Overview

Bài	#submit	#AC	Độ khó (Đánh giá của BTC)	AC đầu tiên
A	185	11	2	Phút 56, đội **HCMUS-NyanCat**
B	11	0	3	
C	8	0	4	
D	532	184	0	Phút 11, đội **int** (VNU)
E	120	45	3	Phút 17, đội **HCMUS-NyanCat**
F	27	5	3	Phút 185, đội **HCMUS-GeomeSmile**
G	155	44	2	Phút 70, đội **unsigned** (VNU)
H	7	3	4	Phút 129, đội **RadixSort** (VNU)
I	584	76	1	Phút 20, đội **HCMUS-Miracle**
J	291	161	1	Phút 12, đội **RadixSort** (VNU)
K	35	6	2	Phút 206, đội **MergeSort** (VNU)
L	2	0	1	

Bộ đề được chuẩn bị bởi: PTIT, HCMUP, Phạm Văn Hạnh, Nguyễn Thành Trung - RR, Lăng Trung Hiếu, Lê Yên Thanh, Lê Đôn Khuê, Hồ Vĩnh Thịnh.

Nếu có thắc mắc về lời giải, bạn có thể hỏi ở: group VNOI.

Hầu hết các thuật toán & kĩ thuật được sử dụng trong lời giải: Segment Tree, mảng cộng dồn, Tham lam, Ternary Search... đều có thể được tìm thấy trên VNOI wiki.

Bạn có thể tham khảo code BTC của tất cả các bài ở đây.

A

Kiến thức cần biết: Segment Tree
Độ khó: 2*
Người ra đề: PTIT

Lời giải 1:

Với giới hạn N <= 10^5, Q <= 10^5 và các thao tác update đoạn và get tổng đoạn quen thuộc, có thể dễ dàng nhận biết đây là một bài sử dụng Segment Tree. Điểm mấu chốt của bài toán là xem ở mỗi nút của Segment Tree ta cần lưu lại những thông tin gì.

- Với thao tác get, ta cần in ra tổng của đoạn [u, v], do đó dễ thấy một nút i quản lý đoạn [l, r] trên Segment Tree chắc chắn phải lưu lại tổng A(l) + ... + A(r).
- Với thao tác update [u, v], ở vị trí i, ta cần tăng thêm
  (i - u + 1) * (i - u + 2) * (i - u + 3)
  = i^3 * 1
  + i^2 * (-3*u + 6)
  + i^1 * (3*u^2 - 12*u + 11)
  + i^0 * (-u^3 + 6*u^2 - 11*u + 6)

  Do với mỗi truy vấn [u, v] thì u là hằng số, nên ta chuyển được truy vấn thành loại đơn giản hơn như sau:
  Với mỗi i thuộc [u, v], tăng vị trí i thêm c*i^k, với c và k là hằng số và 0 <= k <= 3.

Để thực hiện truy vấn tăng phần các phần tử i trong đoạn [u, v] lên **c** * i^k, ở mỗi nút quản lý đoạn [l, r] ta lưu thêm lazy[k], là tổng của các **c**. Ta sử dụng kĩ thuật lazy propagation để update.

Cài đặt:

Một nút của Segment tree như sau:

```
struct Node {
	int sum;
	int lazy[4];
} it[MAXN * 4];
```

Thao tác truy vấn:

```
int get(int i, int l, int r, int u, int v) {
    if (v < l || r < u) return 0;
```

```
    if (u <= l && r <= v) return it[i].sum;

    int mid = (l + r) >> 1;
    down(i, l, r, mid);  // lazy propagation: nếu it[i].lazy[k] > 0 thì cập nhật
it[i*2] và it[i*2+1]

    int res = get(i*2, l, mid, u, v) + get(i*2+1, mid+1, r, u, v);
    it[i].sum = (it[i*2].sum + it[i*2 + 1].sum) % MOD;
    return res;
}
```

Thao tác cập nhật:

```
void update(int i, int l, int r, int u, int v, int sign) { // sign = 1 → add, sign =
-1 → subtract
    if (v < l || r < u) return ;
    if (u <= l && r <= v) {
        for (int t = 0; t < 4; t++) {
            int c = ....;   // c là hằng số, tính theo t và u, như đã phân tích ở
lời giải.
      int cummulativeSum = ...;  // tổng cộng dồn i^t, với i = l..r. Khởi tạo trước
để tính trong O(1).

            it[i].sum += c * sign * cummulativeSum;
            it[i].lazy[t] += c * sign;
        }
        return ;
    }

    int mid = (l + r) / 2;
    down(i, l, r, mid);   // lazy propagation
    update(i * 2, l, mid, u, v, sign);
    update(i * 2 + 1, mid+1, r, u, v, sign);
    it[i].sum = (it[i*2].sum + it[i*2 + 1].sum) % MOD;
}
```

Lời giải 2:

Với mỗi thao tác cập nhật, ta không cập nhật trực tiếp dãy số mà lưu thao tác cập nhật vào 1 mảng **buffer**.

Mỗi khi mảng buffer có sqrt(Q) phần tử, thì ta sẽ cập nhật lại toàn bộ mảng lại 1 lần. Đoạn này ta cũng dùng kĩ thuật giống như cách làm trên:

$(i - u + 1) * (i - u + 2) * (i - u + 3)$
$= i^3 * 1$
$+ i^2 * (-3*u + 6)$
$+ i^1 * (3*u^2 - 12*u + 11)$
$+ i^0 * (-u^3 + 6*u^2 - 11*u + 6)$

Ta dùng mảng cộng dồn để cập nhật.

