ABC 138 解説

camypaper, evima, sheyasutaka, tozangezan, yosupo

2019 年 8 月 18 日

For International Readers: English editorial will be provided in a few days.

A: Red or Not

(原案: tozangezan, 準備・解説: evima)

practice contest の問題 A で足し算の代わりに条件分岐をさせるような問題です。入出力の行い方はそちらのサンプルコードを参照していただき、あとは if 文や条件演算子 (?: など) と比較演算子 (>= など) を組み合わせて a と 3200 の大小に応じて出力を変える処理を加えれば完成です。(抽象的な記述で終わってしまい恐縮ですが、個別の言語について述べるにはこの PDF は狭すぎます。Google 検索を活用してください)

以下は Python(3) での実装例です。

```
a = int(input())
s = input()
print(s if a >= 3200 else 'red')
```

B: Resistors in Parallel

(原案: sheyasutaka, 準備・解説: evima)

入出力例の説明では数式の分母 $\frac{1}{A_1} + \frac{1}{A_2} + \ldots + \frac{1}{A_N}$ を分数のまま計算していますが、はじめから浮動小数点型を使って計算するのが簡単でしょう。あとは、for 文やその他の何らかの繰り返し処理を行う機構を用いて実装を行います。諸言語を代表して C++ と Python(3) での実装例を掲載します。

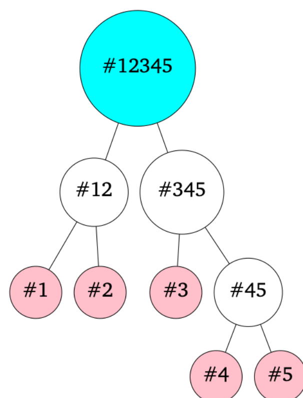
```
#include <iomanip>
#include <iostream>
using namespace std;

int main(){
    int N;
    double ans = 0.0, A;
    cin >> N;
    for(int i = 1; i <= N; ++i){
        cin >> A;
        ans += 1.0 / A;
    }
    ans = 1.0 / ans;
    cout << setprecision(16) << ans << endl;
}
```

```
N = int(input())
A = map(int, input().split())
print('{:.16g}'.format(1 / sum(1 / x for x in A)))
```

C: Alchemist

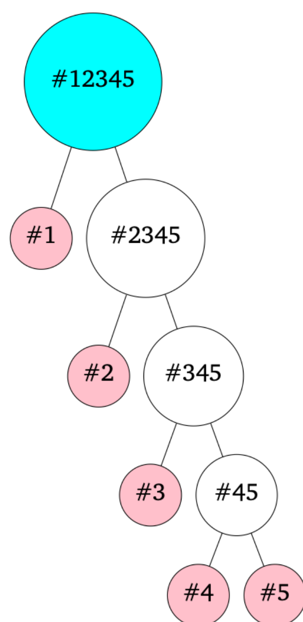
(原案: yosupo, 準備・解説: evima)

以下、具材 $1, \ldots, N$ は価値の大きい順に番号がつけられているとして (はじめに具材をソートしてそのように番号を付け替えます)、合成の過程を下図のように木構造として表すことを考えます (#1 は具材 1、#12 は具材 1, 2 を合成して得られる具材を表します)。



この図で具材 i を表す葉から最終的な具材を表す根までの距離 (間の辺の数) を d_i とすると、最終的な具材の価値は $v_1/2^{d_1} + \ldots + v_N/2^{d_N}$ と書けます。最大の結果を得るには、$d_1 \leq d_2 \leq \ldots \leq d_N$ であるような合成過程のみ考えれば十分です（そうでなければ、具材の番号を付け替えることで最終結果が向上するため)。

さらに考えると、最大の結果を得るには次の図のように $d_1 = 1, d_2 = 2, \ldots, d_{N-1} = d_N = N - 1$ とするべきです。これは、簡潔に述べると、$d_1 \leq d_2 \leq \ldots \leq d_N$ を満たすこれ以外の合成過程 (例えば上図のような) は、もっとも浅い「違反」した「左の子」以下を最も深い「右の子」以下に「移植」することで最終結果が向上するためです。



D: Ki

(原案: tozangezan, 準備・解説: evima)

一度の操作が N 個の頂点すべてに影響する可能性があるため、操作をそのままシミュレートするとカウンターへの加算を NQ 回行うことになり実行時間制限に間に合いません。[*1]

このような木に関する問題では、木の代わりに $1-2-3-\ldots-N$ という直線的なグラフを考えると助けになることがあり、今回はその好例です。この問題でグラフが直線的であるとすると、次のような線形時間の解法が考えられます。

