# A. Beautiful Binary String

## Tóm tắt đề bài

1 xâu nhị phân độ dài n được gọi là đẹp nếu như nó thỏa mãn 2 điều kiện sau:

- Xâu chỉ bao gồm kí tự ‘0’ hoặc ‘1’
- Nếu cắt đôi xâu đó ở 1 vị trí bất kì, xâu ở nửa trái sẽ luôn có thứ tự từ điển nhỏ hơn xâu ở nửa phải.

Cho 1 xâu đẹp S độ dài n, ta phải tìm 1 xâu đẹp độ dài n có thứ tự từ điển nhỏ nhất nhưng lớn hơn S.

## Lời giải

Ta coi vị trí x của 1 xâu là đẹp nếu như ta cắt đôi xâu đó ở vị trí x, xâu ở nửa trái sẽ có thứ tự từ điển nhỏ hơn xâu ở nửa phải. Và 1 xâu gọi là đẹp nếu như mọi vị trí của xâu đó đều là vị trí đẹp.

Trước tiên ta xét 1 bài toán phụ: Cho 1 xâu nhị phân S, ta cần tìm xâu có thứ tự từ điển nhỏ nhất nhưng vẫn lớn hơn S (trong bài này ta tạm gọi là xâu kế tiếp). Bài toán có thể dễ dàng giải bằng cách tìm vị trí x cuối cùng mà $S_x = 0$, đặt $S_x = 1$ và điền mọi bit sau vị trí x bằng 0.

Quay trở lại bài toán ban đầu. Gọi xâu cần dựng là xâu T. Ta xét mọi vị trí i là vị trí đầu tiên mà bit của T khác với S. Vì T có thứ tự từ điển lớn hơn S nên chắc chắn bit thứ i của T phải được bật. Đầu tiên ta điền mọi bit sau i của T bằng 0. Rõ ràng xâu T lúc này có thứ tự từ điển nhỏ nhất đảm bảo được mọi vị trí ở trước i đều là vị trí đẹp, và lúc này ta chỉ cần sửa các vị trí phía sau i. Ta duyệt các vị trí j ở sau i, mỗi lần duyệt ta duy trì 1 ràng buộc rằng T có thứ tự từ điển nhỏ nhất thỏa mãn các vị trí từ 1 đến j là vị trí đẹp. Nếu $T_{1..j-1} > T_{j...n}$ (j không phải vị trí đẹp) ta đặt $T_{j...n} = T_{1...j-1}$ và thay $T_{j...n}$ bằng xâu kế tiếp của chính nó. Dễ thấy $T_{j...n}$ lúc này đã có thứ tự từ điển lớn hơn $T_{1...j-1}$ (j trở thành vị trí đẹp) mà không ảnh hưởng gì đến các vị trí trước đó và sau mọi bước duyệt j, T thỏa mãn mọi ràng buộc đề bài. Nếu không tìm được xâu kế tiếp thỏa mãn, vị trí i mà ta đang duyệt không có lời giải, và ta tiếp tục duyệt vị trí i khác. Đáp số sẽ bằng xâu có thứ tự từ điển nhỏ nhất trong các lời giải ở từng vị trí i đã duyệt. Nếu không có lời giải nào, in ra ‘\n’.

**Đpt $O(n^4)$**

Code: http://cpp.sh/55rsd

# B. Job Scheduling

## Tóm tắt đề

Cho n truy vấn (D[i], T[i]), D[i] là deadline của công việc thứ i và T[i] là thời gian để hoàn thành công việc đó. Gọi A[i] là độ trễ khi hoàn thành công việc thứ i. Với mỗi truy vấn in ra Max(A[1], A[2], ... , A[i]).

## Lời giải

Tham lam

Nhận xét: Nếu D[i] < D[j] thì t nên làm công việc thứ i trước công việc thứ j.

Gọi sum[i] là tổng thời gian để hoàn thành mọi công việc có D[i] < k.

Gọi cnt[i] = $\sum T[j]$ với D[j] = i.

=> sum[i] = $\sum cnt[j]$ với 1 <= j <= i.

A[i] = sum[D[i]] - D[i] (A[i] có thể < 0, lúc này công việc được hoàn thành trước deadline)

Với mỗi truy vấn chúng ta có thể dễ dàng tính sum[i] từ 1 đến n với độ phức tạp O(maxN)

**Độ phức tạp:** O(n * 1e4)

# C. Teleportation

## Tóm tắt đề

Cho 2 điểm A và B có lần lượt là (x1, y1) và (x2, y2). Hỏi xem có thể di chuyển từ A đến B mà sử dụng một dãy các chuyển động: Từ (x, y) có thể đi đến (x + y, y); (x – y, y); (x, y + x); (x, y – x)

## Lời giải

Gọi d = GCD(x, y) trong đó GCD(a, b) là số nguyên dương lớn nhất mà a và b đều chia hết.

Với mọi thao tác chuyển động thì d đều không bị thay đổi nên sau một số lần chuyển động nhất định thì điểm (x, y) sẽ di chuyển đến điểm (d, d).

Vậy nếu điểm GCD(x1, y1) = GCD(x2, y2) thì hai điểm (x1, y1) và (x2, y2) đều sẽ di chuyển về 1 điểm hay nói cách khác có thể di chuyển từ (x1, y1) đến (x2, y2), in ra “YES”, còn nếu không thì in ra “NO”.

Độ phức tạp của GCD(a, b) là O(log(max(a, b)))

Nên độ phức tạp của bài sẽ là O(T * log(max(a, b)))

# D. Prime Number Lover

## Tóm tắt đề

Cho một mảng gồm n số nguyên trong khoảng 1 tới 1e9. Một đoạn con gọi là đẹp nếu số lượng số nguyên tố trong đoạn lớn hơn hoặc bằng số lượng số còn lại. Đếm số lượng đoạn con đẹp.