Như vậy:

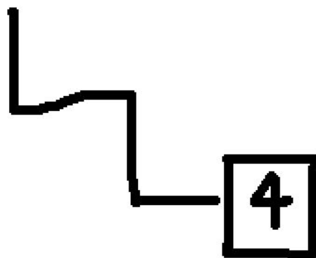
- Mỗi thao tác cập nhật được thực hiện trong O(1).
- Khi **buffer** đầy (kích thước bằng sqrt(Q)), ta cập nhật toàn bộ dãy trong O(N). Số lần cần cập nhật không quá O(sqrt(Q)) nên tổng độ phức tạp của phần này là O(sqrt(Q) * N).
- Với mỗi thao tác truy vấn:
  - Với mỗi thao tác cập nhật trong **buffer**, ta tính phần giao của truy vấn với thao tác cập nhật trong O(1).
  - Do kích thước **buffer** không quá O(sqrt(Q)), độ phức tạp thao tác truy vấn là O(sqrt(Q)).

B

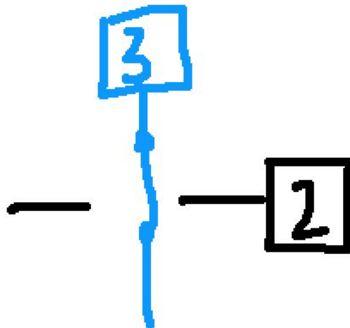
Độ khó: 3*
Người ra đề: Nguyễn Thành Trung (RR)

Nhận xét:
Bài này mình ra đề từ trò
https://play.google.com/store/apps/details?id=com.mindmill.pipes.loops.plumber.puzzle&hl=en.
Trong lúc thi có vài đội cũng thử cài bài này bằng những thuật toán luồng nhìn rất kinh nhưng không thể AC được. Theo BTC không thuật luồng nào có thể giải được bài này bởi sẽ không giải quyết triệt để được ít nhất 1 trong số những vấn đề sau:

- Không giải quyết được cạnh đi rẽ lung tung



- Không giải quyết được cạnh nhảy cóc



Thuật toán chuẩn của bài này là duyệt có độ ưu tiên.

Lời giải

Trước tiên, ta xét thuật toán duyệt hồn nhiên như sau:

```
void attempt(int i, int j) { // Duyệt ô (i, j)
      if (i > m) {
            // Ta đã duyệt qua tất cả các ô của bảng.
```

```
        // In kết quả tìm được
        return ;
    }

    // (ii, jj) = ô tiếp theo (xét thứ tự hàng cột).
    int ii = i, jj = j + 1;
    if (jj > n) {
        jj = 1;
        ii++;
    }

    // Nếu ô (i, j) không phải ô trống
    if (a[i][j] != '.') {
        // Duyệt đến ô tiếp theo
        attempt(ii, jj);
    } else {
        // Xét 4 hướng xung quanh.
        for (int dir = 0; dir < 4; dir++) {
            // (u, v) = ô số đầu tiên theo hướng dir.
            // Nếu không tìm được (u, v) hoặc khoảng cách từ (u, v) đến (i,
        j) lớn hơn giá trị của ô (u, v), bỏ qua.

            // Đánh dấu các ô từ (u, v) đến (i, j)
            // Giảm ô (u, v) đi |u - i| + |v - j|

            attempt(ii, jj);

            // Trả lại giá trị ban đầu của các ô (i, j) đến (u, v).
        }
    }
}
```

Dĩ nhiên, thuật toán này sẽ chạy quá lâu.

Điểm mấu chốt của bài này là ta cần duyệt các ô (i, j) theo thứ tự tốt hơn. Cụ thể ta sẽ chọn ô (i, j) có ít khả năng (ít ô (u, v) thoả mãn nhất để duyệt trước.

Sau đây là cài đặt: Những phần thay đổi được bôi đậm.

void attempt() {
    if (**bảng được điền kín**) {
        // Ta đã duyệt qua tất cả các ô của bảng.
        // In kết quả tìm được
        return ;
    }

```
    int i = -1, j = -1, best = 1000;
    for (int ii = 0; ii < m; ii++) {
        for (int jj = 0; jj < n; jj++) {
            c = số lượng ô số (u, v) mà có thể vẽ đường từ (u, v) đến (ii, jj)
            if (c < best) {
                best = c;
                i = ii;
                j = jj;
            }
        }
    }

    // Xét 4 hướng xung quanh.
    for (int dir = 0; dir < 4; dir++) {
        // (u, v) = ô số đầu tiên theo hướng dir.
        // Nếu không tìm được (u, v) hoặc khoảng cách từ (u, v) đến (i, j) lớn hơn giá trị
        của ô (u, v), bỏ qua.

        // Đánh dấu các ô từ (u, v) đến (i, j)
        // Giảm ô (u, v) đi |u - i| + |v - j|

        attempt();

        // Trả lại giá trị ban đầu của các ô (i, j) đến (u, v).
    }
}
```

Lời giải của Phạm Văn Hạnh

Điểm mấu chốt của bài toán này là nhận ra rằng, trong rất nhiều *trạng thái* của bảng có rất nhiều những nước đi là bắt buộc và có thể suy luận được. Những nước đi này thuộc một trong hai trường hợp sau:

1. Một ô trống hiện chưa bị phủ, nhưng chỉ có duy nhất 1 ô số có thể phủ nó. Ví dụ:

```
..5...
3>>>..
2.?.#4
......
......
..2...
```

Trong trường hợp trên, ô đánh dấu ? (là một ô trống chưa được phủ). chỉ có thể bị phủ bởi số 2 ở bên trái. Số 5 phía trên ko thể phủ dấu ? vì vướng đường phủ của số 3 ở hàng hai.

Số 2 ở phía dưới ko đủ tầm để vươn tới ô dấu ?, số 4 ở bên phải thì vướng qua ô cấm. Do đó số 2 bên trái sẽ phủ ô dấu ? và ta **chắc chắn** phủ thêm được hai ô (ô dấu ? và ô bên trái nó).

