g[i - 1], j - i + 1)»;
    }
    else //f[j] == f[j - 1]
    {
        //Cực tiểu hóa g[j] theo các phương án loại 1
        if (g[j - 1] * f[j - 1] == j - 1)
            g[j] = g[j - 1] + 1;
        else
            g[j] = g[j - 1];
        //Cực tiểu hóa g[j] theo các phương án loại 2
        for (∀i ∈ L[j]: f[j] == f[i - 1] + 1)
            «Cực tiểu hóa g[j] theo max(g[i - 1], j - i + 1))»;
    }
}
```

Cách giải Bài 3. MIỀN LỚN NHẤT

Đọc đồ thị kết hợp với cấu trúc dữ liệu Disjoint-set Forest và xây dựng danh sách kề: Khởi tạo n tập hợp rời nhau, mỗi tập gồm một đỉnh. Khi đọc được cạnh (u, v):

- Nếu u cùng màu với v: Hợp nhất tập chứa u và tập chứa v nếu hai tập này rời nhau
- Nếu u khác màu với v: Đưa u vào danh sách kề của v và v vào danh sách kề của u

Đọc xong dữ liệu ta có ngay:

- Mỗi tập hợp là một miền, xác định được luôn số đỉnh của miền lớn nhất
- Các danh sách kề $adj[u]$ chứa các đỉnh kề u và khác màu với u

Khi ta muốn đổi màu một đỉnh để thu được miền lớn nhất thì miền lớn nhất chắc chắn phải chứa đỉnh đó.

Bây giờ ta thử xét tình huống đổi màu đỉnh u thành màu c thì miền lớn nhất chứa u được tạo thành như thế nào. Dễ thấy rằng miền chứa u khi đó sẽ gồm đỉnh u và tất cả các miền màu c kề với u.

Khởi tạo mảng đếm $Count[1 \dots n] = 0$, với mỗi màu c, ta sẽ tính $Count[c]$ là tổng số đỉnh trong các miền màu c mà kề với u. Vì danh sách kề của u có thể có nhiều đỉnh chung một miền, trong khi để tính $Count[c]$ thì mỗi miền màu c kề u chỉ được tính một lần, ta cần có một mảng đánh dấu để tránh tính lặp: Khởi tạo mảng $Mark[1 \dots n] =$ false, *ở đây* $Mark[x] =$ true cho biết miền x đã được xét còn $Mark[x] =$ false tức là miền x chưa được xét. Cách tính khá đơn giản:

```
for (v: adj[u])
{
    x = FindSet(v); //x = miền chứa v
    if (Mark[x]) continue; //Miền x đã xét, bỏ qua
    Mark[x] = true; //Đánh dấu miền x đã xét
    Count[Color[v]] += Kích_Thước_Miền(x); //Đếm số đỉnh trong các miền màu c kề u
}
```

Tiếp theo ta cập nhật kết quả theo $\max\{Count[1 \dots n]\} + 1$, đồng thời reset lại mảng $Count$ thành 0 và mảng $Mark$ thành false. Lạm dụng các lệnh fill hay sử dụng vòng lặp vụng về sẽ làm độ phức tạp trở thành $\Theta(n)$. Cách đúng đắn là duyệt lại danh sách kề của u để reset trước khi chuyển sang thử đổi màu một đỉnh u khác.

```
for (v: adj[u])
{
    c = Color[v];
    «Cập nhật kết quả theo Count[c] + 1»;
    Count[c] = 0; //Reset mảng Count
    Mark[FindSet(v)] = false; //Reset mảng Mark
}
```

Độ phức tạp khi xử lý đỉnh u: $O(|adj[u]|)$

Độ phức tạp khi xử lý tất cả các đỉnh: $O(m)$

Chi phí đọc dữ liệu và Disjoint-Set Forest: $O(m.\alpha(n))$

Tổng thể: $O(m.\alpha(n))$ mỗi test.

Việc sử dụng Map/Set thay vì đếm/đánh dấu sẽ khiến chương trình tốn bộ nhớ hơn và chạy chậm hơn rất nhiều, cần máy cấu hình mạnh mới qua được.

Một kỹ thuật cải tiến nữa có thể áp dụng:

Thay vì đánh dấu bằng mảng bool, ta khai báo mảng $Mark$ kiểu int và khởi tạo bằng giá trị 0, khởi tạo một biến int $CurrentMark = 0$. Ý nghĩa là nếu $Mark[x] == CurrentMark$ thì miền x đã đánh dấu.

Khi bắt tay vào xử lý đổi màu một đỉnh u, ta chỉ cần $++CurrentMark$ là đủ và khi muốn đánh dấu một miền x đã xét, ta chỉ cần gán $Mark[x] = CurrentMark$. Điều này giúp cho việc reset lại mảng $Mark$ mất thời gian $O(1)$ vì không cần phải duyệt lại $adj[u]$

Vấn đề xử lý tương tự khi muốn reset mảng $Count$. Việc này để các bạn tự làm mới có hy vọng nhanh hơn JuryCPP

Ngoài ra có thể thay đoạn xử lý Disjoint-Set Forest bởi BFS/DFS có độ phức tạp nhỏ hơn (tuy nhiên code dài dòng hơn một chút).