## Lời giải

- Chắc chắn không thể thử từng cặp (l,r) (l <= r) được rồi cho dù ta có thể kiểm tra đoạn (l,r) là đẹp hay không với độ phức tạp O(1) vì số lượng cặp (l,r) có thể đạt đến n * (n – 1) / 2.
- Suy nghĩ:
    - Kiểm tra một số có phải số nguyên tố hay không. (1)
    - Có cách nào để kiểm tra nhanh xem một đoạn (l,r) đẹp hay không. (2)
    - Tìm cách đếm xem có bao nhiêu giá trị r thỏa mãn một giá trị l nào đó. (3)
- (1): Để kiểm tra số a[i] có phải là số nguyên tố hay không
    - Cách 1: Duyệt các số từ 2 tới căn a[i] nếu a[i] chia hết cho số nào đó trong đấy thì a[i] không phải số nguyên tố. Tổng độ phức tạp để kiểm tra hết n số tối đa là O(n * căn 1e6), đạt tối đa = 1e5 * 1e3 = 1e8 (có thể chấp nhận được).
    - Cách 2: sử dụng thuật toán sàng nguyên tố Eratos (không hiểu thì tra Google chứ thuật toàn này giảng giải khá là dài hơi) (độ phức tạp là O(1e6 * log(1e6))).
- (2): Kiểm tra một đoạn (l,r) đẹp hay không
    - Có nhiều cách lắm và thậm chí có cách nhanh hơn cách của tôi. Tuy nhiên, tôi chọn cách này để phục vụ giải quyết vấn đề (3).
    - Ta đánh số lại mảng a, nếu a[i] là số nguyên tố thì ta đánh số a[i] = 1, ngược lại a[i] = -1.
    - Tổng các giá trị từ a[l] tới a[r]: nếu lớn hơn hoặc bằng 0 đồng nghĩa với việc số lượng số nguyên tố trong đoạn (l,r) lớn hơn những số còn lại (tự chứng minh). Làm thế nào để tính tổng a[l] tới a[r] nhanh?
    - Tạo mảng b, b[i] được tính theo công thức b[i] = b[i-1] + a[i]. Tổng giá trị từ a[l] tới a[r] bằng b[r] – b[l-1]. => b[r] – b[l-1] lớn hơn bằng 0 thì thỏa mãn => b[r] lớn hơn bằng b[l-1] thì thỏa mãn.
- (3): Có bao nhiêu giá trị r có thể tạo thành bộ thỏa mãn với một giá trị l nào đó
    - Gọi f[l] là số lượng vị trí r ứng với l.

    - Từ (2) => f[l] = số lượng giá trị b[r] lớn hơn b[l-1].
    - Vì abs(a[i]) = 1 nên giá trị b[i] lớn nhất và nhỏ nhất hơn kém nhau không quá n đơn vị vì thế ta có thể đếm số lượng b[r] lớn hơn b[l] bằng cấu trúc dữ liệu nào đó (ở đây tôi dùng BIT vì tốc độ chạy và code nhanh, sẽ không nói sâu hơn về kĩ thuật BIT – Binary index tree).
    - Để tính f[l], ta Update mọi giá trị b[i] (l <= i <= n) vào BIT (tăng khoảng từ b[i] trở lên thêm 1 đơn vị, tức là khi ta Query phần tử thứ x trong BIT, ta sẽ đếm được số lượng giá trị b[i] nhỏ hơn hoặc bằng x). Sau khi Update hết, ta Query b[l-1] để đếm số lượng b[i] nhỏ hơn và kết quả ứng với l là số lượng phần tử từ l tới r trừ cho Query b[l-1] (f[l] = r – l + 1 – Query(b[l-1])). Tại sao phải tính phần bù? Hiểu rõ về BIT là có đáp án (thực ra có cách không cần phải tính phần bù nhưng cách này code nhanh hơn).
    - Nhận xét là có thể sử dụng những dữ liệu khi tính f[l] để tính f[l-1] (chỉ cần Update thêm b[l-1] vào BIT). Ta sẽ duyệt ngược từ n, mỗi lần sẽ Update b[i] và tính f[i] bằng công thức f[i] = n – i + 1 – Query(b[i-1]).  Như vậy thì mỗi lần sẽ mất độ phức tạp O(logn) và tổng độ phức tạp là O(nlogn).
- Kết quả là tổng các f[i].
- Tổng độ phức tạp của chương trình bằng độ phức tạp để giải quyết vấn đề (1) + giải quyết vấn đề (3).
- Chú ý:
    - Để tiện lợi cho sử dụng BIT, ta nên tịnh tiến b[i] lên sao cho b[i] nhỏ nhất phải lớn hơn 0 (b[i] có thể nhỏ hơn hoặc bằng 0 vì a[i] có thể bằng -1).
    - Ta thiết lập cho phần tử b[0] vì khi tính f[1], ta có thực hiện Query(b[0]).

# E. Repeated Subarray

## Tóm tắt đề

Cho một dãy A = ( $a_1, a_2, ..., a_n$ ) với độ dài n và q truy vấn có dạng l, r, x, y yêu cầu trả lời số lượng giá trị riêng biệt sao cho các giá trị được lặp lại từ x đến y lần trong đoạn con ( $a_l, a_{l+1}, ... a_r$ )

Code: http://cpp.sh/2sm45

## Lời giải

- Đầu tiên ta thực hiện nén số trên dãy A ví dụ A = (1000, 2, 69, 5) sẽ thành (4, 1, 3, 2)
- Sort tất cả các truy vấn theo Mo's Algorithm
- Gọi D(x) = đếm số lần xuất hiện của x
- Gọi S(x) là số giá trị xuất hiện ≤ x lần
- Ta sẽ thêm xóa một số trong O(1) như sau:
    - Thêm 1 số x:
        - Tăng D(x) thêm 1
        - Vì D(x) tăng 1 nên S(D(x) + 1) không đổi và S(D(x)) giảm 1
    - Xoá 1 số x:
        - Giảm D(x) đi 1
        - Vì D(x) giảm 1 nên S(D(x)) không đổi và S(D(x) - 1) tăng 1

# F. National Road System

## Tóm tắt đề:

Cho một cây N đỉnh với Q truy vấn (N, Q <= 50 000). Mỗi truy vấn có một danh sách các đỉnh (u1, u2, …, um) phủ xung quanh chính nó với bán kính (d1, d2, …, dm). Tính số lượng đỉnh phân biệt bị phủ bởi ít nhất 1 đỉnh. Tổng m trong tất cả các truy vấn <= 500, 000.