2. Một ô số cần phủ quá nhiều ô khác nhưng bị "giới hạn" khả năng phủ. Ví dụ:

```
.4.....
.v.^...
.v.^...
.v.7..#
.v.....
.......
...3...
```

Trên bảng, ô số 7 in đậm mới được phủ 2 ô ở phía trên, cần phủ thêm 5 ô nữa. Về hướng trái, số 7 có thể phủ thêm tối đa 1 ô, phía trên 1 ô, hướng phải 2 ô và phía dưới 2 ô. Do đó cần bị phủ ít nhất 1 ô về hướng phải và 1 ô về hướng phía dưới. Ta lại phủ thêm được 2 ô.

Hai suy luận bên trên là điểm quan trọng nhất của bài toán này. Ngay cả trong trạng thái ban đầu của bảng, bạn hoàn toàn có thể suy luận để điền rất nhiều ô một cách chắc chắn.

Đến khi không suy luận được thêm gì, ta bắt đầu phương pháp duyệt. Khi duyệt, bản chất là ta thử hết mọi trường hợp để phủ các ô. Để giảm số trạng thái phải duyệt, ở một trạng thái, ta tìm ra ô còn trống mà khoảng cách đến **số gần nhất phủ được nó** là **lớn nhất**. Khi thử phủ một ô theo phương pháp duyệt, chú ý thực hiện các bước suy luận như trên để điền thêm được nhiều ô mới. Tất nhiên những bước suy luận này sẽ không chắc chắn vì khi ở bước quay lui bạn thử phương án khác thì những phép suy luận trên sẽ bị xóa đi.

Ý tưởng của việc duyệt ô còn trống mà khoảng cách đến **số gần nhất phủ được nó** là **lớn nhất** là để, khi bạn duyệt một phương án khi quay lui, phương án đó sẽ điền thêm nhiều ô, làm cho phương pháp suy luận tham lam ở trên hiệu quả, bạn sẽ dễ dàng đi tới một trạng thái bảng được phủ toàn bộ, hoặc là một trạng thái bảng mà từ đó **không thể** phủ hết các ô còn lại, để bạn thử một nhánh quay lui khác.

C

Độ khó: 4*
Người ra đề: Lăng Trung Hiếu

Lời giải của Nguyễn Đinh Quang Minh

Lưu ý: bài chữa chỉ quan tâm đến thuật toán tìm A_N; việc tính S_N trong quá trình tính A_N khá đơn giản và sẽ được để lại như một bài tập để bạn đọc tự giải quyết.

Ý tưởng ban đầu: Thuật toán O(sqrt(N) * log(N))

Ta gọi tổng các chữ số của X là sod(X) và phép biến đổi từ A_n thành $A_{n+1} = A_n + sod(A_n)$ là một *bước nhảy.* Giả sử ta đang có số $A_{1000000} = 31054319$. Làm thế nào để biết sau bao nhiêu bước nhảy, số A sẽ vượt qua mốc 32000000?
Ta nhận thấy khi A vẫn còn có dạng $31*10^6 + R_n$, thì phần R_n "nhảy" theo công thức: $R_{n+1} = R_n + sod(R_n) + (3 + 1)$. 3 + 1 ở đây chính là tổng các chữ số của 31.
Cụ thể hơn, xét $A_n = Q * 10^k + R_n$ ($0 <= R_n < 10^k$), ta sẽ có công thức: $R_{n+1} = R_n + sod(R_n) + sod(Q)$.
Như vậy, dãy R_n thay đổi như thế nào chỉ phụ thuộc vào sod(Q), tức là tổng các chữ số của Q.
Nếu ta có thể tính được:

+ F(S, R) là số bước nhảy để R vượt qua 10^k khi sod(Q) = S, và
+ Next(S, R) là số R sau khi vượt qua 10^k.

thì sẽ tìm được kết quả bài toán bằng cách:

+ Khởi tạo các giá trị id = 1, A_id = 1, Q = 0.
+ Nếu id vẫn nhỏ hơn N thì "nhảy" A_id lên Q * 10^k + Next(Q, A_id % 10^k), cập nhật giá trị id theo F(Q, A_id % 10^k), và tăng Q lên 1.
+ Nếu "nhảy" A_id lên Next(Q, A_id % 10^k) mà id lớn hơn N thì ta dừng việc nhảy lại và biến đổi từng bước một đến khi id = N.

Nếu chọn k = 7 thì sod(Q) <= 9 * 10 = 90, ta có 90 * 10^7 trạng thái của F(S, R). Số bước "nhảy" theo thuật toán trên sẽ khoảng $10^{15} / 10^7 = 10^8$. Như vậy ta có một thuật toán khá tốt, mặc dù chưa thể chạy trong Time Limit nhưng nó là tiền đề để nghĩ ra ý tưởng tiếp theo.

Thuật toán kì diệu

Nhìn vào công thức và thuật toán ở trên, ta có một vài nhận xét sau:

+ Dãy A tăng chậm do $sod(A_i) < 160$ (A_N có không nhiều hơn 17 chữ số).
+ Thay vì chọn một giá trị k, ta có thể chọn nhiều giá trị k để tăng tốc việc "nhảy".

Cụ thể, ta sẽ thêm tham số k vào hàm QHĐ: vẫn với giả sử A = Q * 10^k + R, F(S, k, R) là số bước nhảy để R vượt qua 10^k khi sod(Q) = S, Next(S, k, R) là số R đầu tiên sau khi nhảy qua mốc 10^k.

Để nhảy R lên vượt qua 10^k, ta có thể nhảy R vượt qua 10^{k-1}, rồi 2*10^{k-1}, …, 9*10^{k-1}, 10*10^{k-1}.

