ABC097 / ARC097 解説

sigma425

For International Readers: English editorial starts on page ???.

A: Colorful Transceivers

問題文の通り実装します。二点 a,b 間の距離は標準的な言語なら絶対値関数で計算できるでしょう。自分で定義しても良いと思います。たとえば C++ なら abs($a-b$) と書けます。あとは、(A と C が直接会話可能) または ((A と B が直接会話可能) かつ (B と C が直接会話可能)) という論理構造を、論理演算子を使ってコードに落とし込めばよいです。以下は C++ のコード例です。

```
#include <bits/stdc++.h>
using namespace std;
int main(){
        int a,b,c,d;
        cin>>a>>b>>c>>d;
        if(abs(a-c)<=d || (abs(a-b)<=d && abs(b-c)<=d)) puts("Yes");
        else puts("No");
}
```

B: Exponential

各 1 以上 X 以下の整数 x に対して、$expo_x := x$ がべき乗数か という真偽値を考えます。これが正しく計算できれば答えは求まります。

1 以上の整数 b と、 2 以上の整数 p 全てのペアに対して $x := b^p$ を計算し、x が 1 以上 X 以下なら $expo_x$ を true に書き換える ということをすると定義より正しく求まりますが、当然すべての整数のペアを試すのは不可能です。

まず、b として試すべき値は 1 以上 X 以下の整数 のみでよいです (なぜなら、$b > X$ なら任意の $p >= 2$ に対して $b^p > X$ だから答えに影響を及ぼさない)。

同様に、 p として試すべき値は (とても雑に見積もっても) 2 以上 X 以下の整数 のみで十分です。よって二重ループで b, p の全てのペアを試すことで $expo_x$ が正しく求まります。 時間計算量は $O(X^2)$ です。

しかし、オーバーフローを起こさないためにも以下の方法が良いと思います。

各 b に対して、$p = 2, 3, ..$ と順番に, $expo_{b^p}$ を true に書き換えていきます。ここで、もし今調べている値 b^k が X より大きかったら、k 以降の値は p として考える必要がありません。従ってそこで p のループを打ち切って良いです。ただし $b = 1$ の時にはこの条件での打ち切りは発生しないので注意してください。

```
#include <bits/stdc++.h>
using namespace std;
```

```
int main(){
        int X;
        cin>>X;
        vector<bool> expo(X+1);

        expo[1] = 1;
        for(int b=2;b<=X;b++){
                int v = b*b;
                while(v<=X){
                        expo[v] = 1;
                        v *= b;
                }
        }
        for(int i=X;i>=1;i--) if(expo[i]){
                cout<<i<<endl;
                return 0;
        }
}
```

C: K-th Substring

解説では $N := |s|$ とおきます。

まず部分点解法 ($N < 50$) を説明します。全ての s の substring を列挙し、重複を除き、ソートします。これにはいろいろな方法がありますが、例えば C++ だと sort したあと unique 関数を使ったり、set を使うなどが考えられます。その後、小さい方から K 番目の文字列を出力すればよいです。

計算量は, $O(N^2)$ 個の 長さ $O(N)$ の文字列をソートするので、 $O(N^3 logN)$ になります。(比較回数が $O(N^2 logN)$, 1 回の比較にかかる時間が $O(N)$)

満点解法では、答えの文字列の長さが高々 K であるということに着目します。(これは、文字列 t の prefix(であって t でないもの) が 全て異なり、かつ strict に t より小さいことから示せます。) この事実からはじめに列挙する必要があるのは 長さ K 以下の substring $O(NK)$ 個だけです。これをソートすると、 時間計算量は $O(NK^2(logNK))$ となり、十分高速です。

D: Equals

以下のような無向グラフ G を考えます。

頂点集合: 1 以上 N 以下の整数

辺集合: 各 $1 \leq j \leq M$ に対して、(x_j, y_j) という辺を張る

すると次のことが示せます:

$S := \{i | G$ において i と p_iが同じ連結成分にある $\}$ とおくと、 最終状態で全ての $i \in S$ に対して同時に $p_i = i$ とすることが出来る