Link bài: http://icpcvncentral.tk/public/practice_problem.php?id=I19CK

Code: https://ideone.com/ZuCf79

## Lời giải

Lần lượt giải quyết các trường hợp nhỏ sau:

- Với **m = 1**, đơn giản là tính số đỉnh cách u1 một khoảng <= d1. Có thể dễ dàng tính được số đỉnh thỏa mãn điều kiện này với centroid decomposition. Và để thuận tiện, **gọi số đỉnh thỏa mãn điều kiện này là**
  $F(u1,\ d1)$**.**
- Với **m = 2**, dễ thấy sẽ có những trường hợp đáp án không phải là $F(u1,\ d1)\ +\ F(u2,\ d2)$. Giả sử chúng ta có thể tìm được một đỉnh p trên cây sao cho $d1\ -\ dist(p,\ u1)\ =\ d2\ -\ dist(p,\ u2)$ . Dễ thấy kết quả của bài toán là $F(u1,\ d1)\ +\ F(u2,\ d2)\ -\ F(p,\ d1\ -\ dist(u,\ p))$. Trong trường hợp không tìm được đỉnh p như ý. Ta có các trường hợp sau:
  - Một trong 2 đỉnh u1, u2 ***bao hàm*** đỉnh còn lại. Hay nói cách khác là $d1\ -\ dist(u1,\ u2)$ □ $d2$. Trường hợp này đơn giản là ta bỏ qua đỉnh u2, và chỉ tính kết quả cho đỉnh u1.
  - Tính chẵn lẻ của d1, d2 và dist(u1, u2) không ủng hộ. Hay nói cách khác là tồn tại 2 đỉnh x và y kề nhau sao cho

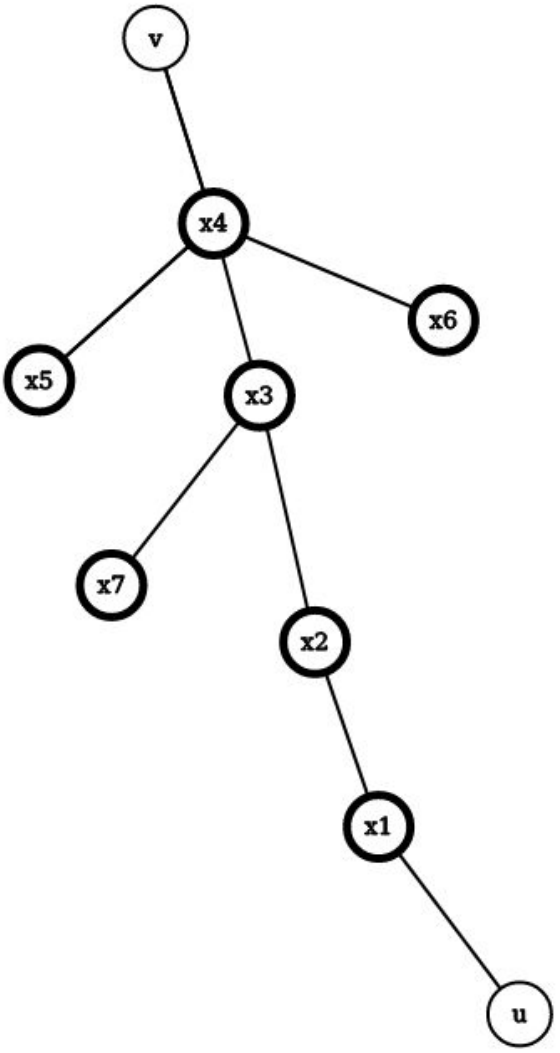
$d1 - dist(u1, x) = d2 - dist(u2, y)$. Để giải quyết trường hợp này, với mỗi cạnh trên cây gốc, ta sẽ sử dụng ý tưởng của manacher, tạo một đỉnh ảo nối tới 2 mút của cạnh thay vì cạnh gốc. Thay thế tất cả di với $di * 2$. Điều này đảm bảo rằng ta sẽ luôn tìm được nút p thỏa mãn.

Với **m > 2**, ta thấy cái khó của trường hợp này nằm ở chỗ, ta không thể tính được phần chung của các tập một cách dễ dàng như với m = 2. Từ đây mình có một ý tưởng như sau.

Dựng *cây nén* của (u1, u2, …, um). Cây nén này sẽ bao gồm tất cả các đỉnh (u1, u2, …, um) và LCA của 2 đỉnh liên tiếp trong thứ tự dfs. Dễ thấy cây nén sẽ có tối đa 2m đỉnh và 2m - 1 cạnh.

Với mỗi đỉnh v trong cây nén, gọi $best[v]$ là $MAX(di - dist(v, ui))$ tức là một nút ui nào đó phủ đến v và cho phép v được mở rộng tiếp một lượng lớn nhất là best[v]. Để tính mảng best cho tất cả các đỉnh trong cây nén, ta có thể làm 1 lần dp bottom up và tiếp đó là topdown.

Key idea của mình là với mỗi cạnh (u, v) trên cây nén, ta sẽ giải quyết phần **subgraph** (hình minh họa) tạo bởi các đỉnh nằm trên đường đi giữa u và v dựa vào best[u] và best[v].




Các đỉnh **x** được tô đậm là các đỉnh trong **subgraph** được nhắc đến.

**X4** là **con trực tiếp** của v và là **tổ tiên** của u.

Bài toán với các **subgraph:**

Dễ thấy khi chúng ta thu hẹp bài toán về 1 cạnh (u, v), việc tính phần bù, trừ giờ đây sẽ dễ hơn rất nhiều.

Trước hết, ta sẽ đi tìm số đỉnh phân biệt được phủ bởi (u, best[u]) và (v, best[v]). Đây chính là bài toán với m = 2 đã được giải quyết ở trên.

Giờ ta sẽ phải trừ đi những đỉnh nằm ngoài **subgraph**.

