ABC 095 / ARC 096 Editorial (Japanese)

問題・解説: evima

2018 年 4 月 21 日

For International Readers: English editorial starts on page 7.

A: Something on It

この問題を解くには、以下の手順を踏む必要があります。

0. （言語によっては不要）文字列変数 S を宣言する。
1. 標準入力から S を文字列として受け取る。
2. S から何らかの方法で答えとなる整数を得る。
3. 求めた値を標準出力に出力する。

手順 0, 1, 3 については、practice contest の問題 A の問題別サンプルコードが参考になる（少し書き換えれば使える）でしょう。手順 2 については、S の 3 つの文字それぞれを if 文でチェックするか、もしくは言語の機能を用いて S に含まれる `o` の個数を数えることになります。前者の C++ による実装例と後者の Python(3) による実装例を示します。

```cpp
#include <iostream>
using namespace std;
int main(){
    string S;
    cin >> S;
    int ans = 700;
    if(S[0] == 'o') ans += 100;
    if(S[1] == 'o') ans += 100;
    if(S[2] == 'o') ans += 100;
    cout << ans << endl;
}
```

```python
print(700 + 100 * input().count('o'))
```

B: Bitter Alchemy

入力例の説明でほのめかされているように、「まずすべての種類のドーナツを 1 個ずつ作り、残ったお菓子の素で最も『安い』ドーナツを作れるだけ作る」という方針で問題ありません。つまり、$m_1, m_2, ..., m_N$ の和を S、$m_1, m_2, ..., m_N$ のうち最小の値を M として、答えは $N + \lfloor (X - S)/M \rfloor$ となります。[*1]主な課題は、$m_1, m_2, ..., m_N$ の値を読み込むことと、S や M の値を実際に求めることです。なお、割り算と小数部分切り捨ての部分に関しては、多くのプログラミング言語では単に `A / B` と書くことで $\lfloor A/B \rfloor$ が求められます。[*2]

言語によってやや事情が異なるかもしれません。C++, Java などの言語では、ループ構造を用いるのが素直でしょう。C++ での実装例を挙げます。

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

int main(){
    int N, X, m[101];
    cin >> N >> X;
    int S = 0, M = 1001001001;
    for(int i = 0; i < N; ++i){
        cin >> m[i];
        S += m[i];
        if(m[i] < M){
            M = m[i];
        }
    }
    int ans = N + (X - S) / M;
    cout << ans << endl;
}
```

Python, Ruby といった言語でももちろん上のようなアプローチが可能ですが、こういった言語では `sum`, `max` といった関数がはじめから用意されていることがあり、便利です。[*3]これを用いた Python(3) での実装例を次ページに挙げます。

[*1] $\lfloor x \rfloor$ は x の整数部分を表します。
[*2] A, B が正でない場合はこの限りでなく、またここに実装例を掲載した Python3 はこれに該当しません。
[*3] このような機能のため、AtCoder Beginner Contest で出題されうる程度の問題の解法を手っ取り早く実装する上で、スクリプト言語はとても役立ちます。ただし、この範囲を超えると、どうしても実行速度が足りない場面が急速に増えます。競技プログラミングに本格的に取り組む場合、C++ とは言わずとも Java に匹敵する実行速度を有する言語を一つ習得する必要がある、ということをお伝えしておきます。