Từ đó ta có thể tính F(S, k, R) như sau:

```
long long ret = 0;                    //trả về F(S, k, R)
int cur_digit = R / 10^(k-1);         //chữ số hàng 10^(k-1) của R.
long long cur_R = R;                  //R hiện tại
while (cur_R < 10^k) {
        ret += F(S + cur_digit, k - 1, cur_R % 10^(k-1));
        cur_R = Next(S + cur_digit, k - 1, cur_R % 10^(k-1)) +  cur_digit * 10^(k-1);
        cur_digit = cur_digit + 1;
}
return ret;
```

Dễ dàng nhận thấy k < 17, S < 160. Câu hỏi đặt ra là: giới hạn của R là bao nhiêu? Vì 2 phần tử liên tiếp của dãy A chỉ hơn nhau ít hơn 160 đơn vị nên khi ta “nhảy” R lên giá trị tiếp theo, R mới sẽ nhỏ hơn 160. Do đó chỉ cần lưu lại giá trị các hàm F và Next đã thăm, chúng ta sẽ giảm độ phức tạp thuật toán về mức chấp nhận được, thậm chí có thể chạy cả 1000 test mà không cần chuẩn bị trước. :)

D

Độ khó: 0*
Người ra đề: Nguyễn Thành Trung (RR)
Thể loại: Tham lam

Để có thể mua được nhiều cốc trà sữa nhất, ta cần mua loại có giá rẻ nhất.

Gọi giá của cốc trà sữa rẻ nhất là c, kết quả bài toán là max(0, X / c - 1).

Note:

- Ban đầu lúc ra đề mình chỉ định để kết quả là X / c. Tuy nhiên sau đấy để tăng độ khó, đề đã được sửa lại để output max(0, X / c - 1). Rất nhiều đội đã bị nộp sai nhiều lần do không lấy max với 0. (sẽ in ra -1).
- Ban đầu mình còn định để đề có nhiều thứ troll hơn nữa, nhưng sợ sang năm không có gì để troll nên mình xin để cái troll đó cho sang năm.

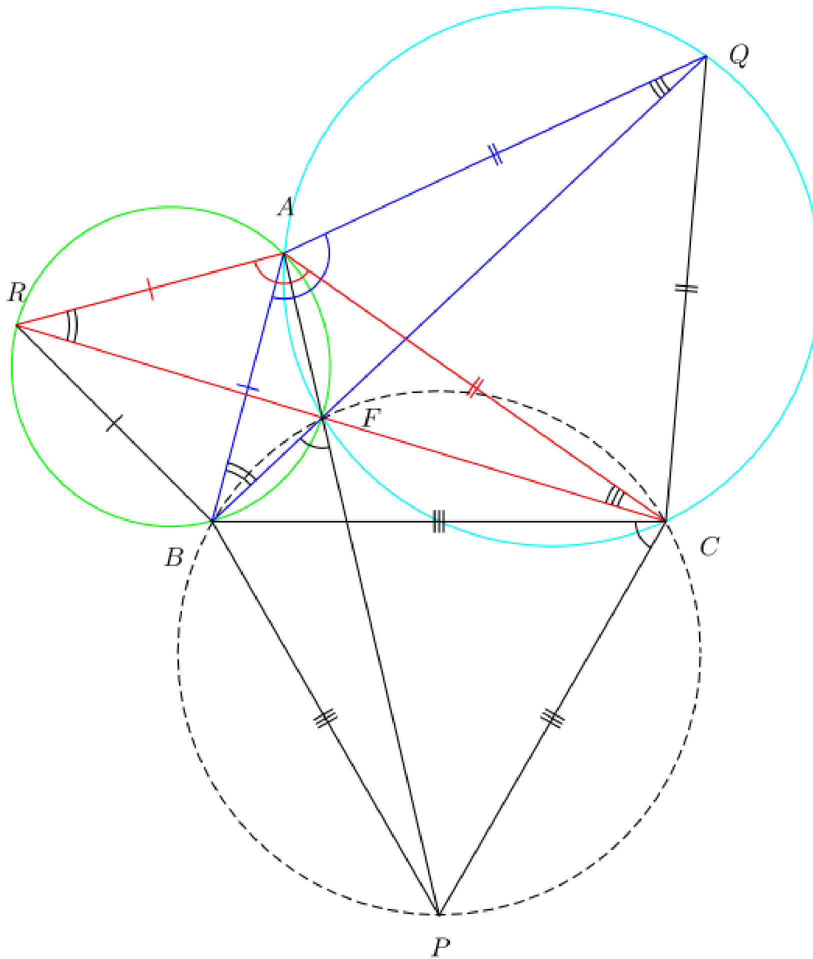
E

Độ khó: 3*
Người ra đề: Lăng Trung Hiếu

Cách 1

Điểm cần tìm chính là Fermat Point. Các bạn có thể đọc cách tìm và chứng minh ở Wikipedia.

Nếu tam giác ABC có 1 góc lớn hơn hoặc bằng 120 độ, giả sử là góc ở đỉnh A → kết quả là A. Ngược lại, vẽ 2 tam giác đều ABR và ACQ như hình vẽ. Kết quả là điểm F, là giao của RC và BQ.



Cách 2

[Ternary Search](#)

Nhận xét:

- Hàm f(x, y) = PA + PB + PC với P = (x, y) là hàm lồi.
- Điểm P có PA + PB + PC nhỏ nhất luôn luôn nằm trong tam giác ABC. Do đó khi cài đặt ta không cần quan tâm đến điều kiện đề bài là P phải nằm trong tam giác ABC, mà chỉ cần tìm điểm P bất kỳ làm tổng PA + PB + PC nhỏ nhất.

Từ 2 nhận xét trên, ta có thể tìm kiếm tam phân 2 chiều x và y để tìm kết quả.