Không mất tính tổng quát, giả sử v là cha của u trên cả cây gốc và cây nén. Dễ thấy những đỉnh nằm ngoài **subgraph** đã bị tính vào kết quả bao gồm:

1. Những đỉnh nằm trong subtree của u và cách u một khoảng không quá best[u].
2. Những đỉnh nằm ngoài subtree **x4** (trong hình vẽ) và cách v một khoảng không quá best[v].

Gọi g(u, k) là số đỉnh nằm trong subtree u cách u một khoảng không quá k. Để tính được g(u, k). Ta có thể coi subtree của u là một đoạn thẳng trong thứ tự duyệt cây, và khoảng cách từ một đỉnh con w đến u không vượt quá k nếu $depth(w) \leq depth(u) + k$. Khi đó ta sẽ có một truy vấn có dạng (l, r, x, y) truy vấn số điểm nằm trong một hình chữ nhật. Hàm này có thể dễ dàng giải quyết trong QlogN bằng cách xử lí offline.

Như vậy với trường hợp 1, kết quả của phần bị trừ đi này là g(u, best[u]).
Với trường hợp 2: Dễ thấy kết quả của phần bị trừ đi này là
$F(v, best[v]) - G(x4, best[v] - 1)$.

Với những đỉnh v nằm trong cây nén. V sẽ được tính vào kết quả nếu best[v] >= 0.

Còn lại những đỉnh nằm ngoài các **subgraph** đã tính đến. Với mỗi đỉnh v, ta cộng vào kết quả 1 lượng là $G(v, best[v])$ và xét các cạnh (u, v) như hình trên,

trừ đi một lượng là $G(x4,\ best[v]\ -\ 1)$, hay nói cách khác là những nhánh đã được xét trong cây khung xương.

Nếu chỉ nhìn qua, nhận thấy số lượng lần cần tính hàm F và G của solution này là khá nhiều và chi phí để tính 1 hàm là O(logN), rất khó để AC với 2s time limit.

Tuy nhiên, nếu nháp kĩ, ta sẽ thấy các hàm F và G trừ khử nhau khá nhiều. Dẫn đến việc ta chỉ cần tính tất cả các F(u, best[u]) và F(mid) (nếu có).

Về phía hàm G, các hàm G tự triệt tiêu nhau gần như hoàn toàn và chỉ còn lại duy nhất G(root, best[root]) với root là gốc của cây khung xương. Nếu ta coi 1 (gốc của cây gốc) luôn nằm trong cây khung xương, thì ta đơn giản là tính số lượng đỉnh có độ cao <= best[1]. Do đó **việc cài đặt truy vấn offline 2 chiều là không cần thiết.** Hàm G chỉ đóng vai trò giúp tiếp cận bài toán 1 cách dễ dàng hơn.

Bỏ qua phần nháp như một exercise nhỏ, mình sẽ tổng hợp kết quả hàm $F$ của 4 trường hợp với một cạnh (u, v) trên khung xương như sau:

1. u và v không giao nhau: $F(u,\ best[u])$
2. u bao hàm v: $F(u,\ best[u])\ -\ F(v,\ best[v])$
3. v bao hàm u: $0$
4. u và v giao nhau nhưng không bao hàm nhau:
   $F(u,\ best[u])\ -\ F(mid,\ best[u]\ -\ dist(u,\ mid))$

# G.

## Tóm tắt đề

Có g nhóm, mỗi nhóm gồm n học sinh. Có n * g gói kẹo, gói thứ i gồm $a_i$ viên kẹo. Ta cần chia các gói kẹo này cho g nhóm sao cho:

- Mỗi nhóm nhận đúng n gói kẹo.
- Với mỗi nhóm, ta cần lấy lại một số lượng viên kẹo nhỏ nhất có thể, sao cho chỗ kẹo còn lại trong nhóm có thể chia đều cho n thành viên (nói cách khác, ta cần lấy lại một lượng kẹo bằng tổng số viên kẹo của nhóm mod n). Ta cần chia sao cho tổng số kẹo cần lấy lại này là nhỏ nhất có thể.

Giới hạn: $1 \le n * g \le 50000$; $1 \le a_i \le 10^9$

## Lời giải

### Thuật toán chuẩn

Trước hết, ta giải quyết bài toán với g = 2. Khi đó, bài toán trở thành: Cho dãy a gồm 2n số nguyên dương, chia dãy trên thành hai dãy con $s_1$ và $s_2$ sao cho:

- Mỗi dãy con gồm đúng n số nguyên
- Kí hiệu sum(p) là tổng giá trị các số nguyên trong dãy p. Gọi S = sum($s_1$) mod n + sum($s_2$) mod n. Ta cần chia sao cho giá trị của S là nhỏ nhất có thể.

Ta nhận xét rằng: $S \equiv sum(s_1) + sum(s_2) \equiv sum(a) \pmod n$. Do đó, giá trị S nhỏ nhất có thể là sum(a) mod n.

Đồng thời, theo định lý Erdős–Ginzburg–Ziv, với mỗi dãy gồm 2n - 1 phần tử, ta luôn có thể chọn một dãy con gồm n phần tử có tổng chia hết cho n. Do đó, ta có thể chọn $s_1$ sao cho sum($s_1$) mod n = 0 và $s_2$ gồm các phần tử còn lại. Khi đó, S = sum($s_2$) mod n = sum(a) mod n (giá trị nhỏ nhất có thể của S).

Bài toán trở thành: Cho dãy $a_1$, $a_2$, …, $a_{2n-1}$, tìm một dãy con s bất kì gồm n phần tử có tổng chia hết cho n.

#### Lời giải với g = 2, n là số nguyên tố

Trước hết, ta sẽ giải bài toán trên với n là số nguyên tố. Ta cần chứng minh bổ đề sau:

**Bổ đề**

Cho một số nguyên tố n bất kì và 2 ≤ k ≤ n. Với một dãy $a_1$, $a_2$, …, $a_{2k-1}$ bất kì không có k phần tử giống nhau, gọi S là tập hợp được xây dựng bằng cách sau: Với dãy con p gồm k phần tử của a, thêm sum(p) mod n vào S. Khi đó, |S| ≥ k.