証明は後に回します。逆に G において i と p_i が異なる連結成分に属してしまっている場合, 操作を行っても $p_i = i$ と出来ないことは明らかです。従って、$|S|$ が答えになります。各 i が S に 属するかは UnionFind 等を使うと求めることが出来ます。

証明: 気持ち的には同じ連結成分内では好きに値を入れ替えられることからわかります。ちゃんとやると、各連結成分 C に対して、C から $\{p_i | i \in C\}$ への全単射 f であって $i \in S$ なら $f(i) = i$ となるようなものが

とれて、連結成分の全域木をとって、ひとつの葉 i に正しい値 $f(i)$ を置くということを繰り返せば示せます。

E: Sorted and Sorted

簡単のため、 i が書かれた 黒い/白い ボールを 黒/白 の i と呼びます。

最終状態を一つ固定した時に、その状態になるまでに必要な最小の操作回数 f は転倒数で表せます。すなわち、$f = \#\{(x, y) : \text{ボールの組 } | x \text{ は初期状態で } y \text{ より左にある} \wedge x \text{ は最終状態で } y \text{ より右にある}\}$

ボール x を固定した時に上の集合に入る (x, y) の個数を f_x と置きます。 $f = \sum f_x$ です。

次のような DP を考えます。$dp_{i,j}$ = 最終状態においてボールを左から順に置いていって、黒いボールを i 個, 白いボールを j 個 置いた時の, これまで置いたボール x に関する f_x の和 の最小値

これは次のように更新できます。$dp_{i,j} = \min(dp_{i-1,j} + costb_{i-1,j}, dp_{i,j-1} + costw_{i,j-1})$ ここで、$costb_{i,j}$ とは、 黒いボールを i 個, 白いボールを j 個既に置いている時に 黒の $i+1$ を置いた時の $f_{\text{黒の } i+1}$ の値です。これは well-defined です。(つまり、これからどう置くかや、これまでどう置いたかには依存していません) このことは x の左にあるボールの集合が i, j だけから一意に定まることからわかります。$costw$ も同様です。

$costb, costw$ は BIT を使って $O(N^2 logN)$ で求めたり, “左 k 個にある i 以下の黒いボールの個数” などを前計算したりすることで $O(N^2)$ で求めることが可能です。すると全ての $dp_{i,j}$ も $O(N^2)$ で求められます。答えは $dp_{N,N}$ です。

F: Monochrome Cat

まず黒い葉をわざわざ訪れる必要はありません。なので、黒い葉があればそれを消す、ということをできるだけ繰り返します (これは queue 等を使うと簡単にできます)。するといつかは全ての葉が白くなります。以下この木について考えます。

“現在いる頂点と隣接した頂点をひとつ選びその頂点に移動する。その後、移動先の頂点の色を反転する。” という操作で 頂点 a から 頂点 b に移動した場合、この操作を $a \Rightarrow b$ で表します。

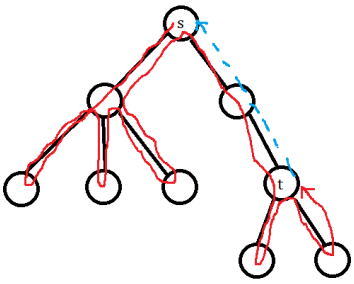
また、” 今いる頂点の色を反転する” という操作を 頂点 a で行った場合、この操作を flip a で表します。

まず、ひとつの辺を 3 回以上渡ることはありません。なぜなら、(操作列 1) $\rightarrow$ ($a \Rightarrow b$) $\rightarrow$ (操作列 2) $\rightarrow$ ($b \Rightarrow a$) $\rightarrow$ (操作列 3) $\rightarrow$ ($a \Rightarrow b$) $\rightarrow$ (操作列 4) という操作列は、(操作列 1) $\rightarrow$ (操作列 3) $\rightarrow$ (flip a) $\rightarrow$ ($a \Rightarrow b$) $\rightarrow$ (flip b) $\rightarrow$ (操作列 2) $\rightarrow$ (操作列 4) と変更すると、操作回数は保ったまま、他の辺の使用回数を変えずに、辺 (a, b) の使用回数を減らせるからです。