```python
N, X = map(int, input().split())
m = []
for i in range(N):
    m.append(int(input()))
print(N + (X - sum(m)) // min(m))
```

C: Half and Half

まず、AB ピザを奇数枚買って 1 枚余らせるのは無意味なので、AB ピザは 2 枚 1 組で考えます。つまり、$2C$ 円で A ピザ 1 枚と B ピザ 1 枚を買えると考えることにし、この 2 枚の組み合わせをこれ以降 AB セットと呼ぶことにします。

この問題の重要な制約は、$X, Y \leq 10^5$ です。これは、A ピザ、B ピザ、AB セットのどれについても、買う個数は 10 万個以下でよいことを意味します。したがって、この三種類のうちどれか一種類を選んで、それを買う個数を 0 から 10 万まですべて試す、という方針が考えられます。[*4]

さて、三つのうちどれについて購入個数を全列挙するべきでしょうか？結論を述べると、AB セットです。AB セットを i 個 $(0 \leq i \leq 10^5)$ 購入した場合、$i \geq X$ であれば A ピザを買い増す必要はなく、$i < X$ であれば A ピザを $X - i$ 枚買い増す必要があります。これらをまとめて、買い増すべき A ピザの数は $\max(0, X - i)$ 枚であるということもできます。同様に、B ピザを $\max(0, Y - i)$ 枚買い増す必要があります。以上から、AB セットを i 枚購入した場合の所要金額は $i \times 2C + \max(0, X - i) \times A + \max(0, Y - i) \times B$ 円であり、この値を 0 以上 10 万以下のすべての整数 i について計算して最小値を取ることで答えが求まります。

なお、定数時間で答えを求めることもできます。練習問題として解説は省きます。

[*4] なお、二種類を選んでしまうと考慮する可能性の数が約 100 億通りになり、2018 年現在の一般的な計算機が 2 秒で探索するには厳しいです。今日の一般的な計算機が 1 秒あたりに処理できる式の数はおよそ 1 億個といったところで、短い単純な式であれば 10 億個程度まで伸び、逆に実数の割り算など複雑な演算を伴う場合は数千万個程度に落ちます。

D: Static Sushi

中橋君の靴にペンキが塗られていて、通ったところの床にペンキが塗られると想像してください。最終的に、ペンキが塗られた区間に含まれる寿司をすべて食べることになります。

初期位置を O とし、退店するときにペンキが塗られている区間を円弧 AB とします（O から時計回りに進んだときに先に到着する方の円弧の端を A とします。なお、円周全体が塗られる場合は最適でないため考えません)。点 A と B の片方もしくは両方（即座に退店したとき）が O と一致する場合もあります。すると、最適な歩き方は、「A まで時計回りに歩き、方向転換して反時計回りに B まで歩いて退店」もしくは「B まで反時計回りに歩き、時計回りに A まで歩いて退店」となります。

前者での移動距離は $OA + AB = 2OA + OB$、後者での移動距離は同様に $OA + 2OB$ となります。A と B を定めたとき、$OA \leq OB$ のとき前者、$OA \geq OB$ のとき後者の歩き方をするのが最適となりますが、実はそれを意識する必要はあまりなく、前者の歩き方をすると決めたときの最適解と後者の歩き方での最適解を別々に求め、よりよい方を答えとするのが単純です。以下では、後者の歩き方での最適解を求めます。前者の場合も同様に求められます。

解を求めるには、もう一つの観察が必要です: A, B はどちらも、寿司のある位置か初期位置のいずれかであるべきです。そうでなければ、含まれる寿司を減らすことなく円弧を縮められるためです。よって、$N \leq 100$ の部分点のためのテストセットは、A, B の位置をそれぞれ全探索することで解けます。愚直に実装すれば $\mathrm{O}(N^3)$ 時間、$v_1, v_1 + v_2, ..., v_1 + v_2 + ... + v_N$ や $v_N, v_N + v_{N-1}, ..., v_N + v_{N-1} + ... + v_1$ を事前に計算しておけば $\mathrm{O}(N^2)$ 時間の解法が得られます。

$N \leq 10^5$ の満点用テストセットも、これを少し改善することで解けます。A, B のうちどちらかの位置を全て試す時間はあるため、ここでは B の位置をすべて試すことにします。B の位置を x_b に固定したとき、A の位置は $x_1, x_2, ..., x_{b-1}$ のいずれかであり、このうち $f(a) := v_1 + v_2 + ... + v_a - x_a$ が最大となるような a に対応する x_a を選択するべきです（差し引きの摂取カロリーは、$f(a)$ に $v_N + v_{N-1} + ... + v_b - (C - x_b)$ を加えたものですが、この部分は a に依存せず一定値をとるためです)。よって、事前に $f(0), f(1), ..., f(N)$ の値を求めておき、さらに $g(a) := \max(f(0), f(1), ..., f(a))$ を $a = 0, 1, ..., N$ に対して計算しておけば（$g(0)$ から順に求めます)、B の位置を固定したときに直ちに最適な A の位置を選ぶことができ、$\mathrm{O}(N)$ 時間の解法が得られます。

E: Everything on It

まず、包除原理 (Wikipedia) を適用しないことには進展はないでしょう。トッピングの対称性から、答えを A として次が得られます。[*5]

$$A = \sum_{i=0}^{N} (-1)^i C(N,i) ways(i)$$

ここで $C(N,i)$ は二項係数であり、$ways(i)$ は何杯かのラーメンの組み合わせであって、トッピング $1,2,...,i$ がいずれも 1 杯以下にしか乗っていないようなものの個数です（トッピング $i+1, i+2, ..., N$ については不問)。各 $ways(i)$ を $O(N)$ 時間で求めることができれば、満点が得られます。