**Chứng minh**

(Nguồn từ https://mathoverflow.net/a/16733)

Ta sẽ chứng minh bổ đề trên bằng phương pháp chứng minh quy nạp.

*Cơ sở quy nạp (k = 2):*

Ta nhận xét rằng, dãy $a_1$, $a_2$, $a_3$ không thể gồm cả 3 phần tử giống nhau (theo giả thiết). Không mất tính tổng quát, giả sử $a_1 \neq a_2$. Khi đó, hai dãy con $\{a_1, a_3\}$ và $\{a_2, a_3\}$ sẽ có tổng mod n khác nhau. Do đó, |S| ≥ 2 = k.

*Phần quy nạp*

Giả sử kết quả trên đúng với 2 ≤ k < n. Ta sẽ chứng minh kết quả trên đúng với k+1.

Thật vậy. Xét dãy A' = [$a_1$, $a_2$, …, $a_{2k-1}$, b, c]. Không mất tính tổng quát, giả sử b và c là hai phần tử có tần số lớn nhất trong dãy A' (với giả sử trên, b ≠ c).

Trước hết, dãy A = [$a_1$, $a_2$, …, $a_{2k-1}$] sẽ thỏa điều kiện không k phần tử giống nhau (vì nếu không, sẽ có 3 giá trị xuất hiện ít nhất k lần trong dãy A', trong khi dãy A' lại chỉ có 2k+1 phần tử). Do đó, theo giả thiết quy nạp, có thể chọn k dãy con $s_1$, $s_2$, …, $s_k$ có tổng modulo n đôi một phân biệt. Gọi:

- B = {sum($s_1$) + b, sum($s_2$) + b, …, sum($s_k$) + b}
- C = {sum($s_1$) + c, sum($s_2$) + c, …, sum($s_k$) + c}

Ta sẽ chứng minh rằng B ≠ C. Giả sử B = C, ta có:

$$sum(B) \equiv sum(C) \ (mod\ n)$$

$$\Leftrightarrow \sum_{i=1}^{k} sum(s_i) + kb \equiv \sum_{i=1}^{k} sum(s_i) + kc \ (mod\ n)$$

$$\Leftrightarrow kb \equiv kc \ (mod\ n)$$

$$\Leftrightarrow k(b-c) \equiv 0 \ (mod\ n)$$

Do k < n nên k $\not\equiv$ 0 (mod n). Mặt khác, b ≠ c nên (b - c) $\not\equiv$ 0 (mod n). Do n là số nguyên tố, k(b - c) $\not\equiv$ 0 (mod n), mâu thuẫn với kết quả trên.

Do B ≠ C, B ∪ C sẽ có ít nhất k+1 phần tử. Nói cách khác, trong các dãy con $s_1$ + [b], $s_2$ + [b], …, $s_k$ + [b], $s_1$ + [c], $s_2$ + [c], …, $s_k$ + [c], sẽ có ít nhất k + 1 dãy con có tổng modulo n khác nhau. Từ đó, |S| ≥ k+1.

Theo nguyên lí quy nạp toán học, kết quả trên đúng với mọi 2 ≤ k ≤ n. ■

Với bổ đề trên, ta có thể giải bài toán trên như sau:

Nếu có n phần tử giống nhau, ta chỉ cần chọn dãy con s gồm n phần tử đó. Ngược lại, từ dãy a, ta xây dựng dãy b như đoạn giả mã sau:

```
b = []
FOR i = 1 TO n-1
BEGIN
      x = phần tử có tần số lớn nhất của dãy a
      y = phần tử có tần số lớn thứ nhì của dãy a
      Thêm x, y vào đầu dãy b
      Xóa x, y khỏi dãy a
END

Thêm phần tử còn lại của dãy a vào đầu dãy b
```

Từ bổ đề trên, ta có hệ quả sau:

Tồn tại một dãy con s = $s_1$, $s_2$, …, $s_n$ có tổng chia hết cho n sao cho

- $s_1 = a_1$
- $s_2 = a_2$ hoặc $s_2 = a_3$
- $s_3 = a_4$ hoặc $s_3 = a_5$
- …
- $s_n = a_{2n-2}$ hoặc $s_n = a_{2n-1}$

Với bổ đề trên, ta có thể tìm dãy s bằng phương pháp quy hoạch động sau:

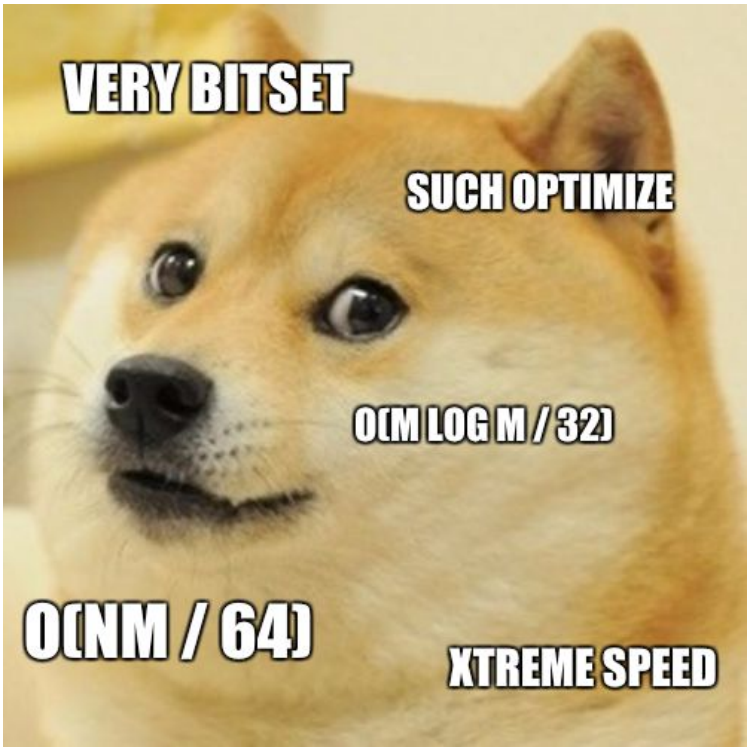
Gọi dp[i][r] = true nếu tồn tại một dãy $s_1$, $s_2$, …, $s_i$ thỏa điều kiện trên và có tổng mod n bằng r.