Cài đặt:

```
// Trả lại tổng khoảng cách PA + PB + PC với P = (x, y)
double f(double x, double y) {
	...
}

// tìm kiếm tam phân theo trục y, khi đã cố định x
// return: giá trị nhỏ nhất của PA + PB + PC
double ternary_search_y(double x) {
	double left = -INF, right = INF;
	for (int turn = 0; turn < 100; turn++) {  // chặt 100 lần → độ chính xác là (⅔)^100
		double y1 = (left * 2 + right) / 3.0;
		double y2 = (left + right * 2) / 3.0;

		if (f(x, y1) < f(x, y2)) right = y2;
		else left = y1;
	}
}

// tìm kiếm tam phân theo trục x
// return: giá trị nhỏ nhất của PA + PB + PC
double ternary_search_x() {
	double left = -INF, right = INF;
	for (int turn = 0; turn < 100; turn++) {  // chặt 100 lần → độ chính xác là (⅔)^100
		double x1 = (left * 2 + right) / 3.0;
		double x2 = (left + right * 2) / 3.0;

		if (ternary_search_y(x1) < ternary_search_y(x2)) right = x2;
		else left = x1;
	}
}
```

Cách 3

[Local Search]

Trong bài này, vì chỉ có duy nhất 1 điểm cực trị (global optimum), và không có local optimum nào, nên Local Search đảm bảo ra được kết quả tối ưu.

Ở đây để đảm bảo kết quả đạt đủ độ chính xác sau K bước (converge sau K bước), mình cài đặt theo một biến thể như sau:

- Xuất phát từ điểm P bất kỳ
- Trong các điểm P' xung quanh P, tìm điểm P' mà tổng P'A + P'B + P'C nhỏ nhất, rồi chuyển P đến P'
    - Khi cài đặt, mình vẽ vòng tròn bán kính R có tâm là P, rồi chọn 100 điểm P' cách đều nhau trên vòng tròn.
- Ban đầu ta cho phép khoảng cách di chuyển PP' lớn (R = 2000), sau mỗi bước, giảm R đi 1% (R = R * 0.99), thì sau khoảng 10,000 bước, độ chính xác là 2000*0.99^10,000, đủ tốt để qua tất cả mọi test.

Cài đặt:

```
double len = 2000;  // độ dài PP'
int N_ITERATION = 10000;  // ta di chuyển P 10,000 lần.
double RATE = 0.99;  // giảm độ dài PP' sau mỗi iteration.

for (int turn = 0; turn < N_ITERATION; turn++) {  // độ chính xác = RATE^N_ITERATION

      Point best = P;  // điểm P' tối ưu.
      double bestDist = độ dài PA + PB + PC;

      for (double angle = 0; angle < 2 * PI; angle += (2 * PI) / 100) {
            Point dir = Point(0, len).rotate(angle);  // vector PP', xét 100 vector
với góc khác nhau.

            Point Q = P + dir;  // Q = P'

            if (QA + QB + QC < bestDist) {
                  best = Q;
                  bestDist = QA + QB + QC;
            }
      }
      P = best;
}
```

F

Độ khó: 3*
Người ra đề: Lăng Trung Hiếu
Thể loại: Quy hoạch động

Lời giải:

Bài toán gồm các bước chính:

Bước 1

Tính số lượng số trong khoảng [A, B] mà gồm các chữ số nằm đúng trong tập S (S là tập con của tập {0, 1, 2, ..., 9})

- Số số nằm trong khoảng [A, B] = (số số nằm trong [0, B]) - (số số trong [0, A]). Do đó khi quy hoạch động ta chỉ cần quan tâm đến cận trên của các số.
- Giống với các bài toán quy hoạch động chữ số, xuất phát từ số 0, ta lần lượt thêm các chữ số vào, và tính f(len, mask, lower, positive) với:
    - **len** là độ dài của số ta đang xây dựng.
    - **mask** là tập hợp các chữ số của số ta đang xây dựng.
    - **lower** = 1 nếu số ta đang xây dựng đã nhỏ hơn cận trên B, = 0 trong trường hợp ngược lại.
    - **positive** = 1 nếu số ta đang xây dựng đã lớn hơn 0.

Cài đặt:
```
f[0][0][0][0] = 1;  // Xuất phát từ số 0
for (int len = 0; len < độ dài số B; len++) {
      for (int mask = 0; mask < 1023; mask++) { // dùng bitmask lưu S.
            for (int lower = 0; lower < 1; lower++) {
                  for (int positive = 0; positive = 1; positive++) {
                        // Thêm 1 chữ số
                        for (int new_digit = 0; new_digit < 10; new_digit++) {
                              // Đảm bảo <= cận trên
                              if (lower == 0 && new_digit > chữ số (len+1) của B)
                                    continue;

                              // Tính mask2, lower2, positive2 là các giá trị của
                  số
                              // mới sau khi thêm chữ số new_digit
                              int positive2 = positive || (new_digit > 0);

                              int lower2 = lower
                                    || (new_digit < chữ số (len+1) của B).
```

```
                    int mask2 = mask;
                    if (positive2) mask2 |= 1<<new_digit;

                    f[len+1][mask2][lower2][positive2] +=
                        f[len][mask][lower][positive];
                }
            }
        }
    }
}
```

Bước 2

Với mỗi tập S, tính xem có bao nhiêu số trong [A, B] có S là tập con của tập các chữ số của nó. Bạn có thể giải phần này trong O(3^10) bằng cách duyệt mọi tập con.

Bước 3

Dùng tổ hợp để đếm số bộ k.

G

Độ khó: 2*
Người ra đề: Phạm Văn Hạnh
Thể loại: Cây khung nhỏ nhất + BFS

Lời giải

Gọi $\mathbf{L_{min}}$ là giá trị nhỏ nhất của trọng số lớn nhất trên một chu trinh đơn.

Nhận thấy chu trình cần tìm có cạnh trọng số lớn nhất là $\mathbf{L_{min}}$, bởi với mọi chu trình có cạnh trọng số lớn nhất là $L' > \mathbf{L_{min}}$ thì do $\mathbf{L_{min}} >= 1e6$ nên $L'^2 - \mathbf{L_{min}}^2 > 2e6 >= S * M$.

Để tìm giá trị $\mathbf{L_{min}}$, ta dùng Disjoint set giống như trong thuật toán Kruskal tìm cây khung nhỏ nhất:

Hơn nữa vì các cạnh có trọng số phân biệt nên chỉ có một cạnh có trọng số là $\mathbf{L_{min}}$. Xóa cạnh này khỏi đồ thị, dùng DFS hoặc BFS tìm số cạnh nhỏ nhất trên đường đi giữa hai đầu mút của cạnh này.