$j = 0, 1, .., i$ のそれぞれに対して、上の条件を満たすようなラーメンの組み合わせであって、トッピング $1,2,...,i$ のうち一つ以上が乗っているラーメンがちょうど j 杯あるようなものの個数（$ways2(i,j)$ とします）を $O(1)$ 時間で求めることができれば、それらを足し合わせることで $O(N)$ 時間で $ways(i)$ が求まります。そして、これは可能です。

このようなものを数える上で最も重要な「部品」は、トッピング $1,2,...,i$ がどのように分割されるか、すなわち、「i 個の区別可能な物体から一部を選んで j 組の空でないグループを作る方法の数（同じ物体は複数組に属せないが、ゼロ組に属することは可能)」です（下記の図を参照してください)。これは、第二種スターリング数 (Wikipedia) に非常に近い概念で、それと同様に動的計画法による $O(N^2)$ 時間の事前計算により必要なすべての値を求めることができます。ここで、i 個のトッピングのうち何個が実際にラーメンに乗っているか、はさほど重要でないことに注意してください（この数で場合分けをしてしまうと、解法の所要時間が $O(N^3)$ になってしまいます)。

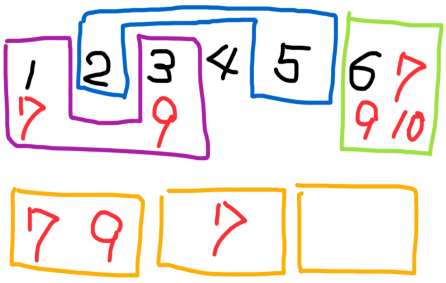

図: $N = 10$ のとき $ways2(6,3)$ に数えられるラーメンの組み合わせの例（各枠がラーメン一杯に対応）

これに、残りのトッピング $i+1, i+2, ..., N$ を考慮して $2^{(N-i)j}$ や $2^{2^{N-i}}$ を掛け合わせることで $ways2(i,j)$ を定数時間で求めることができ（上記の他にも事前計算が必要です)、目標が達成されます。

[*5] この文と次の式の意味がよく分からない方は、$N = 3$ や 4 のケースで包除原理をそのまま適用した結果を観察してみてください。ここでの「対称性」という単語の意味は「交換可能」といったところです。

F: Sweet Alchemy

$d_i = c_i - c_{p_i}$ とします。例外として $i = 1$ (根) のとき $d_i = c_i$ とします。
これを用いると、次のように言い換えられます：

- 各 $i > 1$ について $0 \leq d_i \leq D$ が成り立つ。d_1 は任意の非負整数である。
- 各 i について以下を d_i 回行う: i を根とする部分木内の各頂点 j に対し、ドーナツ j を一個作る。
- 上の条件の下でドーナツの個数を最大化せよ。

これはさらにナップザック問題に言い換えられます。頂点 v を根とする部分木内の頂点を $w_1, \cdots, w_k$ とすると、この部分木に対する操作を行うことは $Y_v := k$ 個のドーナツを $X_v := m_{w_1} + \cdots + m_{w_k}$ グラムの素を使って作ることと同値です。これを重さ X_v 価値 Y_v のアイテムとみなします。
以下を解きたいです：

- N 種類のアイテムがある。
- i 種類めの重さは X_i 価値は Y_i である。
- 重さの合計が X 以下、種類 i を選ぶ個数は Z_i 以下となるようにするとき、価値の合計の最大値を求めよ。
- Constraints: $N \leq 50, Y_i \leq 50$. 他の値は大きい。

直感的には、効率の良いものから greedy にとるべきです。アイテムが効率の良い順にソートされているとします ($Y_1/X_1 \geq Y_2/X_2 \geq \cdots$)
これを厳密に述べると、二種類のアイテム p, q $(p < q)$ を考えます。アイテム p が 50 個以上選ばれずに残り、またアイテム q が 50 個以上選ばれずに残っているとします。このとき、アイテム q を Y_p 個捨てて、アイテム p を Y_q 個新たに選ぶことで、価値の総和を増やすことができます。したがって、このような組 p, q は存在しないと仮定することができます。
これにより、次が示されます：二つの袋があり、各 i に対して、アイテム i のうち $min(50, Z_i)$ 個を一つ目の袋に入れ、残りのアイテム i を二つ目の袋に入れます。このとき、最適解においては、一つ目の袋に入っているアイテムから一部を選び、二つ目の袋からはアイテムを単に greedy に（効率の良い順に）選ぶべきです。
あとは簡単です。一つ目の袋に入っているすべてのアイテムの価値の総和は N^3 以下です。各 $0 \leq Y \leq N^3$ に対し、一つ目の袋から選んだアイテムの価値の総和が Y であると仮定して、それらのアイテムの重さの総和を最小化します（これは $O(N^4 \log N)$ 時間の単純な DP で可能です）。そして、二つ目の袋からさらにアイテムを貪欲に（重さの上限に達するまで）選んでいきます。
この解法は $O(N^4 \log N)$ 時間で動作します。

ABC 095 / ARC 096 Editorial (English)

evima

April 21, 2018

A: Something on It

To solve this problem, you need to go through the procedure below:

0. (Unnecessary for some languages) Declare a string variable S.
1. Receive S as a string from Standard Input.
2. Obtain the answer from S in some way.
3. Print the found value to Standard Output.

For steps 0, 1 and 3, the sample code for each language for Problem A in practice contest would be helpful (can be actually used after modifying a bit). For step 2, you would need to check each of the three characters in S using if statements, or count the number of `o` in S using some feature in your language. The former approach in C++ and the latter approach in Python(3) are shown below:

```cpp
#include <iostream>
using namespace std;
int main(){
    string S;
    cin >> S;
    int ans = 700;
    if(S[0] == 'o') ans += 100;
    if(S[1] == 'o') ans += 100;
    if(S[2] == 'o') ans += 100;
    cout << ans << endl;
}
```

```python
print(700 + 100 * input().count('o'))
```

B: Bitter Alchemy

As implied in the notes for the sample inputs, the following strategy works fine: "first make one doughnut for each kind, then make as many 'cheapest' doughnuts as possible". That is, the answer is $N + \lfloor (X - S)/M \rfloor$ [*1], where S is the sum of $m_1, m_2, ..., m_N$, and M is the minimum value among $m_1, m_2, ..., m_N$. The main challenge is to read the values of $m_1, m_2, ..., m_N$, and actually find the values of S and M. For the "division and rounding down" part, $\lfloor A/B \rfloor$ can be found by just writing `A / B` in many languages [*2].

The situation may be different for different languages. In languages such as C++ and Java, it would be straightforward to use loop structures. A sample implementation in C++ is shown below:

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

int main(){
    int N, X, m[101];
    cin >> N >> X;
    int S = 0, M = 1001001001;
    for(int i = 0; i < N; ++i){
        cin >> m[i];
        S += m[i];
        if(m[i] < M){
            M = m[i];
        }
    }
    int ans = N + (X - S) / M;
    cout << ans << endl;
}
```

The above approach is also possible in languages such as Python and Ruby, but in these languages there are occasionally built-in functions such as `sum` and `max`, which are convenient. [*3] A sample implementation in Python(3) using these functions is shown in the next page:

[*1] $\lfloor x \rfloor$ represents the integer part of x.

[*2] Not necessarily true when A or B is not positive. Also, Python3, which is used here for the sample implementation, is an exception of this statement.

[*3] Because of these functionalities, script languages are very useful in quickly implementing the solution of problems of difficulties suited for AtCoder Beginner Contest. However, for harder problems, there are suddenly many situations where these languages are simply too slow. I'm letting you know that, if you intend to do competitive programming seriously, you would need to learn one language that can rival Java for speed, if not C++.

```python
N, X = map(int, input().split())
m = []
for i in range(N):
    m.append(int(input()))
print(N + (X - sum(m)) // min(m))
```

C: Half and Half

First, since buying an odd number of AB-pizzas and having one leftover is nonsense, we will always buy these in pairs. That is, we will assume that we can buy one A-pizza and one B-pizza for $2C$ yen, and we will call this set an AB-set.