Công thức quy hoạch động:

- dp[1][$b_1$] = true
- dp[i][r] = dp[i-1][(r - $b_{2i-2}$) mod n] | dp[i-1][(r - $b_{2i-1}$) mod n] với 2 ≤ i ≤ n, 0 ≤ r < n.

Thuật toán trên có độ phức tạp về thời gian và bộ nhớ là $O(n^2)$, không đáp ứng được giới hạn n ≤ 50000/g = 25000. Đối với ngôn ngữ C++, ta có thể cải

tiến thuật toán trên bằng cách sử dụng kiểu dữ liệu bitset (ý tưởng của bitset là dùng kiểu dữ liệu số nguyên 32-bit để biểu diễn cùng lúc 32 biến boolean. Chi tiết về bitset các bạn có thể tham khảo bài viết sau: http://www.cplusplus.com/reference/bitset/bitset/).




Trước hết, ta cần viết hàm shrMod(mask, x) (mask là một bitset gồm n bit) để dịch chuyển vòng tròn tất cả các bit 1 của mask sang phải x bit. Ví dụ:

- shrMod(011010, 1) = 001101
- shrMod(011010, 2) = 100110
- shrMod(011010, 3) = 010011
- shrMod(011010, 4) = 101001

Khi đó, công thức quy hoạch động có thể được viết lại như sau:

- dp[1] = shrMod(00...00, $b_1$)
- dp[i] = shrMod(dp[i-1], $b_{2i-2}$) | shrMod(dp[i-1], $b_{2i-1}$) với $2 \le i \le n$

Độ phức tạp về thời gian và bộ nhớ của cách làm này là $O(n^2/32)$, đáp ứng được giới hạn đề bài.

### Lời giải với g = 2, n tổng quát

Trường hợp n tổng quát, ta phân tích n thành các thừa số nguyên tố. Giả sử $n = p_1p_2...p_k$ (với $p_i$ là số nguyên tố).

Từ dãy $a_1, a_2, \ldots, a_{2n-1}$, ta có thể tìm $2(n/p_1) - 1$ dãy con rời nhau (mỗi phần tử của a chỉ nằm trong nhiều nhất một dãy con), mỗi dãy con có độ dài $p_1$ và chia hết cho $p_1$. Do $p_1$ là số nguyên tố, ta chỉ việc lấy $2p_1 - 1$ phần tử đầu của dãy a, rồi làm tương tự trường hợp trên. Gọi $s_i$ là dãy con thứ i tìm được.

Ta có thể xem mỗi dãy con $s_i$ như một phần tử có giá trị là $b_i$ = sum($s_i$) mod ($n/p_1$). Từ dãy $b_1, b_2, ..., b_{2(n/p_1)-1}$ , ta cần tìm một dãy con độ dài $n/p_1$ và có tổng chia hết cho $n/p_1$. Giả sử dãy con tìm được là $b_{x_1}, b_{x_2}, ..., b_{x_{n/p_1}}$ . Khi đó, dãy được ghép từ các dãy con $s_{x_1}, s_{x_2}, ..., s_{x_{n/p_1}}$ sẽ có tổng chia hết cho n, và đây chính là dãy cần tìm.

Ta đã đưa bài toán có kích thước n về bài toán có kích thước $n/p_1$. Với bài toán kích thước $n/p_1$, ta tiếp tục thực hiện tương tự như trên để đưa về bài toán có kích thước $n/(p_1p_2)$. Ta lặp lại việc đưa về bài toán con cho đến khi kích thước của bài toán là một số nguyên tố.

Ta gọi T(n) là độ phức tạp của bài toán có kích thước n.

Việc tìm 2m - 1 dãy con rời nhau, mỗi dãy con có độ dài k và chia hết cho k sẽ có độ phức tạp là O(nm/32).

Do đó:

$$T(n) = O(\frac{np_1}{32}) + T(\frac{n}{p_1})$$

$$T(n) = O(\frac{np_1}{32}) + O(\frac{np_2}{32p_1}) + O(\frac{np_3}{32p_1p_2}) + ...$$

$$T(n) = O(\frac{p_1{}^2p_2p_3p_4...p_k}{32}) + O(\frac{p_2{}^2p_3p_4...p_k}{32}) + O(\frac{p_3{}^2p_4...p_k}{32}) + ...$$

Do $p_i \le n$ nên $T(n) = O(\frac{n^2}{32})$

### Lời giải với g, n tổng quát

Với g > 2, ta chỉ việc thực hiện g-1 lần thao tác sau: Chọn 2n - 1 phần tử đầu tiên của dãy a, tìm một dãy con có độ dài n và tổng các phần tử chia hết cho n và xóa các phần tử của dãy con đó ra khỏi dãy a.

Với cách làm trên, g - 1 dãy con đầu tiên sẽ có tổng chia hết cho n, dãy con cuối cùng sẽ có tổng mod n bằng sum(a) mod n. Do đó, cách chia này sẽ cho kết quả tốt nhất có thể.

## Thuật toán random

Chào các bạn, tuy rằng bài G là bài có sol rất trí tuệ, được hỗ trợ bởi một định lí đỉnh cao nhưng bài này có thể giải được bằng random không khó lắm.

Có 2 điều quan trọng trong sol random này:

1) Bạn cần đoán được định lí trên, còn không cần chứng minh nó đúng, lí do ở phần thứ 2
2) Bạn cần nghĩ xem người sinh test thế nào, có sinh để giết được sol random của bạn không. Lúc mình làm thì mình thấy rất khó sinh được test giết sol random của mình nên mình random thôi

Oke, vậy giả sử định lí trong 2*n số luôn chọn được n số có tổng chia hết cho n đúng. Ta sẽ tìm n số này như thế nào? Mục tiêu của chúng ta là kiểm tra được các nhiều subset n số càng tốt, đến khi tìm ra thì thôi. Vậy ta làm như sau đến khi tìm ra bộ số thoả mãn:

- shuffle cả dãy lên
- Với mỗi đoạn n+1 số liên tiếp trong dãy, kiểm tra xem có số nào đồng dư với tổng n+1 số này không(O(1) dùng mảng đánh dấu), nếu có thì dãy n+1 số xoá số này chính là dãy cần tìm

Tại sao cách này rất tốt? Vì với 1 đpt O(n) bạn có thể kiểm tra được n^2 subset khác nhau, và trong thực tế thì tất cả các test mình sinh ra đều không cần chạy quá 2 lần shuffle. Bonus: Nếu các bạn chỉ check các dãy n số liên tiếp thì thuật sẽ chạy rất lâu do 1 lần O(n) các bạn chỉ kiểm tra được n subset.