H

Độ khó: 4*
Người ra đề: Lê Yên Thanh, Lăng Trung Hiếu

Lời giải của Nguyễn Đinh Quang Minh

Trước hết, ta giải bài toán mà không có điều kiện phải giữ lại ít nhất 2 đống sỏi, tức là đếm số bộ (X1, X2, …, XN) sao cho Xi <= Ai và X1 ^ X2 ^ … ^ XN == 0.

Giả sử bit lớn nhất của các số trong mảng A là bit thứ **b**, và có **M** số trong mảng A (đánh số từ A_1 đến A_M có bit **b** bật. Giả sử ta chọn **K** số trong **M** số này để giữ nguyên bit **b** bật, **M-K** số còn lại bit **b** bị tắt (**K** phải chẵn để đảm bảo tổng xor = 0). Khi đó xảy ra 2 trường hợp:

1. K < M hay có ít nhất 1 số Ai có bit b bật nhưng bit b của Xi bị tắt. Khi đó phần còn lại của Xi có thể chọn tùy ý từ $0..2^b-1$. Khi đó bất kể ta điền các bit còn lại của các số còn lại như thế nào thì cũng luôn có duy nhất một cách chọn Xi. Như vậy số cách điền sẽ là: $P * Q * (2^b)^{M-K-1}$, trong đó:
    - P là tích các (Ai+1), với những số Ai có bit b tắt.
    - Q là tích các $(Ai - 2^b + 1)$, với những số Ai có bit b bật và Xi cũng có bit b bật.
2. K = M. Trường hợp này ta chỉ cần bỏ bit thứ b của tất cả các số và giải tiếp với bit thứ (b-1).

Trong trường hợp (i), P là hằng số nên ta có thể bỏ ra ngoài. Ta cần tính:

$$T_even = ½^b \ \Sigma\ (A_i1 - 2^b + 1)*(A_i2 - 2^b + 1)*...*(A_iK - 2^b + 1) * (2^b)^{M-K}$$

Với mọi bộ số i1 < i2 < … < iK và K chẵn.

Nếu ta định nghĩa T_odd tương tự như trên nhưng với điều kiện K lẻ thì sẽ có:

$$\begin{aligned} T_even + T_odd &= ½^b \ \Sigma\ (A_i1 - 2^b + 1)*(A_i2 - 2^b + 1)*...*(A_iK - 2^b + 1) * (2^b)^{M-K} \\ &= ½^b\ (A_1 + 1)*(A_2 + 1)*...*(A_M + 1) \end{aligned}$$

Điều trên có được dựa vào đẳng thức:

$$(x + A_1) * (x + a_2) * ... * (x + a_N) = \Sigma\ a_i1 * a_i2 * … * a_iK * x^{N-K},$$

Với $x = 2^b$ và $a_i = A_i - 2^b + 1$.

Tương tự, ta có:

$$\begin{aligned} T_even - T_odd &= ½^b \ \Sigma\ (A_i1 - 2^b + 1)*(A_i2 - 2^b + 1)*...*(A_iK - 2^b + 1) * \mathbf{(-1)^K} * (2^b)^{M-K} \\ &= ½^b\ (2^{b+1} - A_1 - 1)*(2^{b+1} - A_2 - 1)*...*(2^{b+1} - A_M - 1) \end{aligned}$$

Như vậy ta có thể tính T_{even} bằng tổng hai biểu thức trên chia đôi.
Lưu ý, tổng số cách tìm được ở trường hợp (i) phải loại trường hợp $K = M$ (nếu có).

Tổng độ phức tạp của thuật toán trên là $O(N * \log(\max(A_i)))$.

Với ràng buộc phải giữ lại ít nhất 2 đống sỏi, ta có thể làm như sau:

- Giải bài toán với mảng $(A_1, A_2, \ldots, A_N)$.
- Giải bài toán với mảng $(A_1 - 1, A_2 - 1, \ldots, A_N - 1)$ (để loại đi các cách không giữ lại đống sỏi nào).
- Giải bài toán với mảng $(A_1 - 1, A_2 - 1, \ldots, A_i, \ldots, A_N - 1)$ (để loại đi các cách chỉ giữ lại đúng một đống sỏi).

I

Độ khó: 1*
Người ra đề: PTIT

Lời giải:

Xét lần lượt tất cả các số trong khoảng [A, B].

Với mỗi số i, xét tất cả các cách rotate nó (có tối đa 7 cách rotate). Giả sử sau khi rotate thu được số j, nếu i < j, và cặp (i, j) chưa xuất hiện trước đó thì ta tăng kết quả thêm 1.

Chú ý cặp (i, j) có thể lặp lại trong trường hợp i = 121212.

Cài đặt:

```
int cnt = 0;  // kết quả
memset(mark, 0, sizeof mark);  // mảng đánh dấu để xem cặp (i, j) đã xuất hiện chưa

for (int i = a; i <= b; i++) {
      int j = i;

      // Lưu 10^k để tính rotate nhanh hơn.
      int p10 = 10;
      while (p10 <= u) p10 *= 10;
      p10 /= 10;

      for (int turn = 0; turn < 7; turn++) {
            j = (j % 10) * p10 + (j / 10);

            if (j > i && j <= b) {
                  if (mark[j] == i) break;
                  mark[j] = i;
                  ++cnt;
            }
      }
}
```

Lời giải của Phạm Văn Hạnh:

Gọi $f(x)$ là **số tự nhiên nhỏ nhất** tạo được bằng cách di chuyển một số chữ số ở cuối của x lên đầu (không kể các số bắt đầu bằng chữ số 0). Ví dụ $f(2715) = 1527$ $f(39359) = 35939$ $f(2001) = 1200$. Ta dễ dàng thấy rằng x và y là một cặp *rotated number* khi và chỉ khi $f(x) = f(y)$. Từ đây ta có ý tưởng sau:

Đặt $S(k) = \{x|1 \le x \le 10^6, f(x) = k\}$. Trước khi trả lời các truy vấn, ta duyệt qua các số từ 1 tới 10^6, tính $f(x)$ và lưu lại $S(k)$ với các số được sắp xếp theo thứ tự tăng dần. Để trả lời một truy vấn (A, B), ta duyệt qua mọi số k từ 1 tới 10^6, với mỗi $S(k)$ đếm số lượng số nằm trong đoạn [A..B] bằng phương pháp tìm kiếm nhị phân. Giả sử đếm được x số, ta cộng đáp số với $\frac{x(x-1)}{2}$.