1. $c_1, c_2, \ldots, c_N = 0, 0, \ldots, 0$ とする。
2. 各操作 j に対し、c_{p_j} に x_j を加算する。
3. $i = 2, \ldots, N$ の順に、c_i に c_{i-1} を加算する。
4. $c_1, c_2, \ldots, c_N$ を出力する。

そして、この解法は木に対してもほぼそのまま利用できます。具体的には、手順 3 の c_{i-1} を c_{q_i} (q_i は頂点 i の親) と読み換え~~ればそのまま元の問題への解法となります。~~、この処理を根に近い頂点から順に行えば元の問題への解法となります。(8/19 0:05 JST 修正) [*2] [*3]

[*1] N と Q がともに最大の 20 万の場合、$NQ = 400$ 億回。C++ などの高速な言語でも 2 秒間に処理できる加算の回数は数億回程度で、数百億回となると絶望的です

[*2] ~~ただし、このアルゴリズムは制約 $a_i < b_i$ を利用していることに注意してください。この制約がない場合は、~~各 i ($1 \leq i \leq N-1$) に対して頂点 a_i が b_i の親であるとは限らず、(8/19 0:05 JST 修正) 再帰関数を用いるなどして明示的に根から順に処理を行う必要があります

[*3] このあたりの問題から、言語によっては「正しい」解法でも実行時間制限に間に合うかきわどいことがあるかもしれません。競技プログラミングに真剣に取り組む場合、このあたりで言語の乗り換えを検討し始めることをお勧めします

E: Strings of Impurity

(原案: sheyasutaka, 準備・解説: evima)

一般に、文字列 t が文字列 u の部分列であるかは t を u と貪欲にマッチさせれば判定できます。つまり、t の最初の文字が u に現れる最初の位置を見つけ、その位置より後ろで t の次の文字が u に現れる位置を見つけ、... という要領で t の最後の文字まで u の中に見つけられるかを試せばよく、今回は $u = s'$ (s の無限回の繰り返し) としてこの処理を効率的に行うことが求められています。

s' は無限に長いですが、t の長さは限られている点がポイントで、上の $|t|$ 回の処理の一回一回を高速に行えれば十分です。そのためには、まず `a` から `z` までの各文字種 ch に対して s 内の ch の出現位置をすべて抽出するべきでしょう。あとは周期性を利用し、$i = 0, \ldots, |s| - 1$ のそれぞれに対して $\mathrm{next}_{\mathrm{ch,i}}$:「$s'$ の i 文字目以降で文字種 ch が最初に現れる位置」を O((文字種数) $\times$ |s|) 時間かけて事前にすべて求めておくという方針や、二分探索を用いて ch の次の出現位置を毎回 $\mathrm{O}(\log |s|)$ 時間かけて求めるという方針が考えられます。いずれにせよ、s を 2 個連結した文字列を用いると実装が楽になります。

F: Coincidence

(原案: camypaper, 準備・解説: evima)

y の MSB (Most Significant Bit: 最上位ビット) の位置を m とします (つまり、y の 2^m の位の桁は 1 で、任意の整数 n $(n > m)$ に対して x の 2^n の位の桁は 0 です)。ここで x の 2^m の位の桁が 0 であると仮定すると、XOR の定義より x XOR y の 2^m の位の桁は 1 となります。しかし、一般に整数 x, y $(y \geq x)$ に対して y を x で割った余りは y の半分未満であり[*4]、これは y を x で割った余りの MSB の位置が m 未満であることを意味するため、y を x で割った余りは x XOR y と決して一致しません。よって、x の MSB の位置も m である場合のみ考えれば十分です。このとき y を x を割った商は 1 であるため、y を x で割った余りは単に $y - x$ と書けます。以上を整理すると、答えは次の条件を満たす整数の組 (x, y) の個数です。

- $L \leq x \leq y \leq R$
- x と y の MSB の位置が同じである。
- $y - x = x$ XOR y

$y - x = x$ XOR y という条件は、$y - x$ という二進数の引き算において繰り下がりが起こらないことと同値であり、「どの整数 i に対しても、y の 2^i の位の桁が 0 であって x の 2^i の位の桁が 1 であるということはない」と書き換えられます (このとき $x \leq y$ は無条件に満たされ、この不等式は以後考慮する必要がなくなります)。あとはこの条件に加えてさらに「$L \leq x$」「$y \leq R$」「x と y の MSB の位置が同じ」 という三つの条件を満たす二本の 01 列を数える問題となり、これは動的計画法により $O(\log Y)$ 時間で解けます。詳細は実装例[*5]を参照してください。