Oke, code xong nộp lên AC ngay, mất 20ms. Vậy là mình đã giải được bài này với “cảm giác” là định lí kia đúng và việc thấy giám khảo khó có cách sinh test nào counter được.

Bonus: Nhờ anh Nguyễn Đinh Quang Minh có chỉ ra 1 test counter được thuật random trên, đó là n = 50000, k=2, n/2 số = 1 và n/2 số = 25000. Dễ thấy với test này, chỉ có đúng 1 cách phân hoạch thoả mãn. Vậy fix sol random như thế nào? Trước khi random sort lại dãy 1 lần và chạy kiểm tra trước thì sẽ qua được phần lớn các test sinh cố tình.

# H. Boring Bits

## Tóm tắt đề:

Yêu cầu đếm xem có bao nhiêu xâu nhị phân (chỉ có kí tự ‘1’ và ‘0’) độ dài n mà chỉ có nhiều nhất K kí tự ‘0’ liên tiếp.

## Lời giải

### Quy hoạch động – *O(NK)*

Gọi dp(i, j) là số xâu nhị phân thỏa mãn đề bài có độ dài i và có j kí tự ‘0’ ở phía cuối. Khi đó ta có:

Với i = 1 thì dp(i, j) = 1 nếu j < 2 và dp(i, j) = 0 nếu $j \geq 2$

Với i > 1 thì

dp(i, j) = $\sum_{t=0}^{K} dp(i-1, t)$ với j = 0 (đặt kí tự ‘1’ ở vị trí thứ i)

dp(i, j) = dp(i – 1, j – 1) với j > 0 (đặt kí tự ‘0’ ở vị trí thứ i)

Đáp án là $\sum_{t=0}^{K} dp(n, t)$

### Nhân ma trận – $O(K^3 \log N)$

Gọi $T_i$ là ma trận chứa các giá trị [dp(i, 0), dp(i, 1),…, dp(i, k)]. Ma trận $T_{i+1}$ sẽ được tính từ $T_i$ bằng cách nhân với ma trận M nghĩa là $T_{i+1} = T_i * M$.

Dựa vào công thức quy hoạch động trên, ta xây dựng ma trận M kích thước (k + 1) * (k + 1) như sau: nếu j = 0 hoặc (j > 0 và i = j – 1) thì $M_{i,j} = 1$ còn các trường hợp còn lại $M_{i,j} = 0$

Khi đó, $T_n = T_1 * M^{n-1}$

# I. Greatest Common Divisor

## Tóm tắt đề bài

Cho 1 dãy $a_1, a_2, ..., a_n$ bao gồm n phần tử. Có q truy vấn, truy vấn thứ i cho 1 số $q_i$ và yêu cầu đếm số đoạn $(l, r)$ sao cho $gcd(a_l, a_{l+1}, ... a_r) = q_i$

## Lời giải

Cố định 1 vị trí l, ta xét 1 vị trí r bất kì.

Đặt $A = gcd(a_l, a_{l+1}, ... a_r)$, $B = gcd(a_l, a_{l+1}, ... a_{r+1})$.

Ta dễ thấy được rằng $gcd(A, a_{r+1}) = B$ nên chắc chắn B sẽ phải là ước của A.

Nếu A > B rõ ràng A ≥ B * 2, vì vậy các số phần tử khác nhau trong tập $\{gcd(a_l, a_{l+1}, ... a_r | r \geq l)\}$ sẽ không vượt quá $log\ a_l$

→ Nếu xét tất cả mọi vị trí l, tổng số phần tử khác nhau trong tập $\{gcd(a_l, a_{l+1}, ... a_r) | r \geq l\}$ sẽ không vượt quá $nlog\ a_{max}$.

Do đó, ta hoàn toàn có thể lưu hết số lần xuất hiện các giá trị này vào 1 cái map. Ta duyệt mọi vị trí l, ban đầu đặt r = l, ta tiến hành nhảy trên 1 cái bảng sparse-table đã được chuẩn bị trước để tìm vị trí x nhỏ nhất sao cho $gcd(a_l, a_{l+1}, ... a_r) > gcd(a_l, a_{l+1}, ... a_x)$ như vậy ta biết được giá trị $gcd(a_l, a_{l+1}, ... a_r)$ đã xuất hiện thêm (x – r) lần, ta đặt r = x và tiếp tục tiến hành việc nhảy. Số lần nhảy sẽ không vượt quá $log\ a_{max}$ và số bước mỗi lần nhảy không vượt quá $log\ n$

Độ phức tạp: O($n * log\ amax * log\ n$)

Code: http://cpp.sh/7pff4

# J. Easy Practice Problem

## Tóm tắt đề

Cho n điểm a[0] , a[1] , a[2] , … , a[n - 1]. Giá trị của 1 tập n điểm là bình phương khoảng cách manhattan lớn nhất giữa 2 trong số n điểm. Ta gọi A là tập ngẫu nhiên k điểm trong n điểm hỏi giá trị kỳ vọng của tập này là bao nhiêu.

## Lời giải

Nếu k = 1 hiển nhiên đáp án = 0. Ta xét k ≥ 2.

Dễ thấy khoảng cách giữa 2 điểm (x , 2019 - x) và (y , 2019 - y) là $5(x - y)^2$. Như vậy khi ta sắp xếp a tăng dần thì khi chọn 2 vị trí i , j ta thấy số cách để chọn sao cho giá trị của tập k phần tử mà có giá trị là $5\,(a[j] - a[i])^2$ là $C_{j-i-1}^{k-2}$.