Lưu ý rằng ta chỉ quan tâm tới các tập $S(k)$ có ít nhất hai số. chỉ có **189681** tập hợp thỏa mãn điều kiện này. Do đó thuật toán này sẽ phát huy hiệu quả rõ rệt so với thuật toán nêu trên khi các truy vấn có khoảng cách giữa A và B lớn.

J

Độ khó: 1*
Người ra đề: Lê Đôn Khuê
Thể loại: tìm kiếm tam phân, Toán sơ cấp

Lời giải 1 (phù hợp với sinh viên đại học)

Khi ta biết h, ta dễ dàng tính được thể tích của hộp:

f(h) = h * (X - 2*h) * (Y - 2*h)

Dễ thấy f không phải hàm lồi nhưng đồ thị hàm số f có dạng lồi trong đoạn từ 0 cho đến min(X/2, Y/2), vì vậy ta có thể dùng tìm kiếm tam phân để tìm cực đại của f(h).

Lời giải 2 (phù hợp với học sinh trung học phổ thông)

f(h) = h * (X - 2*h) * (Y - 2*h)
    = - 4 * h^3 - 2*(X + Y) * h^2 + X*Y * h

Đây là một đa thức bậc 3. Để tìm cực trị, ta tính đạo hàm của f và tìm những điểm mà f' = 0.

Lời giải 3 (phù hợp với học sinh trung học cơ sở)

Nhìn vào công thức của f(h), ta nghĩ tới sử dụng bất đẳng thức Cauchy cho ba số không âm để tìm max f(h): $abc \leq \frac{(a+b+c)^3}{27}$.
Ta sẽ chọn hai hằng số α, β dương và sử dụng bđt Cauchy:
$\alpha\beta f(h) = h\alpha(X - 2h)\beta(Y - 2h) \leq \frac{(h+\alpha(X-2h)+\beta(Y-2h))^3}{27}$
Để vế phải là hằng số thì $1 - 2(\alpha + \beta) = 0$
Để dấu bằng có thể xảy ra thì $h = \alpha(X - 2h) = \beta(Y - 2h) \leftrightarrow \frac{\alpha X}{2\alpha+1} = \frac{\beta Y}{2\beta+1}$
Giải hệ phương trình trên ta tìm được α, β và h.

K

Độ khó: 2*
Người ra đề: Dựa trên bài đóng góp bởi HCMUP, có chỉnh sửa bời Lê Đôn Khuê
Thể loại: Hình học, bao lồi

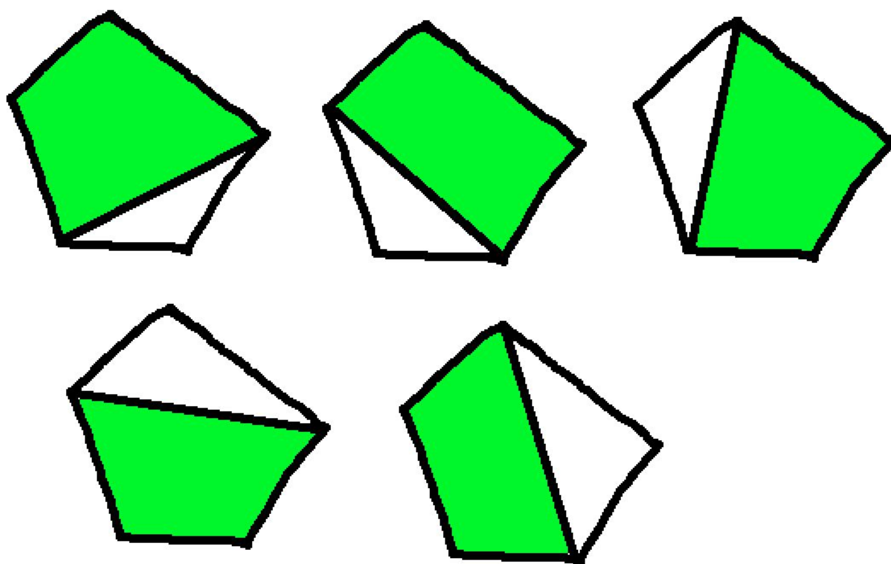
Phát biểu lại bài toán

- Cho A điểm xanh
- Cho B điểm đỏ.
- Đếm **số lượng điểm đỏ** nằm trong **ít nhất** 1 tứ giác có 4 đỉnh màu xanh (không tự cắt, không có 3 đỉnh thẳng hàng)

Lời giải

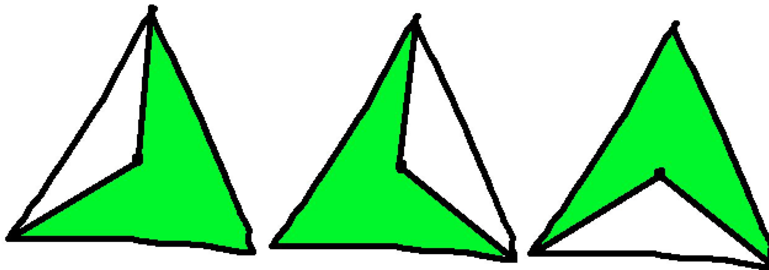
Tìm bao lồi của các điểm xanh.