[*4] y を x で割った商を q, 余りを r とすると $y = xq + r > r + r = 2r$ であるため

[*5] `f(pos, flagX, flagY, flagZ)`: x, y の 2^{pos} 以上の桁がすべて確定しており、そこまでで $L < x$ であることが確定しており (おらず)、$y < R$ であることが確定しており (おらず)、両者の MSB がすでに現れている (いない) 場合の残りの桁の決め方の数

ABC 138 Editorial

camypaper, evima, sheyasutaka, tozangezan, yosupo

August 18, 2019

A: Red or Not

(Original writer: tozangezan, preparation and editorial: evima)

This problem is like problem A of practice contest, but instead of addition, you are asked to write a conditional statement. For the ways of input and output, please refer to the sample code there, and then you can add a process to change the output depending on which of a and 3200 is larger, combining the if statement or conditional operator (e.g. `?:`) with comparison operator (e.g. `>=`), and that's it. (Sorry to end up with abstract explanation, but the margin of PDF is too narrow to explain the ways of each languages. Please make use of Google search)

The following is an implementation example in Python(3).

```
a = int(input())
s = input()
print(s if a >= 3200 else 'red')
```

B: Resistors in Parallel

(Original writer: sheyasutaka, preparation and editorial: evima)

In the explanation in sample input, $\frac{1}{A_1} + \frac{1}{A_2} + \ldots + \frac{1}{A_N}$ is calculated in fractional notation, but it would be easier to calculate using floating point type all the way. Then implementation can be done with looping structure like for-statement or something. On behalf of many languages, here are implementation examples by C++ and Python(3).

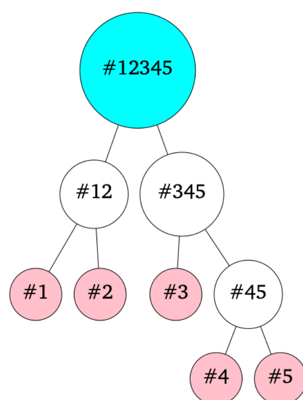
```cpp
#include <iomanip>
#include <iostream>
using namespace std;

int main(){
    int N;
    double ans = 0.0, A;
    cin >> N;
    for(int i = 1; i <= N; ++i){
        cin >> A;
        ans += 1.0 / A;
    }
    ans = 1.0 / ans;
    cout << setprecision(16) << ans << endl;
}
```

```python
N = int(input())
A = map(int, input().split())
print('{:.16g}'.format(1 / sum(1 / x for x in A)))
```

C: Alchemist

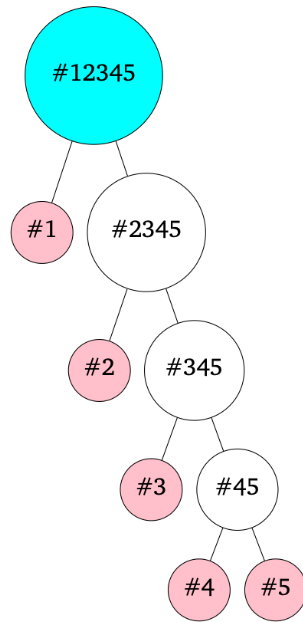
(Original writer: yosupo, preparation and editorial: evima)

Let's assume that ingredients $1, \ldots, N$ are indexed in the decreasing order of value (you can first sort the ingredients and change the indices in such way), and represent the process of composition with tree structures like the following diagram (where #1 denotes ingredient 1, and #12 denotes the ingredient that can be obtained by composing ingredients 1 and 2).



Let d_i be the distance from the leaf that denotes ingredients to the root (the number of edges between them), then the value of the final ingredients will be $v_1/2^{d_1} + \ldots + v_N/2^{d_N}$. To obtain the maximum result, you only have to think about the composition process such that $d_1 \leq d_2 \leq \ldots \leq d_N$ (otherwise, you can swap some indices of ingredients so that the total result will increase).

Moreover, to obtain the maximum result, it should be that $d_1 = 1, d_2 = 2, \ldots, d_{N-1} = d_N = N - 1$. This is because, briefly speaking, if you have a composition process such that $d_1 \leq d_2 \leq \ldots \leq d_N$ other than that (for example, like the diagram above), then you can increase the final result by *transplanting the violating "left-side child" nearest to the root under the "right-side child" farthest from the root.*


#12345
#1
#2345
#2
#345
#3
#45
#4
#5

D: Ki

(Original writer: tozangezan, preparation and editorial: evima)

An operation can affect at most all the N *vertices, so if you simulate the operations naively, you will have to apply addition to the counters* NQ *times in total, and it won't fit in the time limit.* [*1]

For the tree-like problem like this, sometimes it is helpful to think about linear tree like $1-2-3-\ldots-N$ *instead of general tree, and this problem is a good example. Assuming that the graph is linear, a linear-time solution like the following is available:*

1. *Let* $c_1, c_2, \ldots, c_N = 0, 0, \ldots, 0$.
2. *For each operation* j, *add* x_j *to* c_{p_j}.
3. *For each* $i = 2, \ldots, N$, *in this order, add* c_{i-1} *to* c_i.
4. *Output* $c_1, c_2, \ldots, c_N$.