Gọi $S(len) = \sum_{i=0}^{n-len-1} (a[i] - a[i + len])^2$

→ Đáp án cuối cùng = $\frac{5 * \sum_{len=k-1}^{n-1} S(len) * C_{len-1}^{k-2}}{C_n^k}$

$$S(len) = \sum_{i=0}^{n-len-1} (a[i]^2 + a[i + len]^2) - \sum_{i=0}^{n-len-1} 2 * a[i] * a[i + len]$$

$\sum_{i=0}^{n-len-1} (a[i]^2 + a[i + len]^2)$ có thể được tính dễ dành với tổng cộng dồn. Bài toán còn lại là tính $\sum_{i=0}^{n-len-1} 2 * a[i] * a[i + len]$.

Để tính được giá trị này ta sử dụng FFT như sau.Ta có 2 đa thức

$$A(x) = a[0]\,x^0 + a[1]\,x^1 + \cdots + a[n-1]\,x^{n-1}$$

$$B(x) = a[n-1]\,x^0 + a[n-2]\,x^1 + \cdots + a[0]\,x^{n-1}$$

$$C(x) = A(x) * B(y) = c_0\,x^0 + c_1\,x^1 + \cdots + c_{2n-2}\,x^{2n-2}$$

Khi đó $2 * \sum_{i=0}^{n-len-1} a[i] * a[i + len]$ = 2 * $c_{n-1-len}$ .ĐPT O(nlogn).

# K. New Sticks Game

## Tóm tắt đề

Hai bạn Bách, Khoa chơi trò chơi như sau: Ban đầu có n que gỗ, hai người lần lượt lấy một số lượng que gỗ thuộc các giá trị sau:

1. 1
2. Số lượng số nguyên dương không lớn hơn m nguyên tố cùng nhau với m.
3. Số chữ số của m.

mà ở đó, m là số lượng que gỗ ở lượt chơi hiện tại. Người lấy được cái que cuối cùng là người chiến thắng. Hãy xác định với mỗi giá trị n, ai là người chiến thắng, biết rằng cả hai người đều chơi tối ưu.

## Lời giải

- Gọi F(n) là trạng thái thắng/thua của người chơi có n que gỗ để bốc khi đến lượt mình.
- Gọi D(n) là số lượng chữ số của n.
- Khi n = 0: Dễ thấy rằng F(0) = false vì theo đề, người bốc que cuối cùng là người chiến thắng hay nói cách khác người chơi nào không còn que để bốc là người thua cuộc.
- Khi n > 0: Theo đề, người chơi sẽ có 3 lựa chọn để bốc và để ý rằng, lựa chọn số 2 chính là giá trị của φ(n) (phi hàm euler). Người chơi sẽ thua khi và chỉ khi cả 3 trạng thái của lượt tiếp theo đều là trạng thái thắng (do lúc đó đến lượt người còn lại bốc), nghĩa là nếu cả ba giá trị F(n – 1), F(n – φ(n)), F(n – D(n)) đều mang giá trị true, người chơi sẽ thua. Từ đây, ta có công thức truy hồi sau:

$$F(n) = \neg(F(n-1) \wedge F(n-\varphi(n)) \wedge F(n-D(n))$$

- Công thức truy hồi trên có thể dễ dàng cài đặt trong O(n).
- Việc còn lại chỉ là tính phi hàm Euler một cách hiệu quả. Do giá trị của n ≤ $10^6$ nên ta có thể chuẩn bị trước φ(n) trong O($10^6$ * log$10^6$) bằng sàng.

Code mẫu: https://ideone.com/UtQIPb

# L. Wheel of fortune

## Tóm tắt đề

A và B chơi một trò chơi kì diệu. Khi quay A có xác suất được i điểm (1 <= i <= N) là $a_i$ , B có xác suất được i điểm là $b_i$ . A chơi trước, giả sử quay được X điểm. A có thể dừng lại hoặc quay tiếp. Giả sử quay tiếp được Y điểm, A sẽ nhận được (X + Y) điểm nếu (X + Y <= N) và (X + Y - N) điểm nếu (X + Y > N). Tới lượt của B, B cũng quay và tính điểm giống A. Hết lượt của B, ai nhiều điểm hơn người đó thắng, nếu bằng điểm nhau thì chơi tiếp. Hỏi xác suất A thắng là bao nhiêu, biết rằng xác suất trò chơi diễn ra vô tận là bằng 0.

## Lời giải

- Để giúp A đưa ra quyết định quay tiếp hay không, đầu tiên ta tính xác suất có điều kiện sau: probB(x): xác suất B thắng nếu biết điểm của A là x. Ta tính xác suất trên bằng cách duyệt qua điểm của B trong lần quay đầu và lấy max của trường hợp dừng lại và quay tiếp.
- Khi đã có probB, ta tính kết quả của cả bài toán một cách tương tự, bằng cách duyệt qua điểm của A trong lần quay đầu, lấy max của trường hợp dừng lại và quay tiếp.
- Để ý rằng khi tính các xác suất trên, ta sẽ gặp vấn đề: trường hợp hòa thì xử lý như thế nào? Gọi xác suất A thắng cả trò chơi là p. Khi ta xét một trường hợp mà kết quả là hòa, xác suất mà A thắng trong trường hợp đó sẽ là p (vì trò chơi lặp lại). Ta đang tính p, nhưng ta lại cần có p. Vậy ta sẽ làm như sau: ta giả sử p = x, sau đó ta tính toán như trên và kiểm tra xem giá trị cuối cùng ta tính ra được có phải là x không.
- Ý tưởng trên đưa ta đến thuật toán chặt nhị phân. Gọi xác suất ta đang kiểm tra là x, xác suất ta tính được là y.
    - Nếu y = x: đây chính là kết quả bài toán.
    - Nếu y > x: xác suất ta đang tìm lớn hơn x, vì vậy ta bỏ đi khoảng nhỏ hơn x.
    - Nếu y < x: xác suất ta đang tìm nhỏ hơn x, vì vậy ta bỏ đi khoảng lớn hơn x.
- Gọi số bước chặt nhị phân là K, ta có độ phức tạp: $O(n^3 * K)$