よって各辺は高々 2 回しか使わないとしてよいです。

また、全ての葉が白なので一度は訪れる必要があることから、すべての頂点を訪れる必要があることがわかります。特に各辺を少なくとも 1 回は使います。

この 2 つの考察から、猫の移動の様子は次の図の赤線のように限定されます:



すなわち、始頂点 s から Euler tour をする途中、すべての頂点を訪れたあとなら 終頂点 t で移動をやめて良い という形です。

移動操作の列が決まっている時、flip 操作 を何回行えばいいかは一意に定まります。なぜなら、各頂点を少なくとも一度は訪れているので、もし” すべての移動操作が終わったあとに頂点 v の色が白になっている” ならば、 頂点 v を訪れたタイミングで flip v をすることにすればよいからです。

ここで、Euler tour が最後まで完了するように、操作列に青の破線のように追加することを考えます。この時の操作回数は、$2(N-1)+\#\{v \in G |$ (v : white) xor ($\deg_v$ is odd) $\}$ となり、s,t や動き方には依存しません。

ここから青の追加分を消すことを考えます。操作回数がどのくらい減るかは、青の破線の path を $t = b_0, b_1, .., b_{K-1}, b_K = s$ とおくと、次の f で表せます。

$f := K + \#\{v \in \{b_1, b_2, ., b_K\} |$ (v : white) xor ($\deg_v$ is odd) $\}$ - $\#\{v \in \{b_1, b_2, ., b_K\} |$ (v : white) xor ($\deg_v$ is even) $\}$

変形して、

$f = 2 \times \#\{v \in \{b_1, b_2, ., b_K\} |$ (v : white) xor ($\deg_v$ is odd) $\}$

この f を最大化するのが目的です。これは t, s のみに依存することに注意してください。

この問題は次のように言い換えられます:

無向グラフが与えられる。各頂点には値 0 か 1 が書いてある。最大で何個 1 が書いてある頂点を含める path がとれるか?

($\because b_0, .., b_K$ は任意の path をとれます. path をわざわざ短くする必要はないので, b_0, b_K は葉としてよいです。**頂点数が 2 以上なら** 定義から葉, 特に b_0 に書いてある値は 0 なので、結局 (上の問題の答え) $\times 2$ が f になります。)

これは木の直径を求めるのと同様のアルゴリズムで dfs 2 回で可能です。 これで $O(N)$ で元の問題の答えが求まりました。

初期状態で白い頂点が 0 個,1 個のコーナーケースに注意してください。(1 個の場合は上の $\because$ 内の太字部分が成り立たないので)

ABC097 / ARC097 Editorial

sigma425

A: Colorful Transceivers

C++ example:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
        int a,b,c,d;
        cin>>a>>b>>c>>d;
        if(abs(a-c)<=d || (abs(a-b)<=d && abs(b-c)<=d)) puts("Yes");
        else puts("No");
}
```

B: Exponential

C++ example:

```
#include <bits/stdc++.h>
using namespace std;
int main(){
        int X;
        cin>>X;
        vector<bool> expo(X+1);