The important constraint in this problem is $X, Y \leq 10^5$. This means that we need to buy at most 100000 A-pizzas, at most 100000 B-pizzas and at most 100000 AB-sets. Thus, we can select one of these three menus and try every possible number of that menu purchased from 0 to 100000. [*4]

Which among the three should we focus on? Our conclusion is AB-sets. When we buy i AB-sets ($0 \leq i \leq 10^5$), we don't need to buy extra A-pizzas if $i \geq X$, and we need to buy $X - i$ extra A-pizzas if $i < X$. In other words, we should buy $\max(0, X - i)$ extra A-pizzas. Similarly, we should buy $\max(0, Y - i)$ extra B-pizzas. Therefore, we need $i \times 2C + \max(0, X - i) \times A + \max(0, Y - i) \times B$ yen if we buy i AB-sets, and the answer can be found by computing this value for every integer i from 0 to 100000, and taking the minimum obtained value.

A constant time solution is also possible, which is left as an exercise.

[*4] If we select two, there are about 10 billion possibilities to consider, which is too much for an ordinary computer today to enumerate in 2 seconds. A rough guess of the number of formula that an ordinary computer today can process in 1 second is about 100 million. This number can grow up to about 1 billion if the formula is a short, simple one. On the other hand, it can decline down to about several ten millions if the formula involves complex operation such as division of real numbers.

D: Static Sushi

Imagine that his shoes are wet with paint, and he paints the floor where he walks. In the end, he eats all sushi in the painted segment.

Let the initial place be O, and the painted segment in the end be arc AB (If we walk clockwise from O, we reach A first. We ignore the case where the whole counter is painted, which is not optimal). Note that A and/or B may coincide with O. The optimal way to walk is to "walk clockwise until A, then turn around and walk counterclockwise until B", or "walk counterclockwise until B, then turn around and walk clockwise until A".

The distance covered in the first way of walking is $OA + AB = 2OA + OB$, and $OA + 2OB$ in the second way. When A and B is fixed, it is optimal to perform the first way if $OA \geq OB$, and the second way if $OA \geq OB$, but we actually don't have to care this too much. We just need to find the optimal solution when we stick to the first way, and the optimal solution when we stick to the second way, and take the better one. We will deal with the second way from now on (the first way can be dealt similarly).

One more observation is required to find the solution, both A and B must be a position where a sushi is placed or O. This is because the arc could be shortened without losing sushi otherwise. Thus, the partial test set can be solved with trying every possible choice of A and B. This results in an $\mathrm{O}(N^3)$ time solution if implemented naively, and $\mathrm{O}(N^2)$ if $v_1, v_1+v_2, ..., v_1+v_2+...+v_N$ and $v_N, v_N+v_{N-1}, ..., v_N+v_{N-1}+...+v_1$ are precomputed.

The full test set can be solved with just a little improvement from here. There is enough time to try every choice of A or B, and we will go with B here. When B is fixed as x_b, A is one of $x_1, x_2, ..., x_{b-1}$, and we should select x_a corresponding to a that maximizes $f(a) := v_1 + v_2 + ... + v_a - x_a$. (This is because, the calories taken in is equal to f(a) plus $v_N + v_{N-1} + ... + v_b - (C - x_b)$, which has a constant value independent of a when b is fixed). Thus, if we find $f(0), f(1), ..., f(N)$ beforehand, and find $g(a) := \max(f(0), f(1), ..., f(a))$ for each $a = 0, 1, ..., N$ (starting from $g(0)$), we can immediately make the optimal choice of A when B is fixed, thus we have an $\mathrm{O}(N)$ time solution.

E: Everything on It

We have no choice but to apply inclusion-exclusion principle (Wikipedia). Let A be the answer. From the symmetry of the toppings, the following holds:[*5]

$$A = \sum_{i=0}^{N} (-1)^i C(N, i) ways(i)$$

Here, $C(n, i)$ is binomial coefficients, and $ways(i)$ is the number of the sets of some ramen such that each of the toppings $1, 2, ..., i$ is on at most one of those ramen (we don't care toppings $i+1, i+2, ..., N$). If we can find each $ways(i)$ in $O(N)$ time, we can obtain the full score.