- Nếu bao lồi có ít nhất 4 điểm, kết quả chính là số lượng điểm đỏ nằm trong bao lồi. Ví dụ khi bao lồi có 5 điểm, ta xét 5 tứ giác như thế này:

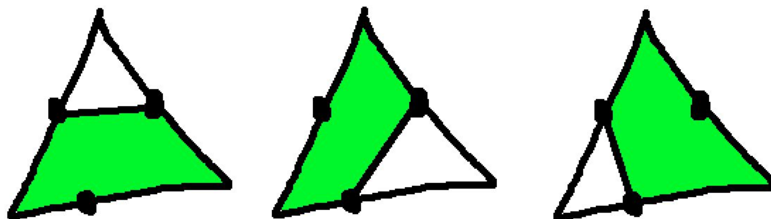


- Nếu bao lồi có 1 hoặc 2 điểm, kết quả = 0.

- Nếu bao lồi có **đúng 3 điểm**, và có **ít nhất 1 điểm** nằm hoàn toàn trong bao lồi: Kết quả vẫn chính là số lượng điểm nằm hoàn toàn trong bao lồi.



- Nếu bao lồi có **đúng 3 điểm**, không có điểm nào nằm hoàn toàn trong bao lồi:
    - Nếu **không có điểm nào** trên cạnh của bao lồi → kết quả = 0.
    - Nếu có điểm nằm trên **đúng 1 cạnh** của bao lồi → kết quả = 0.
    - Nếu có điểm nằm trên **đúng 2 cạnh** của bao lồi → giả sử bao lồi là tam giác ABC, có điểm màu xanh nằm trên AB, AC nhưng không có điểm xanh trên BC, giả sử có 1 một tứ giác đỉnh A lúc đấy sẽ phải tồn tại ít nhất 2 đỉnh khác thuộc AB hoặc AC => đó là một tứ giác suy biến không thoả mãn. Khi đó, xóa điểm A ra khói tập đỉnh màu xanh, tìm lại bao lồi tập đỉnh màu xanh (lúc này bao lồi không còn là tam giác), và ta quy về trường hợp trên.
    - Nếu có điểm nằm trên **cả 3 cạnh** của bao lồi → kết quả chính là số lượng điểm đỏ nằm trong bao lồi.



Nhận xét:

Tồn tại thuật toán kiểm tra một điểm thuộc đa giác lồi trong thời gian O(logN), do đó bài toán có thể giải với 10 vạn điểm màu đỏ và 10 vạn điểm màu xanh. Tuy nhiên, đề ra cho giới hạn một ngàn điểm mỗi màu nhằm mục đích bạn có thể sử dụng nhiều thuật toán khác để kiểm tra một điểm nằm trong đa giác.

Một bài toán cơ bản, từng xuất hiện trong nhiều kì thi ACM yêu cầu ta xét các **tam giác** thay vì **tứ giác** màu xanh. Không khó để nhận ra lời giải hai bài toán là không mấy khác biệt, tuy nhiên bạn cần sự tinh tế rất lớn để nhận ra những trường hợp cần phải xử lí.

Để tránh việc phải cài đặt nhiều và dễ gặp lỗi, trong bài này, bạn chỉ nên cài một hàm tìm bao lồi (nhận vào một vector các điểm và trả về một vector các điểm là bao lồi), sau đó sử dụng lại hàm này nhiều lần. Để kiểm tra một điểm A có nằm trong đa giác lồi P, bạn chỉ cần tìm

bao lồi của tập hợp điểm P , A nằm trong P khi và chỉ khi A không xuất hiện trong bao lồi vừa tìm được (lưu ý rằng mọi điểm là phân biệt, kể cả hai điểm khác màu).

L

Độ khó: 1*
Người ra đề: Phạm Văn Hạnh
Thể loại: Đọc kĩ đề bài, kiểm tra thị lực.

Lời giải

- Vì với mỗi cạnh (u, v) của đồ thị, u < v, nên đồ thị **không có chu trình.**
- Do đồ thị có hướng và (renting_cost) < (buying_cost) / PI → bạn không bao giờ "buy property".
- Do N < 10^6 <= K và đồ thị có hướng → 2 người chơi không bao giờ đi đủ K lượt → ta không cần quan tâm đến K.

Do đó, 2 người chơi không ảnh hưởng đến nhau (cách ảnh hưởng duy nhất là buy property nhưng chỉ đem lại lỗ :))).

Với mỗi người chơi, ta dùng quy hoạch động để tính điểm tối đa mà người chơi đó đạt được:

Cài đặt

```
const long long INF = -1e18;

// Trả lại điểm tối ưu nếu xuất phát từ u.
// Khởi tạo f[i] = -INF với mọi i.
// adj[i] = danh sách kề của đỉnh i.

long long get(int u) {
    if (f[u] > -INF) return f[u];  // Nếu f[u] đã được tính từ trước, trả lại luôn
f[u].

    if (adj[u].empty()) return f[u] = 0; // Đỉnh lá, f = 0.

    for (int v : adj[u]) {  // Xét tất cả v kề u.
        f[u] = max(f[u], val[v] + get(v));
    }
    return f[u];
}
```

Nhận xét

Đây là một bài dễ, tuy nhiên để làm được các bạn cần phải đọc đề cẩn thận. Các bạn có 3 người, mỗi người 5h → tổng là 15h. Đọc đề mỗi bài mất chưa đến 5 phút. Vì vậy các bạn bắt buộc phải đọc cẩn thận tất cả các chi tiết trong đề. Ở đây bọn mình đã đặt những chi tiết quan trọng nhất của đề vào phần Constraints - cũng là phần quan trọng nhất của đề bài. Một số đề ACM mình từng làm còn giấu những chi tiết quan trọng trong phần kể chuyện - phần mà ai cũng đọc lướt. Rất tiếc là vẫn không đội Việt Nam nào làm được. Còn ở online mirror, có khá nhiều đội nước ngoài làm được bài này:
https://open.kattis.com/contests/vietnam-national17-open/standings.