And this solution can be also applied to trees. Specifically, if you replace the c_{i-1} *in the third step with* c_{q_i} *(where* q_i *is parent of node* i*),* ~~*it will directly be the solution for the original problem.*~~ *and do the operations from the vertices nere to the root, the original problem can be solved. (8/19 0:05 JST corrected)* [*2] [*3]

[*1] If both N and Q are at maximum of 200 thousands, $NQ = 40$ billion times. Even fast languages like C++ can perform addition at most several hundreds million time, and when it comes to tens of millions of times, it's hopeless

[*2] ~~Note that this algorith, relies on the constraints $a_i < b_i$. Without this constrains,~~ For each i $(1 \leq i \leq N-1)$, vertex a_i may not be the parent of b_i, (8/19 0:05 corrected) you have to explicitly perform the operations from the root to children, with recurrence functions or something

[*3] From this problem on, depending on programming languages, even the "correct" solution may not be fast enough to meet the time limit. If you tackle competitive programming seriously, we recommend you think of altering the programming language

E: Strings of Impurity

(Original writer: sheyasutaka, preparation and editorial: evima)

Generally, in order to judge if a string t is a substring of u, you can greedily match t with u. That is, find the first appearance of the first letter of t in u, then find the second letter of t in u after the first match, ... and so on, until checking if the last letter of t appears in u, and this time, it is required to do this operations efficiently under $\mathrm{u} = s'$ (an infinite repetition of s).

The key point is that, while s' is infinitely long, the length of t is limited, so it is sufficient to perform each operation of $|t|$ operations mentioned above quickly enough. To do so, for each letter ch from `a` to `z`, you should extract all the appearance of ch in s. Then you can precalculate $\mathrm{next}_{\mathrm{ch,i}}$*: "the first appearance of letter ch after the i-th character of s'" for each $i = 0, \ldots, |s| - 1$ with* $\mathrm{O}((\mathrm{thenumberofkindsofcharacters}) \times |\mathrm{s}|)$ *time, or use binary search to find the position of the next appearance of ch with* $\mathrm{O}(\log |s|)$ *time each. Anyway, if you use 2-time repetition of s, the implementation will be easier.*

F: Coincidence

(Original writer: camypaper, preparation and editorial: evima)

Let y's MSB (Most Significant Bit) be m (in other words, 2^m's place of y is 1, and for any integer n ($n > m$), 2^n's place of x is 0). Here, assuming that 2^m's place of x is 0, then by the definition of XOR, 2^m's place of x XOR y will be 1. However, generally for any integers x, y ($y \geq x$), the remainder of y divided by x is less than half of y[*4], which means that the MSB of remainder of y divided by x is less than m, so the remainder of y divided by x does never corresponds to x XOR y. Therefore, we can always assume that x's MSB is also m. Here, the quotient of y divided by x is 1, so the remainder of y divided by x can simply be denoted as $y - x$. The answer is the number of pairs of integers (x, y) that meets the following conditions.

- $L \leq x \leq y \leq R$
- The MSB position of x and y are the same.
- $y - x = x$ XOR y

The condition $y - x = x$ XOR y is equivalent to the condition that there are no borrowings in binary subtraction of $y - x$, and in other words, "for any integer i, it will never occur that 2^i's place of y is 0 and 2^i's place of x is 1 (in such case, $x \leq y$ always holds, and you don't have to care about it anymore). The remaining task is to count the number of pairs of 01-sequences that meets the three conditions, "$L \leq x$", "$y \leq R$" and "the MSB position of x and y are the same", and this can be solved in a total of $O(\log Y)$ time with DP. For details, please refer to the implementation example[*5].

[*4] Let q be the quotient of y divided by x, and r be the remainder, then $y = xq + r > r + r = 2r$ holds

[*5] `f(pos, flagX, flagY, flagZ)`: The number of ways to determine the remaining digits where the 2^{pos}'s place and higher of x and y is already determined, it's already (still not) confirmed that $L < x$, it's already (still not) confirmed that $y < R$, and their MSB has already (still not) been appeared