For each $j = 0, 1, .., i$, if we can find the number of those sets of ramen that satisfy the condition above such that there are exactly j ramen topped with one or more toppings among toppings $1, 2, ..., i$ (let this number be $ways2(i, j)$) in $O(1)$ time, we can find $ways(i)$ in $O(N)$ by summing them up. This is indeed possible.

The most important "part" for counting these entities is, how the toppings $1, 2, ..., i$ are divided, that is, "the number of ways to make j non-empty groups out of i distinguishable objects (an object can't belong to multiple groups, but can belong to zero group)". This is a very similar concept to Stirling numbers of the second kind (Wikipedia), and all the required values can be computed in $O(N^2)$ time beforehand by dynamic programming. Note that it does not really matter how many of the i toppings are actually used in some ramen (if we classify the sets according to this number, the solution would take $O(N^3)$ time).

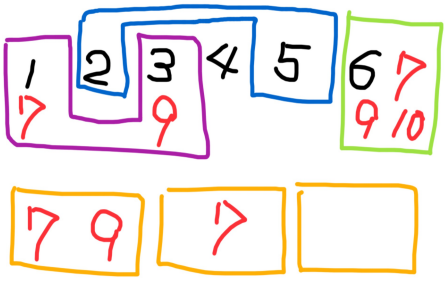

Figure: One example of a set of ramen counted in $ways2(6, 3)$ where $N = 10$ (Each frame corresponds to a ramen)

We then multiply this by $2^{(N-i)j}$ and $2^{2^{N-i}}$, consider the remaining toppings $i+1, i+2, ..., N$, and we can find $ways2(i, j)$ in a constant time (other pre-computations are also required), which was our goal.

[*5] If you are not sure of this sentence and the following formula, please try observing the direct application of inclusion-exclusion principle to the case $N = 3$ or 4. The meaning of the word "symmetry" here is "interchangable".

F: Sweet Alchemy

Let $d_i = c_i - c_{p_i}$. As an exception, when $i = 1$ (the root of the tree), we define $d_i = c_i$.

Using this, the problem can be restated as follows:

- For each $i > 1$, $0 \leq d_i \leq D$ must hold. d_1 can be an arbitrary non-negative integer.
- For each i, we do the following d_i times: for each node j in the subtree rooted at j, make one Doughnut j.
- Under the conditions above, compute the maximum number of Doughnuts you can make.

This can be further restated as a knapsack problem. Consider a node v, and let $w_1, \cdots, w_k$ be the nodes in the subtree rooted at v. Then, performing an operation for this subtree is equivalent to making $Y_v := k$ doughnuts using $X_v := m_{w_1} + \cdots + m_{w_k}$ grams of Moto. Regard it as an item with weight X_v and value Y_v.

Now we want to solve the following problem:

- There are N types of items.
- The weight of an i-th type of item is X_i. Its value is Y_i.
- You want to choose some items such that the total weight is at most X. You can choose at most Z_i items of type i. What is the maximum total value you can achieve?
- Constraints: $N \leq 50, Y_i \leq 50$. Other parameters can be very large.

Intuitively, we should choose the most "efficient" ones greedily. For simplicity, assume that the items are sorted in the decreasing order of efficiency: that is, $Y_1/X_1 \geq Y_2/X_2 \geq \cdots$.

Now, we formally state the intuition. Consider two types of items p, q ($p < q$). Assume that there are at least 50 unchosen items of type p and at least 50 chosen items of type q. Then, we can improve the value by taking Y_q itmes of type p and discarding Y_p items of type q. Thus, we can assume that there are no such pairs of p, q.

This proves the following: We have two bags of items. For each i, we put $min(50, Z_i)$ items of type i to the first bag, and the remaining items of type i to the second bag. In the optimal solution, we should choose some subset of items from the first bag, and from the second bag we just choose items greedily (by efficiency).

The remaining part is easy. The total value of all items in the first bag is at most N^3. For each $0 \leq Y \leq N^3$, assume that the total value of all items you choose from the first bag is Y, and minimize their total weight (this can be done by a simple DP in $O(N^4 \log N)$ time). Then, we choose more items from the second bag greedily (until we reach the weight capacity).

This solution works in $O(N^4 \log N)$ time.