        expo[1] = 1;
        for(int b=2;b<=X;b++){
                int v = b*b;
                while(v<=X){
                        expo[v] = 1;
                        v *= b;
                }
        }
        for(int i=X;i>=1;i--) if(expo[i]){
                cout<<i<<endl;
                return 0;
        }
}
```

C: K-th Substring

For an arbitrary string t, each of its proper suffix is lexicographically smaller than t, and the lexicographic rank of t is at least $|t|$. Thus, the length of the answer is at most K.

Generate all substrings of s whose lengths are at most K. Sort them, unique them, and print the K-th one. This solution works in $O(NK^2(logNK))$ time, where $N = |S|$.

D: Equals

Consider a graph G: there are N vertices numbered 1 through N, and for each j there is an edge (x_j, y_j).

Then, by performing some operations, p_i can be i if they are in the same connected component of the graph. We can also prove that we can satisfy $p_i = i$ for all such i simultaneously. (Formal proof: take a spanning tree of a connected component. Choose its leaf v, perform opeartions such that $p_v = v$ is satisfied, and never make operations involving v after that.)

We can compute the number of such i by DFS or DSU.

E: Sorted and Sorted

Suppose that we know the final order of balls. Then, we can compute the number of operations as an inversion number: the number of ordered pairs of two balls (x, y) such that x is to the left of y at the beginning, but x is to the right of y in the final order.

Let's decide the positions of balls and place them one by one, from left to right (in the final order). From the constraints in the statement, at any moment, the set of placed balls must be of the form "i black balls numbered 1 through i, and j white balls numbered 1 through j". Define $dp_{i,j}$ as the minimum possible inversion number among them when we place those balls. The answer is $dp_{N,N}$.

Then, the recurrence formula will be as follows:

$dp_{i,j} = \min(dp_{i-1,j} + costb_{i-1,j}, dp_{i,j-1} + costw_{i,j-1})$

Here, $costb_{i,j}$ is the cost required to append black $i+1$ when the first i black balls and the first j white balls are already placed. That is, the number of balls that are already placed and initially placed to the right of black $i+1$. Similarly, $costw_{i,j}$ are defined.

We can pre-compute the arrays $costb_{i,j}$, $costw_{i,j}$ like prefix sums in $O(N^2)$. Thus, this solution works in $O(N^2)$ time.

F: Monochrome Cat

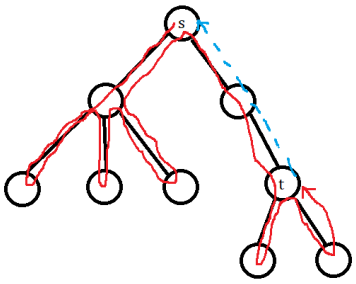
If the tree contains a black leaf, it never makes sense to visit it. Thus, we can repeatedly remove black leaves until all leaves become white. (This can be implemented by, for example, using a queue.) From now on, we assume that all leaves in the input tree are white. We also assume that the tree has at least two white vertices to avoid some special cases.

Suppose that we are given two vertices s and t, and the cat must start at s and end at t. How many operations do we need in this case?

First, for each edge e, let's compute the number of times the cat passes through e.

- It never makes sense to pass through it three or more times. For example, suppose that we move between two vertices a and b three times. The sequence of operations will look like "W $\to$ ($a \Rightarrow b$) $\to$ X $\to$ ($b \Rightarrow a$) $\to$ Y $\to$ ($a \Rightarrow b$) $\to$ Z" However, we can replace it with "W $\to$ Y $\to$ (flip a) $\to$ ($a \Rightarrow b$) $\to$ (flip b) $\to$ X $\to$ Z" and the number of operations will be smaller. Here, W, X, Y, Z represent some sequences of operations, $a \Rightarrow b$ denotes an operation of the first type, and flip a denotes an operation of the second type.
- We must pass through each edge at least once (otherwise some white leaves will be left unvisited.)

Thus, if e is on the path between s and t, we pass through it exactly once, otherwise we pass through it exactly twice. The trajectory of the cat will look as follows:



Now, we know the number of operations of the first type. How can we compute the number of operations of the second type?

Suppose that if we never make operations of the second type, k white vertices remains. Then, we need k extra steps to change the color of those vertices. This is always possible because we visit all vertices.

Let's call those k vertices "bad vertices", and other vertices "good vertices". Depending on the initial color and the parity of degree of the vertex, there are two types of vertices:

- It becomes a bad vertex if it is on the $s - t$ path, otherwise it becomes a good vertex.
- It becomes a good vertex if it is on the $s - t$ path, otherwise it becomes a bad vertex.

In summary, the total number of opeartions can be computed as follows (for a given pair of s, t):

- Add twice the total number of edges.

- Subtract the distance between s and t.
- Subtract the number of bad vertices.

This can be simplified as follows. For each vertex, the cost of zero or two is assigned depending on the type of the vertex mentioned above. Then,

- Add twice the total number of edges, plus one.
- Subtract the total cost of all vertices on the path between s and t.

Now our objective is to find a path that maximize the total cost of vertices on it. This is similar to the computation of diamater of a weighted tree, and can be done in $O(N)$.